CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2009)
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1387

Multiversion concurrency control for the generalized search tree

Walter Binder[1,*,†], Adina Mosincat[1], Samuel Spycher[2], Ion Constantinescu[3] and Boi Faltings[2]

[1]*Faculty of Informatics, University of Lugano, CH-6900 Lugano, Switzerland*
[2]*Ecole Polytechnique Fédérale de Lausanne (EPFL), CH-1015 Lausanne, Switzerland*
[3]*Founder Opt.im - Digital Optim LLC, U.S.A.*

SUMMARY

Many read-intensive systems where fast access to data is more important than the rate at which data can change make use of multidimensional index structures, like the generalized search tree (GiST). Although in these systems the indexed data are rarely updated and read access is highly concurrent, the existing concurrency control mechanisms for multidimensional index structures are based on locking techniques, which cause significant overhead. In this article we present the multiversion-GiST (MVGiST), an in-memory mechanism that extends the GiST with multiversion concurrency control. The MVGiST enables lock-free read access and ensures a consistent view of the index structure throughout a reader's series of queries, by creating lightweight, read-only versions of the GiST that share unchanging nodes among themselves. An example of a system with high read to write ratio, where providing wait-free queries is of utmost importance, is a large-scale directory that indexes web services according to their input and output parameters. A performance evaluation shows that for low update rates, the MVGiST significantly improves scalability w.r.t. the number of concurrent read accesses when compared with a traditional, locking-based concurrency control mechanism. We propose a technique to control memory consumption and confirm through our evaluation that the MVGiST efficiently manages memory. Copyright © 2009 John Wiley & Sons, Ltd.

Received 10 May 2007; Revised 4 August 2008; Accepted 26 August 2008

KEY WORDS: concurrency control; multidimensional index structures; generalized search tree (GiST); web service directories; multiversion concurrency control

*Correspondence to: Walter Binder, Faculty of Informatics, University of Lugano, Via Giuseppe Buffi 13, CH-6900 Lugano, Switzerland.
†E-mail: walter.binder@unisi.ch

Contract/grant sponsor: The Swiss National Science Foundation

1. INTRODUCTION

In the last decade vast research in information systems has concentrated around two key issues: concurrency control and manipulation of data with transactional semantics. As a result, different solutions have been proposed, some generic, others aimed at specific use-cases.

The feature every efficient concurrency control technique has to take into consideration is that information systems differ according to their usage pattern, which makes a concurrency control technique developed for one kind of system not applicable to another. Based on the read/write ratio, there are two major kinds of information systems: first, there are databases, where essentially the frequency with which data in the system are changed is comparable to the number of data reads. And second, there are systems such as on line analytical processing (OLAP) tools [1], where fast access to multiple views of multidimensional data is more important than the rate at which data can change, or directories [2] and content repositories [3]. For modern databases, the most successful concurrency control techniques to date rely on multiversion concurrency control (MVCC) [4], which allows read-only transactions to execute without any need for synchronization with read–write transactions. The price to pay is a trade of space requirements for speed. OLAP-like systems need to provide the same independence of read-only transactions. They often have an extremely high read to write ratio, usually make heavy usage of index structures, and make the same trade-off regarding space and speed.

In this article we present an in-memory mechanism aimed at providing concurrency control for OLAP-like systems, proposing an MVCC strategy through which transactional semantics can be efficiently supported within multidimensional, tree-based data indexes. While the target of our solution is data indexes residing in memory, these indexes can also provide an easy navigation means to persistent data, as it is possible that the leaf nodes reference records that are on secondary storage.

There have been many concurrency control techniques developed, but mostly for one-dimensional index structures such as B-trees [5,6]. Kornacker *et al.* [7] introduced a concurrency control technique for multidimensional index structures like the generalized search tree (GiST) [8] based on the link technique for B-trees (see Section 6.1); however, their approach makes use of multiple locks, which may restrict the level of concurrency. In highly concurrent systems with a great number of read operations, like the ones we are targeting, this restriction is not acceptable.

Furthermore, to the best of our knowledge, all existing solutions for concurrency control in multidimensional indexes are based on locking, which means that even read-only transactions will have to wait [9–11]. As our goal is to provide fast concurrent access to readers, the focus lays on avoiding locking by read-only transactions. Our solution achieves this goal by providing read-only snapshots and so ensuring wait-free queries.

Our system is built on two well-known concepts, MVCC and the GiST:

First, we propose an algorithm based on MVCC for concurrent access to the index structure. MVCC is a technique that manages access to shared data by replicating and versioning it where needed. By replicating data upon modification, read accesses are isolated from updates to the index. Therefore, reads have a consistent and unchanging view of the data and do not interfere with update processes through locking of individual nodes, such as in other concurrency control schemes. While visibility of updated data can be delayed and memory consumption increased, feedback on successful update is immediate, and the fact that read-only access is non-blocking is a crucial advantage for many applications.

In order to limit the drawback of an increased memory consumption, we use the common practice of timeouts to control memory consumption. Whereas in the majority of information systems, query and connection timeouts are used for performance and security reasons, our solution benefits from timeouts also in terms of memory consumption control.

Our implementation of MVCC provides for a fixed snapshot of the index across multiple read operations. The reader is essentially free to request a new snapshot at any time if available, or retain the same snapshot for as many read operations as she/he wishes up to a specified timeout constraint.

Second, the index structure we chose to implement our MVCC design on is the GiST [8]. The GiST is a balanced tree, which contains algorithms for navigating as well as modifying the tree structure. The tree stores keys and records references in its leaf nodes, and the inner nodes contain predicates and references to their child nodes. These predicates evaluate true for any key in their child nodes. This hierarchy of predicates is essentially what is common to all tree-based index structures. The GiST itself is, however, not a fully implemented search tree. It is a generic structure that allows its user to define the data types to be stored in the tree, as well as the query predicates with which the data tuples can be inserted and retrieved. The advantage of this is that the GiST provides a 'template' index structure for most of the tree-based access methods, making it easier to integrate these index structures into databases. Notably, the GiST has been integrated into PostgreSQL [12] and is used e.g. in the PostGIS project [13], a spatially enabled database for geographic information systems.

In this article we present the Multiversion-GiST (MVGiST), a concurrent index structure based on MVCC and the GiST. The features that the MVGiST inherits from the GiST, flexibility and query capabilities, are successfully combined with the high read concurrency offered by MVCC. Moreover, the MVGiST presents the reader with an unchanging view of the data allowing for consistency across multiple queries. This article includes a thorough evaluation of the MVGiST, which we implemented in Java because our use-case is a Java-based system and we take advantage of Java features such as automated memory management and multithreading. We show the efficiency of our MVGiST implementation compared with a locking-based technique, as well as its relatively moderate memory consumption.

The MVGiST may be of interest to a great variety of applications that are based on multidimensional index structures, especially those that have a high read/write ratio and those that would benefit from consistency across multiple queries. An example use-case, where we recently applied the MVGiST, is a directory indexing web service advertisements in a way that enables efficient, automated service composition. In our previous work [14–16] we have introduced the idea of the MVGiST as a possible concurrency control technique for large-scale service directories. We have continued that work, and, in this article, we give a complete and comprehensive presentation of the MVGiST, we propose a concrete mechanism to control memory consumption, we show how the MVGiST can be integrated into a real system, and we evaluate the performance of our implementation, covering concurrency issues at the implementation level.

The remainder of this article is structured as follows: Section 2 summarizes the features of the GiST. Section 3 presents the MVGiST, introducing the design principles and also discussing implementation issues. In Section 4 we consider a directory of web service advertisements as a use-case, where we successfully applied the MVGiST. In Section 5 we evaluate the performance and scalability of the MVGiST, comparing it with the locking-based concurrency control scheme for the GiST presented in [7]. Furthermore, we show that the MVGiST efficiently manages memory. Finally, Section 6 discusses related work and Section 7 concludes this article.

2. GENERALIZED SEARCH TREE (GiST)

In the following, we give an overview of the GiST in the form of a summary of the description in [8]. For our mechanism, the MVGiST, we make use of a simplified implementation of the GiST that omits two of its key methods, which are not relevant to our scope, as stated later in Section 2.4.

2.1. Search trees

A search tree is a balanced tree. The internal nodes are used as index and the leaf nodes contain the actual data. Every internal node has a series of keys and references to its child nodes. A query on the tree must supply a predicate q. Starting from the root node, a query checks for consistency of q with the keys associated with the child nodes and moves to a child node if its key is consistent with q. It traverses the tree in this manner until it reaches the leaf nodes containing the data that match the query. In classical trees, predicates are constrained to specific types, such as range predicates, where keys delineate a range $[c_{\min}, c_{\max}]$, and a predicate is of the form $c_{\min} \leq i \leq c_{\max}$ (B+ trees [17]). But essentially a search key may be an arbitrary predicate that holds for each datum below the key.

A search tree is, therefore, a hierarchy of categorizations, in which each categorization holds for the data stored under it in the hierarchy. By exposing the key methods and the tree re-balancing methods to the user, arbitrary search trees may be constructed, which is exactly what is accomplished with the GiST.

As the name implies, the GiST is a search tree that is independent of the data types it indexes, it provides basic search tree logic and supports queries that are natural to the target data. In a single piece of code, it unifies the common functionality of search trees; the user of the GiST only needs to provide the necessary extensions for the type of tree that is desired.

For a given type of search tree, the complexity of an operation in a GiST-based implementation corresponds to the complexity of an equivalent operation in an implementation that does not use the GiST. For example, in a GiST-based B+ tree [17] of order b, the complexity of a search operation is $O(\log_b N)$, where N is the number of records in the tree.

2.2. GiST structure

A GiST is a search tree of variable fanout between kM and M, where M is the maximum number of child nodes and $\frac{2}{M} \leq k \leq \frac{1}{2}$, except for the root node, which may have fanout between 2 and M. Inner nodes contain (p, ptr) pairs, where p is a predicate that functions as a search key and ptr references another node. Leaf nodes contain the same pairs, but here ptr identifies some tuple of user data. Predicates can contain any number of free variables, with the restriction that each user data tuple referenced by the leaves of the tree can instantiate all the variables. Figure 1 visualizes the GiST structure.

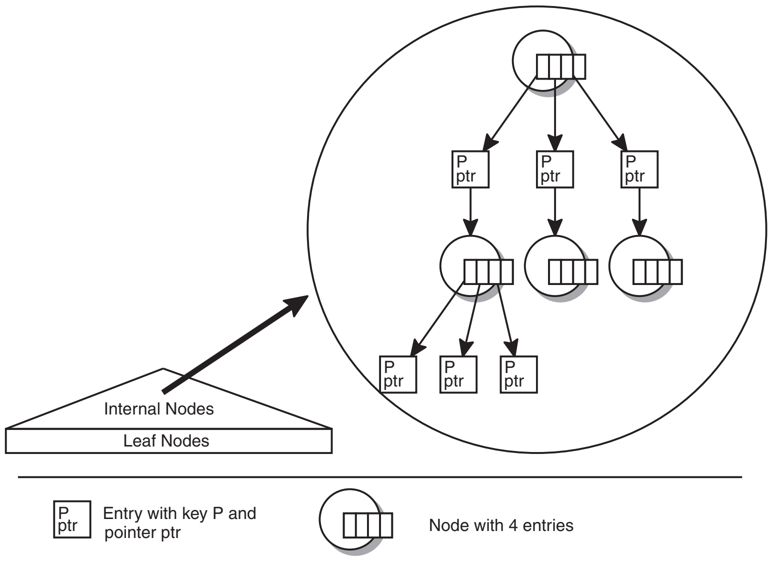


Figure 1. The GiST structure.

2.3. Properties

The following are the invariant properties of the GiST:

- Every node contains between kM and M index entries unless it is the root.
- For each index entry (p, ptr) in a leaf node, p is true when instantiated with the values from the data tuple referenced by ptr.
- For each index entry (p, ptr) in an inner node, p is true when instantiated with the values of any tuple reachable from ptr.
- The root has at least two children unless it is a leaf.
- All leaves appear on the same level (balanced tree).

Note that unlike, e.g. R-trees [18], for any entry (p', ptr') reachable from ptr, p' can express a predicate that is entirely independent of p, where (p, ptr) is an entry in the tree.

2.4. Key methods

The following methods need to be implemented by the user of the GiST:

- **Consistent** (E, q): Given an entry $E = (p, ptr)$ and a query predicate q, returns false if $p \wedge q$ can be *guaranteed* unsatisfiable, true otherwise.
- **Union** (P): Given a set of entries $(p_1, ptr_1), \ldots, (p_n, ptr_n)$, returns some predicate that holds true for all tuples reachable from ptr_1 through ptr_n.
- **Compress** (E): Given an entry $E = (p, ptr)$ returns an entry (π, ptr), where π is a compressed representation of p.

- **Decompress** (E): Given a compressed representation $E=(\pi, ptr)$ such that $\pi=\texttt{compress}(p)$, returns an entry (r, ptr) such that $p \rightarrow r$.
- **Penalty** (E_1, E_2): Given two entries $E_1=(p_1, ptr_1)$ and $E_2=(p_2, ptr_2)$ returns a penalty for inserting E_2 into the subtree rooted at E_1.
- **PickSplit** (P): Given a set P of $M+1$ entries (a node containing these entries is therefore overfull) splits P into two sets of entries P_1 and P_2.

In our implementation we omit `compress`(E) and `decompress`(E). This is because for our purpose, speed is more important than memory consumption. In addition, under the condition that the representation of a compressed predicate is itself a functioning predicate, compress can actually be integrated into union (for inner nodes) and the instantiation of the predicate (for leaf nodes). In this case, decompress corresponds to the identity function and can be ignored.

3. MULTIVERSION-GiST (MVGiST)

Proposed solutions to concurrency control in multidimensional index structures [7,9–11] synchronize individual operations on the tree. There are, however, application domains where long-lasting read sessions are required; by a ‘read session’ we refer to multiple read operations that need to be done on the same set of data. An example of such a use-case is given in Section 4. Generally, applications that rely on such long read sessions do some amount of processing after retrieving potentially disparate data elements, which however need to be coherent as a whole. Our proposition supports long-lasting read sessions; the structure and content of the whole tree remain unchanged to a reader for the duration of its session.

In the following we refer to the version of the tree that is open for modifications as the write tree, respectively, to any version of the tree that is only used for reading as a read tree. A read node is a node from a read tree, and a write node refers to a node in a write tree. The term ‘read session’ defines a series of read operations issued from one client on an unchanging version of the tree.

3.1. Multiversion concurrency control (MVCC)

When processes or threads read and write simultaneously from shared data, the results attained can be very different from what is expected. In the case of a search tree, concurrent reader and writer access could, e.g. lead to retrieval of partial data if a reader executes a query whose predicate is consistent with the predicate of a deleting writer. The goal of concurrency control is to produce a concurrent execution of reads and writes that has the same effect as a serial execution. An execution of this type is called *serializable*. Non-serializable executions may arise when a data item is concurrently accessed by readers and *at least* one writer. It is up to the concurrency control system to order conflicting operations so as to attain the desired execution. In practice, many concurrent access algorithms display some non-serializable executions. This is mostly done in the name of efficiency, and such executions need to be considered and taken into account on a level above the index structure itself.

MVCC is a database technique that adds versioning to shared data. In multiversion databases, every write on a data item x creates a new version of x. As writes do not overwrite each other, this gives greater flexibility to the database system in its ordering of conflicting operations. MVCC has existed for many years and there are several algorithms that exploit multiversions, which all work on the same basis. However, to our knowledge there has been no prior implementation of MVCC for the GiST. Bernstein and Goodman [4] give an in-depth view of MVCC for database systems.

We have implemented MVCC for the GiST as described below. In our implementation, we periodically create new read tree versions from a master write tree, which is constantly under modification. Once a read tree has been created, it is left untouched until all readers have left, then the memory used by the tree can be reclaimed. An arriving reader is always handed the root of the newest read tree, from which it can start multiple queries into a completely fixed search tree. It is clear from this that within one read tree replication interval, any execution containing writes and subsequent reads on the same data is not serializable, because the write only becomes visible to readers after replication of the write tree. However, this is to be expected with multiversioning and is in accordance with the assumptions that our design follows, as is shown in the following section.

3.2. Assumptions

The following assumptions underly the design of the MVGiST:

1. Read accesses are much more frequent than updates.
2. High concurrency for read accesses (high number of concurrent read sessions).
3. Read sessions must offer a consistent view of the tree data; they have to be isolated from concurrent updates to data and index structure.
4. Read accesses shall not be delayed (i.e. not involve any operations that potentially block).
5. Updates may become visible with delay, but feedback concerning the update (success/failure) shall be returned immediately.
6. The duration of a read session can be limited (timeout).

3.3. MVGiST structure

As indicated, the assumptions for the MVGiST with respect to concurrent access lead us to a design in which the readers access a tree that is separate from the write tree. A periodic full replication of the write tree would, however, be far too costly both for memory and performance considerations. We have overcome this obstacle by sharing read nodes between read trees. In this manner, at read tree creation, only the nodes that have been modified since the last tree replication are actually copied to the new read tree. Every write tree node contains a reference to its corresponding read node twin, which is *null* at creation of the write node and is nullified whenever the node is updated. The read nodes themselves are reduced copies of the write nodes, they only contain the fields and methods necessary for read access. Write tree access is sequential, read tree access is concurrent.

Figure 2 visualizes the process of read tree creation, and below there is a generic outline of the algorithm for read tree management, according to the visualized steps. For an easier understanding of the relations between the write and read nodes, as well as those between the read trees, it is assumed that the read trees are created after each modification of the write tree. In reality, a new read tree is only created after a defined number of updates or after a defined time span, and usually not after every update of a write node.

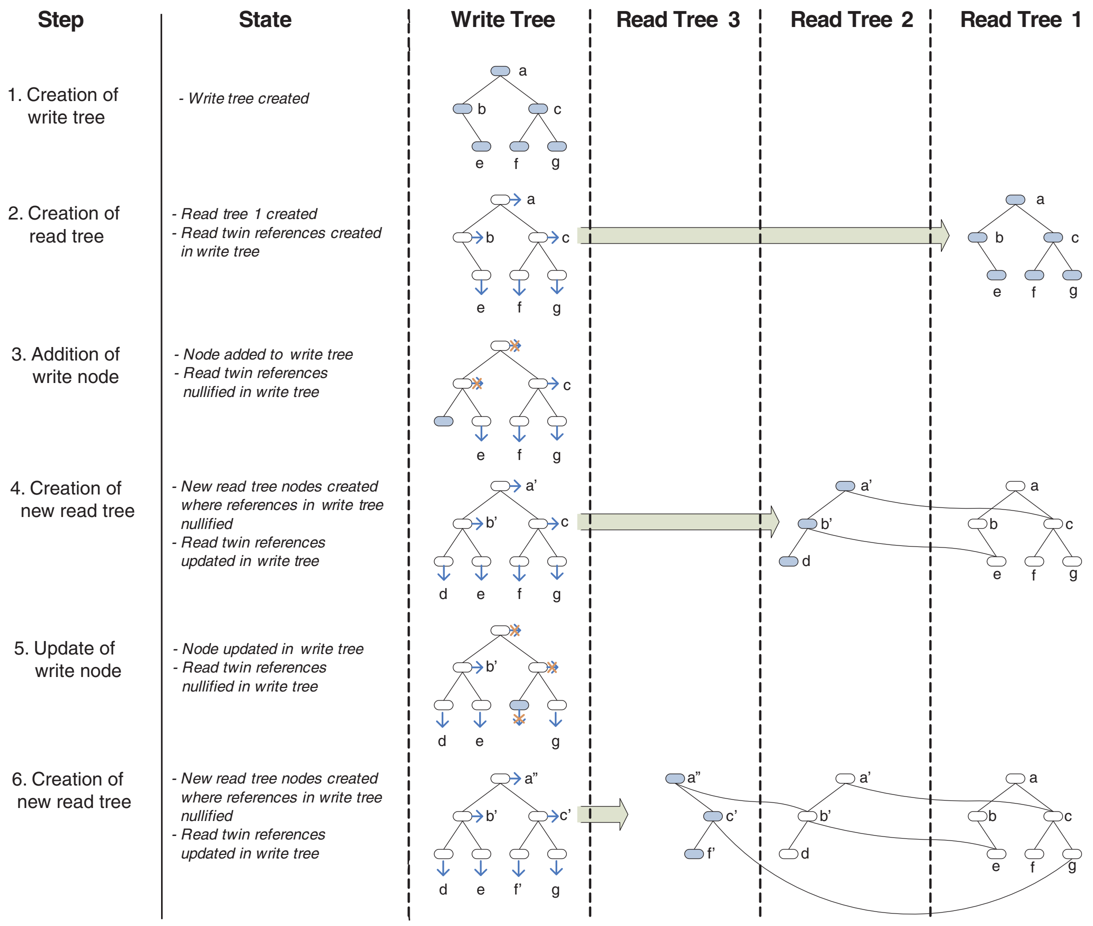

Step
State
Write Tree
Read Tree 3
Read Tree 2
Read Tree 1
1. Creation of write tree
- Write tree created
2. Creation of read tree
- Read tree 1 created
- Read twin references created in write tree
3. Addition of write node
- Node added to write tree
- Read twin references nullified in write tree
4. Creation of new read tree
- New read tree nodes created where references in write tree nullified
- Read twin references updated in write tree
5. Update of write node
- Node updated in write tree
- Read twin references nullified in write tree
6. Creation of new read tree
- New read tree nodes created where references in write tree nullified
- Read twin references updated in write tree
a
b
c
d
e
f
g
a'
b'
c'
f'
a''

Figure 2. The MVGiST read tree creation process.

1. *Creation of write tree*. After instantiation of the MVGiST, the write tree is populated with existing data. All write nodes have their read twin references *null*.
2. *Creation of read tree*. The `createReadTree` procedure is called. This procedure recurses down the write tree and instantiates[‡] a read node for every write node whose read twin reference is *null*. Because in this step `createReadTree` is first called, the procedure creates a read node for every write node. The parent of this new read node (which is also a read node) has an array of child nodes where the procedure places an additional reference to the new read node; thus, creating a complete read tree. The variable holding the reference to the current read tree root is now assigned the new read tree root, which in this case is node *a* from *Read*

[‡]Instantiation of read nodes involves deep copying of predicates.

Tree 1. All write nodes now have references to their corresponding read twins. The read tree remains constant for read access, while the write tree continues to be modified. From this moment on read sessions may be created.

3. *Addition of write node*. A new node is added to the write tree. All modified nodes in the write tree have their read twin references nullified. All references to read twins on the paths from the root to the modified nodes are nullified as well. The added node has no read twin; hence, its reference to the read twin is *null*. The path from the root to the added node contains node a and node b; therefore, these nodes have their read twin references nullified.
4. *Creation of new read tree*. The `createReadTree` procedure is called again, and only the nodes whose read twin references are *null* are duplicated. These new read nodes contain in their child node array references to other new read nodes as well as references to existing read nodes from the first read tree for the regions of the tree that have not been modified since the last `createReadTree` call. As no other modification except the addition of a write node has occurred in the write tree, only the new added write node (d) and the modified write nodes (the nodes on the path from the root to the inserted node, which in our case are node a and node b) are duplicated. Nodes a' and b' in the *Read Tree 2* contain references to new created nodes (a' to b', respectively, b' to d) as well as references to nodes that have not been modified from *Read Tree 1* (a' to c, respectively, b' to e). The new read tree root now becomes the current read tree root reference, and incoming readers receive the new read tree root to access the read tree, which in this case is node a' from *Read Tree 2*.
5. *Update of write node*. Node f is updated in the write tree. According to the algorithm, the reference of this node to its correspondent in the read tree is nullified. References of write nodes a and c to their read twins are also nullified.
6. *Creation of new read tree*. The `createReadTree` procedure is called again and the modified write nodes (a, c, and f) are duplicated. In *Read Tree 3*, node a'' contains references to the new node c' and to node b' from *Read Tree 2*; node c' contains references to the new created node f' and to node g from *Read Tree 1*.

   As soon as all readers have left a specific version of the read tree (either through timeout or on completion), the read nodes that are not referenced by other read tree versions will now gradually be removed by means of garbage collection. Supposing *Read Tree 1* is no longer needed (no reader uses it), read nodes a and b will become eligible for garbage collection, while read nodes c, e, f, and g will be retained as they are referenced by active versions of read trees (*Read Tree 2* and *Read Tree 3*).

Note that upward navigation within the read trees is impossible, because this could lead a reader to outdated versions of the tree. In addition to the properties stated for the GiST, the MVGiST has the following invariant property: within a read session, a reader's view of the tree never changes.

3.4. Supported operations and synchronization issues

There are three general operations that need to be distinguished for the MVGiST:

- *Read session*: One or more read queries, possible timeout.
- *Write operation*: Batch of inserts and deletes to be completed on the write tree.
- *Read tree creation*: Creates a new read tree, does not split write batches.

It is important to note that for the read trees to mirror consistent states across the data tuples, certain sets of writes to the write tree may have to be applied in sequences that are not interrupted by any read tree creation. This is the case if there is some form of semantic dependency between the tuples to be written. A minimum of such dependency could be update atomicity, i.e. an update consisting of a delete and insert operation must not be separated by the read tree creation process. A read tree creation that splits such a set of updates would produce inconsistent read trees w.r.t. the data tuple semantics. We therefore define write operations as batches of insertions and deletions, and a read tree creation must be constrained only to begin after an entire batch has committed.

As all nodes in read trees are immutable, they can be concurrently accessed by any number of threads without synchronization, i.e. concurrent query processing does not involve any mutual exclusion. Therefore, threads executing queries in different read sessions can proceed in parallel, if multiple CPUs are available. The absence of synchronization enables an ideal speedup with an increasing number of CPUs (if there are at least as many read sessions active as there are CPUs).

Starting a new read session requires reading a variable that holds a reference to the root node of the most recent read tree. That variable is mutable, as upon read tree creation, a new root node may be assigned, i.e. we need some form of synchronization to ensure that updates to that variable become visible to all threads accessing that variable. As our MVGiST is implemented in pure Java, we have to follow the rules of the Java memory model [19,20] to ensure visibility of updates to the variable. Concretely, we may use synchronization, a `volatile` declaration, or an atomic reference (an instance of `java.util.concurrent.atomic.AtomicReference`). As the variable is not involved in invariants with any other mutable state, we chose the 'lightest' of these options, the `volatile` declaration. Note that only the creation of a read session requires a single access to the variable holding the most recent read tree root, whereas queries issued within the read session do not access that variable anymore, i.e. the performance impact of the `volatile` declaration is negligible, because the variable is not frequently accessed.

We now have a data repository in which readers can concurrently access the data without any danger of incoherence due to writers simultaneously accessing the tree. However, the write tree is still exposed to the same concurrent access problems as the GiST. As already mentioned, previous research work has addressed the issue of concurrency control in the GiST [7,9–11]. As tree modification is anticipated to be far less frequent than read access, our current implementation simply serializes the updates on the write tree and read tree creations. A concurrent access algorithm for the write tree would have to take into account the periodic read tree creation, which must output a coherent new read tree according to the constraints of the application domain.

3.5. Implementation issues

As we planned to integrate the MVGiST into a Java-based system (see Section 4), we implemented the MVGiST in Java. The following are the principal implementation differences to a non-concurrent GiST:

- The MVGiST comprises two types of trees: the write tree, which is similar to the GiST, as well as read trees, which do not contain the user-implementable methods summarized in Section 2 and other methods related to updating the tree.
- The MVGiST class contains additional methods for read tree creation and for accessing the root of the most recent read tree.

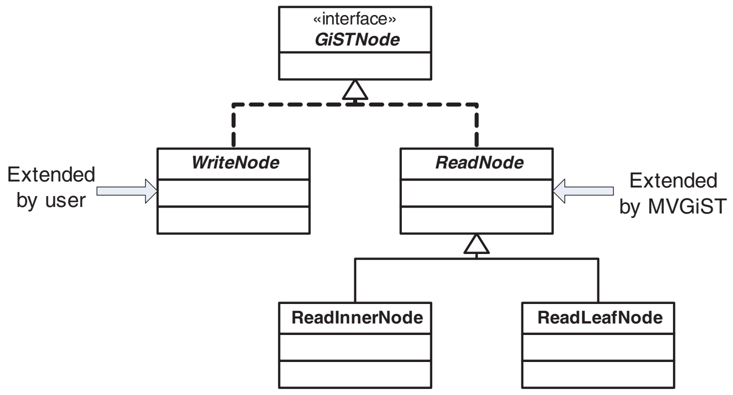


Figure 3. The MVGiST node inheritance model.

- In view of the fact that our MVGiST may well be used for very large data sets, some effort was taken to reduce the size of nodes in the read tree. Read nodes are reduced copies of their corresponding write nodes, containing only the fields relevant for tree traversal, e.g. the inner nodes do not contain an empty data object and leaf nodes do not contain an empty array of child nodes.
- Because upwards navigation in read trees is impossible, there is no parent node reference in a read node.

The existing commonalities between all nodes for the different trees naturally lead to a class structure based on the inheritance of common functionalities. Figure 3 displays the class inheritance model for MVGiST nodes.

The `GiSTNode` interface defines aspects of all nodes. Two abstract classes implement this interface: `WriteNode` (nodes belonging to the write tree) and `ReadNode` (nodes belonging to a read tree). The `ReadNode` class is extended by `ReadInnerNode` and `ReadLeafNode` for the above-mentioned reason. The `WriteNode` class was not extended to two subclasses, although this also would have allowed for less memory usage. In fact, if this was the case, the MVGiST user would be obliged to implement the key methods doubly in both subclasses, which is poor OO-design (due to the single-inheritance model of Java, we cannot outsource these methods to another class).

3.6. Read tree creation strategies

Below we consider the freshness and memory consumption of MVGiST read trees. From these considerations, we derive two simple read tree creation strategies.

An important issue is the freshness of the most recent read tree (differences between write tree and read tree). The parameters involved are the frequency of tree modification by writers and the frequency of read tree creation. The freshness of a read tree R is indirectly proportional to the number of changes in the write tree since the creation of R.

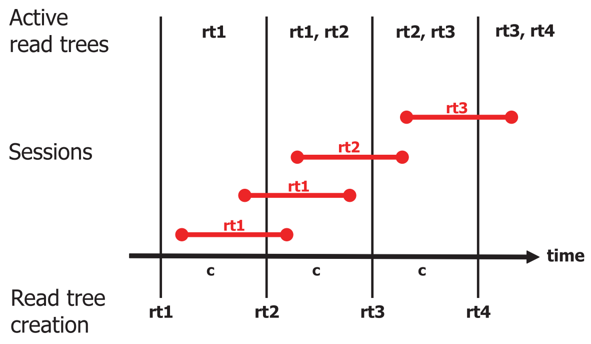


Figure 4. At most two read trees are active at the same time when the read session timeout t does not exceed the read tree creation interval c.

Memory consumption is proportional to the number of read trees in existence plus the write tree itself. The maximum duration of a read session t and the time between subsequent read tree creations c control the number of read trees that can be active at the same time. The number of active read trees is $1+\lceil \frac{t}{c} \rceil$. Active read trees are the current read tree (whose root node is referenced by the volatile variable holding a reference to the current read tree root node, as well as possibly by active read sessions) and older read trees referenced by active read sessions. In order to minimize memory consumption, c and t may be chosen such that $t \leq c$, in which case only two read trees have to be kept in memory. Ensuring a limit t on the duration of read sessions requires a timeout mechanism for read sessions. As query processing in read sessions is application-specific, such a timeout mechanism cannot be part of the MVGiST itself, but has to be provided by the system using the MVGiST. In Section 4.3 we give an example of a timeout mechanism.

Figure 4 displays the number of active read trees in memory when $t = c$. This case illustrates the situation where long-lasting read sessions are terminated by a timeout. As shown, each read session operates on the newest active read tree upon session start. At any moment, only two read tree versions are active (i.e. accessed by read sessions). In Section 5.2 we will analyze the memory consumption of the MVGiST in detail.

The MVGiST provides two read tree creation strategies, freshness-triggered and timing-triggered (the user of the MVGiST may also implement different strategies). For both strategies, there is a dedicated thread T_{update} that is responsible for processing updates and for periodically creating a new version of the read tree. T_{update} gets batches of update requests from a synchronized queue Q.

In the case of freshness-triggered read tree creation, T_{update} keeps track of the number of updates since the last read tree creation. If this number exceeds a given threshold, T_{update} creates a new read tree before obtaining the next batch of update requests from Q. In this read tree creation strategy, T_{update} blocks until a new batch of updates is available.

In the case of timing-triggered read tree creation, T_{update} guarantees a given minimum time span c between consecutive read tree creations. The actual time span between read tree creations is $c+\varepsilon$ ($\varepsilon \geq 0$), and T_{update} aims at minimizing ε (e.g. if there is a batch of updates in progress, $\varepsilon > 0$). T_{update} checks whether a new read tree is to be created before and after processing each batch of updates. T_{update} blocks until a new batch of updates is available or until the time span c since the last read tree creation has passed (T_{update} blocks on Q with a timeout).

4. USE-CASE: WEB SERVICE DIRECTORY SUPPORTING EFFICIENT SERVICE COMPOSITION

Directories of web service advertisements are an example of data repositories, which can profit from the MVGiST concurrency control mechanism. In this section we give an overview of our web service directory, which is based on the MVGiST and supports special features to enable efficient, automated service composition.

4.1. Context: web services, service discovery, and service composition

Service-oriented computing enables the construction of distributed applications by integrating services that are available over the web [21]. The building blocks of such applications are *web services* that are accessed using standard protocols. A web service description is a formal specification of the functionality of a web service. Through this specification, a software agent can either request or advertise a specific functionality. Web service descriptions based on WSDL [22] define the input/output behavior and grounding of web services, whereas more expressive formalisms, such as OWL-S [23], also allow to specify the service semantics (semantic web services), e.g. in the case of OWL-S, preconditions and effects can be used to model service semantics.

Service discovery is the process of locating providers advertising web services that can satisfy a given service request. *Service composition* addresses the problem of assembling individual web services based on their functional specifications in order to create a value-added, composite service that fulfills a service request. Most approaches to automated service composition are based on AI planning techniques [24,25]. They assume that all relevant web service advertisements are initially loaded into a reasoning engine.

However, due to the large number of web service advertisements and to the loose coupling between service providers and consumers, web services are indexed in directories. As loading a large repository of web service advertisements into a reasoning engine is not practical, service composition algorithms have been modified in order to dynamically retrieve relevant web service advertisements from a directory during composition [26,27], i.e. the service composition algorithm is a client querying the directory.

4.2. Web service directory based on the MVGiST

We developed a specialized directory that indexes (semantic) web service advertisements according to their input parameters, output parameters, preconditions, and effects [28,29]. For the following reasons, the MVGiST is particularly well suited as index structure for our directory:

- (Semantic) web service descriptions can be considered multidimensional data (input and output parameters, preconditions, effects).
- Read access is far more frequent than updates.
- The process of automated service composition requires a long-term, constant view of the directory data; therefore, calling for consistency over multiple reads.
- Because queries generated by service composition algorithms can be complex and a high number of concurrent accesses is to be expected (the directory is a shared resource), overhead for read concurrency has to be as small as possible.

Thanks to the MVGiST, our directory enables efficient, automated service composition in an open environment populated by a large number of advertised web services. It supports the incremental retrieval of query results that are obtained by a best-first search over an MVGiST read tree, which greatly improves the efficiency of service composition [28,29], as the composition algorithm does not have to wait for the directory to compute the complete result set. The service composition algorithm can already work on partial results and retrieves additional results on demand. This approach also reduces the workload in the directory, as it may not have to compute the complete result set (if the composition algorithm succeeds with the retrieved partial results, it may close the result set; hence, avoiding the computation of the complete result set).

Incremental retrieval of the results of a directory query and interleaving service composition with directory accesses requires the directory to support 'long reads', i.e. the result set may be kept open for a longer time, while the client incrementally retrieves the results. The correctness of the incremental query processing described in [28] relies on a constant view of the index structure during the whole query. The MVGiST is perfectly suited to support these features.

4.3. Controlling memory consumption with a timeout mechanism

Our directory relies on the timing-triggered read tree creation strategy introduced in Section 3.6. We chose the timeout for read sessions t equal to the read tree creation interval c in order to minimize memory consumption. In this use-case, typical values for $t = c$ are 1–60 min.

As mentioned before, limiting the duration of read sessions is essential to control the number of read trees that are active at any moment. The implementation of a timeout mechanism for read sessions is not part of the MVGiST, but has to be provided by the system that uses the MVGiST; in our case, this is the web service directory. In the following we describe the timeout mechanism used in the web service directory.

The thread T_{update} examines all active read sessions before creating a new read tree. If it detects a read session that has been active for too long (i.e. the session is using an outdated read tree version that should become eligible for garbage collection before creating the next read tree version), T_{update} has to invalidate that session before creating a new read tree.

Invalidation means that the session must not keep any read tree nodes alive. If no thread is processing any query in that session, session invalidation is trivial; the state of the session (which may include references to read tree nodes) is nullified and the session is marked as expired. Any subsequent attempt to execute a query in the expired session will raise an exception.

However, if a thread T_{query} is processing a query in the session, T_{update} first interrupts T_{query} in order to cancel the query processing. This is necessary because T_{query} may reference read tree nodes in local variables or on the stack. Our query processing logic (which is application-specific and not hard-coded in the MVGiST) is designed to support interruption; i.e. we are using a collaborative mechanism for canceling read sessions that are lasting too long[§].

This approach implies that T_{update} may have to wait for some (short and bounded) amount of time until T_{query} responds to the interruption. Consequently, the effective read tree creation interval

[§]Note that Java does not provide any reliable mechanism for asynchronous task termination. Hence, it is necessary to design a collaborative task cancelation protocol [19,20].

may exceed c. But anyway, c is to be considered only as an approximate value, because an ongoing update may delay read tree creation and because thread scheduling cannot be controlled in Java[¶].

In order to guarantee that T_{query} reacts to an interruption within a bounded time, we have to ensure that T_{query} does not perform any non-interruptible blocking operations (actually, T_{query} does not perform any blocking operations, as read tree access does not require any synchronization) and that the number of instructions executed between subsequent checks of the interruption status is bounded. This constraint is enforced by polling the interruption state in each loop, upon method entry and upon method exit, i.e. we are using a form of call/return polling as described by Feeley [30]. Such a collaborative task cancelation mechanism causes some overhead because of the introduced polling code. Moreover, accessing the interrupted state of the current thread may require memory barrier instructions in order to ensure visibility of a state change, also contributing to the polling overhead. However, note that any reliable implementation of a task cancelation protocol in Java incurs such overhead [20].

5. EVALUATION

In this section we evaluate the performance and scalability of the MVGiST, as well as its memory consumption. All experiments operate on an initial tree with fanout of 4–8, which stores 10 000 keys. For the purposes of our evaluation, we used an R-tree [18] implementation for the GiST. Read operations search for keys that are known to be stored in the tree (i.e. all read operations are guaranteed to succeed). Updates consist of one delete and one insert operation, in order to keep the tree size constant; the deletion is guaranteed to succeed and the insertion stores a new unique key in the tree.

5.1. Performance and scalability

In the following we analyze the throughput achieved by the MVGiST, measured as the number of read respectively update operations per second, for different workloads and different levels of concurrency (1–100 concurrent threads). Each thread represents a client accessing the MVGiST; hence, in the following we call these threads 'client threads'. A workload is a mix of read and update operations (0–100% updates).

In order to assess the strengths and drawbacks of the MVGiST, we compare the MVGiST with a traditional, locking-based concurrency control scheme for the GiST as presented by Kornacker *et al.* [7]. To the best of our knowledge, there is no implementation of this concurrency control mechanism in Java, hence, we developed our own Java reference implementation, henceforth named Kornacker's concurrent GiST (KCGiST). The scheme presented by Kornacker *et al.* represents a concurrent access system complete with logging and recovery facilities. To avoid any unfairness in the comparison, we restricted ourselves exclusively to the concurrency and consistency aspects of this scheme. A description of the KCGiST is given in Section 6.1.

[¶]Java supports thread priorities, but does not specify how these priorities affect thread scheduling; i.e. scheduling of Java threads is platform-dependent.

Please note that the functionality offered by the MVGiST and the KCGiST is not exactly the same. The KCGiST supports transactions and ensures repeatable read isolation [31], whereas the MVGiST provides a constant read-only view of the whole index tree.

5.1.1. Benchmark settings

The benchmark to measure the throughput was set up to create identical workload for the two competing concurrency control schemes. Each client thread executes a randomly generated workload (a mix of read and update operations) with a given percentage of update operations. The workload is represented as a list that is processed sequentially by the client thread.

In the case of the KCGiST, all read and update operations issued by the client threads directly access the same tree. A dedicated thread T_{cleanup} takes care of cleaning up the logically erased entries in the tree.

In the case of the MVGiST, only read operations are performed directly by the client threads, which obtain the most recent version of the read tree upon each read request. In contrast to the KCGiST, updates are not performed by the client threads. Rather, there is a common, synchronized queue Q, where client threads enqueue update requests. The update operations are handled by the dedicated, high-priority thread T_{update}, which is the only thread allowed to access the write tree. Our benchmark uses the freshness-triggered read tree creation strategy explained in Section 3.6. T_{update} creates a new read tree whenever it has processed 1000 update requests.

For each run of the KCGiST respectively MVGiST benchmark, we first populate the tree. For the MVGiST, we also create an initial read tree. Afterwards, we generate the workloads and start the client threads. We use synchronization barriers to ensure that all client threads are ready to process their workload before starting the measurement. Upon completion of all client threads, we wait for the dedicated threads T_{cleanup} respectively T_{update} to process pending tasks, before finishing the measurement.

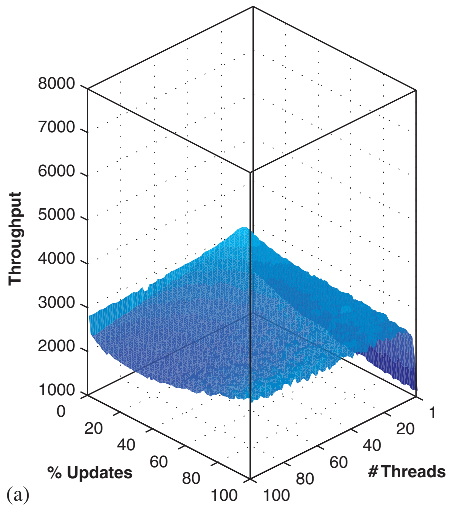
The parameters of our measurements are the percentage of updates in the workload (0–100%) and the level of concurrency (1–100 client threads). For each setting, we execute the KCGiST respectively MVGiST benchmark 15 times and take the median of the measured throughput values. After each run, we force garbage collection.

As platform we used a machine with four CPUs (two dual-core Xeon 3 GHz) and 4 GB of RAM, running a 64-bit Windows XP installation. We employed the Sun JDK 1.5.0 with its 64-bit Hotspot Server Virtual Machine. In this environment, Java threads are mapped to kernel threads scheduled by the operating system; i.e. up to four Java threads can execute in parallel on the available CPUs. We disabled background processes as much as possible in order to ensure consistent system conditions.

5.1.2. Measurements

Figure 5 shows three-dimensional surface plots of the measured throughput, depending on the level of concurrency (number of client threads) and the percentage of updates in the workload.

For a lower percentage of updates, the MVGiST achieves about 2.5 times the throughput of the KCGiST. However, the throughput of the MVGiST significantly degrades with an increasing percentage of updates, whereas the throughput of the KCGiST remains rather stable independent of the workload. The KCGiST throughput does not suffer from a high update percentage, because node



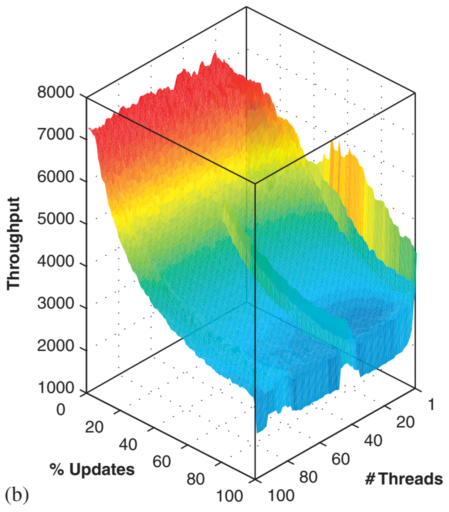


Figure 5. Throughput of KCGiST (a) and MVGIST (b), depending on the level of concurrency (1–100 client threads) and the percentage of updates in the workload (0–100%). Number of leaf nodes: 10 000; fanout: 4–8; median of 15 runs. MVGiST read tree creation after each 1000 updates.

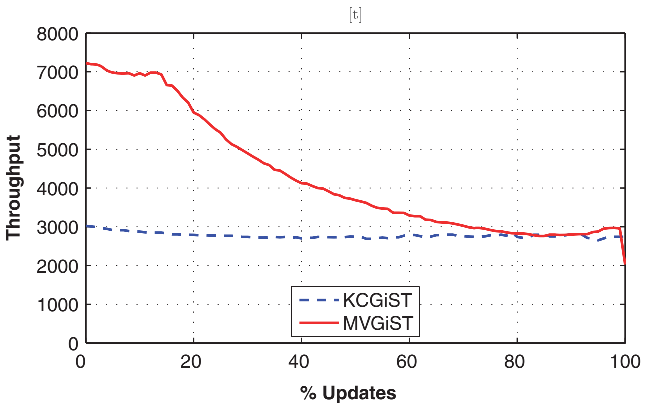


Figure 6. Throughput depending on the percentage of updates in the workload (0–100%) for a concurrency level of 20 client threads. Number of leaf nodes: 10 000; fanout: 4–8; median of 15 runs. MVGiST read tree creation after each 1000 updates.

reorganization is not frequent, as the tree is kept at a constant size. In contrast, for the MVGiST, the overhead due to read tree creation increases with a higher update percentage. In addition, updates are serialized, because only a single thread (T_{update}) can access the write tree. Figure 6 illustrates a slice plane of Figure 5 for a concurrency level of 20 client threads, confirming that for a low percentage of updates, the MVGiST significantly outperforms the KCGiST.

[t]

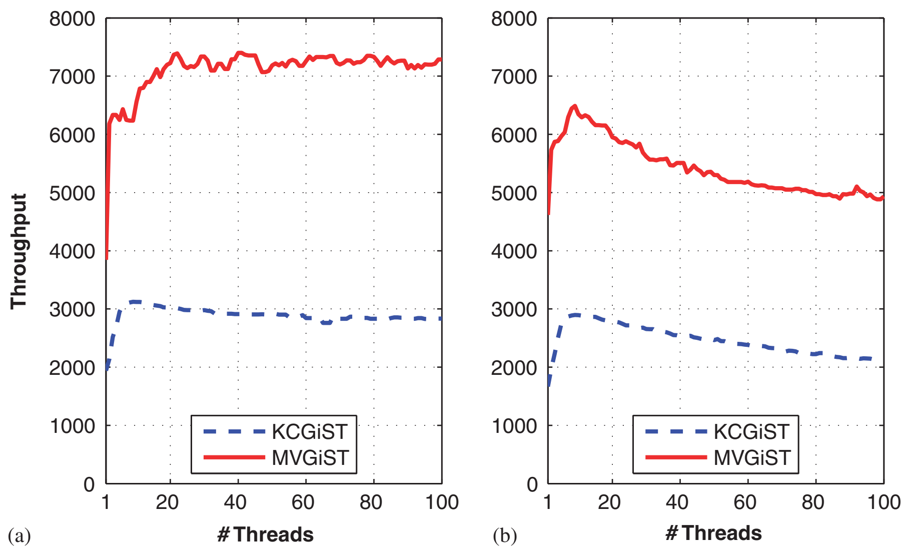


Figure 7. Throughput depending on the level of concurrency (1–100 client threads) for a workload with 0% updates (a) respectively 20% updates (b). Number of leaf nodes: 10 000; fanout: 4–8; median of 15 runs. MVGiST read tree creation after each 1000 updates.

For a lower percentage of updates, both concurrency control schemes scale well with the number of client threads. Figure 7 depicts two slice planes of Figure 5 for a workload with 0% respectively 20% updates. In the case of 0% updates, an increasing number of threads does not deteriorate throughput. However, a higher percentage of updates impacts scalability for both concurrency control mechanisms.

If there are only very few client threads (1–3), the throughput drops significantly, because some CPUs of our multiprocessor machines are idle. Interestingly, in a setting with very few client threads, the throughput of the MVGiST increases with the percentage of updates in the workload, reaching a peak at about 50% updates, before dropping again (see Figure 5). The reason for this behavior is that the MVGiST uses a dedicated thread T_{update} to process update requests. On a multiprocessor, T_{update} may execute (accessing the write tree) in parallel with client threads processing read requests (accessing a read tree).

5.2. Memory consumption

An obvious drawback of the MVGiST is the extra memory consumed by the read trees. As we have seen in Section 3.6, in general at least two read tree versions are active at the same time. While in a naive implementation, each read tree would represent a deep copy of the write tree at a certain moment, our approach allows common subtrees to be shared among different read tree versions; thus, reducing memory consumption. In the following we explore the memory consumption of the MVGiST and analyze the memory savings thanks to the sharing of subtrees.

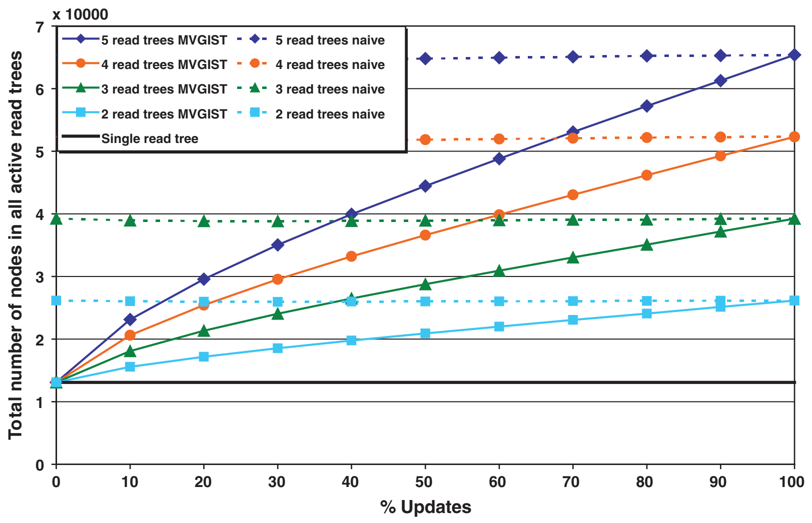


Figure 8. Total number of nodes in all active read trees (2–5). Number of leaf nodes: 10 000; fanout: 4–8.

If we assume the size of the write tree constant upon read tree creation, the memory consumption of the MVGiST depends on two parameters; (1) on the number of active read trees and (2) on the percentage of tree nodes that have been modified between subsequent read tree creations.

There is no inherent limit for the number of active read trees or for the size of a read tree. However, since the MVGiST is intended as an in-memory index, the nodes of the write tree and of all active read trees have to be kept on the heap. As the MVGiST is implemented in Java, the size of the trees and the number of active read trees are limited by the amount of heap space allocated to the JVM process.

For the purpose of measuring memory consumption, we first consider the total number of nodes in all active read trees (instances of the type `ReadNode`) as a platform-independent metric. Second, we also investigate physical memory usage in one particular environment.

The measurements shown in Figure 8 demonstrate a practically linear increase in memory usage in proportion to the percentage of updates in the workload and the number of active read trees. The steeper incline at a lower update percentage is due to the fact that most paths from the root to the affected leaf nodes are not shared among different updates. As the update percentage increases, the gradient straightens out, as more and more inner tree nodes that are being updated are commonly modified by different updates.

The series ‘Single read tree’ in Figure 8 is intended as reference, representing the number of nodes in the initial read tree, which equals the number of nodes in the write tree (about 13 000 nodes in this example). The ‘naive’ series correspond to a tree replication algorithm that does not share any nodes among different read tree versions.

These measurements confirm that for workloads with a low update percentage, the MVGiST efficiently manages memory, even if there are several versions of the read tree active at the same time. In such settings, the number of nodes in all active read trees is not much larger than the number of

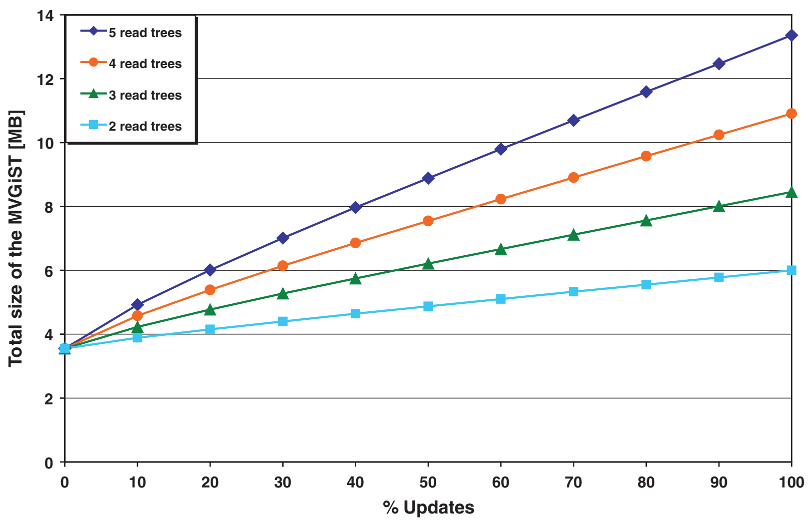


Figure 9. Total size (megabytes) of the complete MVGiST, including the write tree and 2–5 read trees. Number of leaf nodes: 10 000; fanout: 4–8.

nodes in the write tree, i.e. for a low update percentage, the total number of nodes in the MVGiST is only slightly more than twice the number of nodes in a GiST without concurrency control. The memory consumption of the MVGiST can be even below twice the memory consumption of a GiST without concurrency control, because nodes in a read tree are more compact than nodes in the write tree.

An important point is that the MVGiST is far more lightweight than other concurrency control schemes concerning the size of the tree nodes themselves, especially with respect to the read tree nodes. These nodes only contain the absolute minimum, which is necessary for reader navigation. The write nodes have only one field more than in the non-concurrent GiST implementation: the reference to the read node twin. Other concurrency control schemes often maintain multiple lists and locks on a per-node basis in addition to the standard GiST node elements.

In order to measure the physical memory consumed by the MVGiST, we implemented a profiling method that estimates the size of a given object graph. Passing an MVGiST instance to that method yields an estimate of the bytes consumed on the heap by the whole MVGiST (including the write tree and all read trees). Our profiling method relies on the `java.lang.instrument` API that has been introduced in JDK 1.5. The abstraction `Instrumentation` offers the method `getObjectSize(Object)` to estimate the size of a given object. Our profiling method uses reflection to process all reachable, relevant[‖] objects in order to compute the overall size of an object

[‖]Relevant objects include those that are referenced by non-static fields or by array elements. Objects referenced by static fields are not considered.

graph. A set of processed objects is maintained to avoid counting the size of the same** object multiple times.

Figure 9 shows the physical memory consumption (in megabytes) of our MVGiST-based R-tree implementation for different settings (number of active read trees respectively update percentage). For the measurements, we used a Windows XP machine with an Intel Core Duo T2600 CPU and a 1 GB of RAM, running Sun's JDK 1.6.0_03 Hotspot Client Virtual Machine. For a low update percentage, the whole MVGiST consumes less than 4 MB memory. In the worst measured setting (5 active read trees, 100% updates), the memory consumption reaches 14 MB.

We conclude that after an initial burst, memory consumption increases linearly for a higher update percentage. Considering the relative cheapness of memory today, we argue that the extra memory consumption of the MVGiST is not a hurdle in practice. If updates are relatively rare, the improved performance and scalability of the MVGiST outweigh the costs due to increased memory consumption.

6. RELATED WORK

In the following we present the reference scheme we used for our benchmark as well as some other algorithms for concurrency control on multidimensional index structures. Furthermore, we mention other research and applications in the two domains of multiversioning and of multidimensional data structures.

6.1. Concurrency control for the GiST by Kornacker *et al.*

Kornacker *et al.* [7] introduced the initial, locking-based concurrency control mechanism for the GiST, which is also the basis for concurrency control in PostgreSQL [12]. We call this scheme KCGiST. In the following, we give a brief description of the KCGiST, which we also implemented for our performance and scalability evaluation.

The KCGiST achieves basic concurrent protection by adding node-locks and two other components to the tree: node sequence numbers and rightlinks from every node to its split off right twin. This allows every operation traversing the tree to detect node-splits when switching from one node to the next, and ensures concurrent access protection (S-mode for readers, X-mode for writers) when an operation is within a node.

The KCGiST implements repeatable read isolation [31]. This level of transactional isolation implies that if a search operation is run twice within the same transaction, it must return exactly the same result. The KCGiST achieves this by relying on a hybrid mechanism of two-phase locking of data records and predicate locking. Existing data records are protected by a two-phase locking protocol and phantom insertions are avoided with a restricted form of predicate-based locking. These predicate locks are not registered in a tree-global list as in pure predicate-based locking, but are directly attached to the nodes. Again in contrast to pure predicate locking, it is only the search operations that attach their predicates to the nodes whose keys they are consistent with. Delete

**Here, we refer to object identity, not to object equality.

operations are performed only logically. These two factors allow search and delete operations to ignore predicate locks, only insert operations check for search predicates. Furthermore, they only do this when they arrive at their target leaves, preventing phantom insertions in this way. The search predicates attached to the nodes must always remain consistent with the tree structure; hence, they are transferred between nodes during node-split operations and bounding-predicate (the key contained within every node) modifications.

As the delete operation only does a logical delete, these nodes need to be deleted later on at some point. This can be accomplished through operations that happen to pass through the corresponding nodes. However, deletion may leave a node completely empty, upon which it also should be removed from the tree. This can be a problem, because ongoing operations may still have direct or indirect (e.g. rightlinks) references to such nodes, making a deletion impossible. This is why an additional signaling lock is introduced, which processes attach onto a node when they reference it as part of their tree traversal process. These locks are released as soon as the operations visit the corresponding nodes. A physical deletion operation now tries to attain an X-mode lock on these locks for a node that is to be deleted, and only deletes the nodes whose locks it can all attain. In our implementation of the KCGiST, we delegate the physical deletion to a separate thread, which hands off execution when there are no nodes to delete.

Our KCGiST implementation relies on the concurrency classes offered with Java 1.5 [20], such as `ReentrantReadWriteLock`, which replace the X- and S-mode locks described in [7], and concurrent collections such as the `LinkedBlockingQueue`. This queue is filled with the entries that are to be deleted, and as long as it is non-empty it is processed by the physically deleting thread.

6.2. Other concurrent access algorithms for the GiST

Below we summarize prevailing approaches to add concurrency control to the GiST or to multidimensional index structures in general apart from the method already presented as our reference algorithm. These algorithms demonstrate some speedup in comparison with the KCGiST, but they all suffer the same issues of locking-induced overhead.

In [9] a node-locking-based approach to concurrency on multidimensional index structures is optimized as follows: simultaneous node-locking is avoided when updating bounding predicates by directly modifying the indexes while operations traverse down the tree. This scheme is then extended to reduce the blocking overhead during node-splits through local copying of the nodes, processing of the node-split, and then copying back the resulting changes.

An alternative concurrency control mechanism for the GiST is discussed in [10], which uses granular locking instead of predicate locking. In granular locking, the predicate space is divided into a set of lockable resource granules. Transactions acquire locks on granules instead of on predicates. The locking protocol guarantees that if two transactions request conflicting locks on predicates p and q such that $p \wedge q$ is satisfiable, then the two transactions will request conflicting locks on at least one granule in common. In [10] the authors conclude that while granular locking causes a lower lock overhead than predicate locking, it may reduce concurrency.

In [11] the authors describe an enhanced concurrency control algorithm that reduces blocking time during split operations. They avoid lock coupling during bounding-predicate updates with a partial lock coupling technique, and have developed an update method that allows readers to access nodes where updates are being performed. In particular, such enhancements improve performance if updates are frequent (e.g. in a moving objects database).

In contrast to the aforementioned concurrency control algorithms for the GiST, the MVGiST is optimized for settings where read access is much more frequent than updates and where readers require a consistent view of the tree for a longer period of time. By introducing multiple read-only versions of the tree, we gain non-blocking read access and isolation at the expense of increased memory consumption, although our tree replication algorithm minimizes the allocation of nodes and allows sharing of unchanged subtrees between different read trees.

To the best of our knowledge, the MVGiST is the first MVCC scheme for the GiST.

6.3. General multiversioning

Independent of the GiST or other multidimensional index structures, there has been a large amount of research on MVCC. A look at these various approaches would go beyond the scope of this text, but the interested reader may find a selection of articles related to multiversioning in [4,6,32–40].

One may note that versioning is also applied in applications where the aim is not concurrency control: the common need of applications that use versioning is to have consistent, unchanging views of the data. Examples may be file storage systems such as logical volume management [41] where a ‘snapshot’ mechanism allows the user to dynamically create a consistent view of his data, or version control systems [42,43] that create data versions, so as to make available not only the current data but the entire data history, and also to enable mechanisms specific to these systems such as branching and merging.

6.4. Multidimensional index structures and their applications

As far as search tree logic is concerned, the GiST subsumes most multidimensional index structures. We refrain from indicating a series of references to different multidimensional index structures but instead point to two excellent overviews of such access methods given in [44,45]. Various database management systems implement multidimensional access, although they generally tend to prefer stable and tightly integrated, but simple algorithms rather than some of the more complex versions presented in the aforementioned overviews.

A category of applications that may be worth mentioning are OLAP tools. These are systems that process analytical queries that are dimensional in nature, and their repositories employ a multidimensional data model. To date there are a host of commercial OLAP tools on the market, primarily aimed at the area of business reporting for sales, marketing, and management/financial reporting. Another domain of multidimensional index structures are spatially enabled databases. E.g., the PostGIS [13] spatial database for geographic information systems relies on the GiST implementation of PostgreSQL.

7. CONCLUSION

In this article we have introduced the MVGiST, which enhances the GiST with MVCC, and evaluated its properties both on a performance and on a memory consumption level.

Like the GiST, the MVGiST provides a customizable index for multidimensional data and can be extended to nearly all types of tree-based access methods. In addition, it provides immutable snapshots of the index that enable lock-free concurrent read access. The fact that the MVGiST supports consistency of data across queries makes it attractive for applications where some form of consistency beyond transactional isolation is required. This may typically be the case for systems where a large amount of post-query processing is done on multiple query retrievals.

The common drawback of versioning algorithms is an increased memory consumption. By making use of timeouts and node sharing between read trees, it is possible for the MVGiST to control memory consumption and keep this increase moderate, as the number of read trees and the update percentage between subsequent read tree creations can be maintained low.

For application domains with a high read/write ratio, the MVGiST outperforms a locking-based reference scheme by a factor of about 2.5. Owing to the non-blocking nature of its read access, it scales very well with an increasing number of concurrent read accesses.

REFERENCES

1. Chaudhuri S, Dayal U. An overview of data warehousing and OLAP technology. *SIGMOD Record* 1997; **26**(1):65–74.
2. LDAP. The Lightweight Directory Access Protocol. http://www.kingsmountain.com/directory/doc/ldap/ldap.html [16 September 2008].
3. Java Community Process. JSR 170—Content Repository for Java. http://www.jcp.org/en/jsr/detail?id=170.
4. Bernstein PA, Goodman N. Multiversion concurrency control—Theory and algorithms. *TODS* 1983; **8**(4):465–483.
5. Lomet D, Salzberg B. Concurrency and recovery for index trees. *The VLDB Journal* 1997; **6**(3):224–240.
6. Becker B, Gschwind S, Ohler T, Seeger B, Widmayer P. An asymptotically optimal multiversion b-tree. *The VLDB Journal* 1996; **5**(4):264–275.
7. Kornacker M, Mohan C, Hellerstein JM. Concurrency and recovery in generalized search trees. *Proceedings*, *ACM SIGMOD International Conference on Management of Data*: *SIGMOD 1997*, Tucson, Arizona, U.S.A., Peckman JM (ed.), 13–15 May 1997.
8. Hellerstein JM, Naughton JF, Pfeffer A. Generalized search trees for database systems. *Proceedings of the 21st International Conference on Very Large Data Bases*, *VLDB*, Dayal U, Gray PMD, Nishio S (eds.). Morgan Kaufmann: Los Altos, CA, 1995; 562–573.
9. Kanth K, Serena D, Singh A. Improved concurrency control techniques for multi-dimensional index structures. *IPPS '98*: *Proceedings of the 12th International Parallel Processing Symposium*. IEEE Computer Society Press: Washington, DC, U.S.A., 1998; 580–586.
10. Chakrabarti K, Mehrotra S. Efficient concurrency control in multidimensional access methods. *SIGMOD '99*: *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*. ACM Press: New York, NY, U.S.A., 1999; 25–36.
11. Song SI, Kim YH, Yoo JS. An enhanced concurrency control scheme for multidimensional index structures. *IEEE Transactions on Knowledge and Data Engineering* 2004; **16**(1):97–111.
12. PostgreSQL. http://www.postgresql.org/ [16 September 2008].
13. PostGIS. http://postgis.refractions.net/ [16 September 2008].
14. Binder W, Constantinescu I, Faltings B, Spycher S. Multiversion concurrency control for large-scale service directories. *Third European Conference on Web Services* (*ECOWS-2005*). IEEE Computer Society Press: Växjö, Sweden, November 2005; 50–61.
15. Binder W, Spycher S, Constantinescu I, Faltings B. An evaluation of multiversion concurrency control for web service directories. *2007 IEEE International Conference on Web Services* (*ICWS-2007*), Salt Lake City, U.S.A., July 2007; 35–42.
16. Binder W, Spycher S, Constantinescu I, Faltings B. Multiversion concurrency control for multidimensional index structures. *18th International Conference on Database and Expert Systems Applications* (*DEXA-2007*) (*Lecture Notes in Computer Science*, vol. 4653). Springer: Regensburg, Germany, September 2007; 172–181.
17. Comer D. Ubiquitous b-tree. *ACM Computing Surveys* 1979; **11**(2):121–137.
18. Guttman A. R-trees: A dynamic index structure for spatial searching. *SIGMOD '84*: *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*. ACM Press: New York, NY, U.S.A., 1984; 47–57.

19. Gosling J, Joy B, Steele GL, Bracha G. *The Java Language Specification* (*The Java Series*) (3rd edn). Addison-Wesley: Reading, MA, 2005.
20. Goetz B, Peierls T, Bloch J, Bowbeer J, Holmes D, Lea D. *Java Concurrency in Practice*. Addison-Wesley: Reading, MA, 2006.
21. Papazoglou MP, Georgakopoulos D. Introduction: Service-oriented computing. *Communications of the ACM* 2003; **46**(10):24–28.
22. W3C. Web services description language (WSDL) version 1.2. http://www.w3.org/TR/wsdl12 [16 September 2008].
23. OWL-S. DAML Services. http://www.daml.org/services/owl-s/ [16 September 2008].
24. Thakkar S, Knoblock CA, Ambite JL, Shahabi C. Dynamically composing web services from on-line sources. *Proceedings of the AAAI-2002 Workshop on Intelligent Service Integration*, Edmonton, Alberta, Canada, July 2002; 1–7.
25. McIlraith SA, Son TC. Adapting Golog for composition of semantic web services. *Proceedings of the 8th International Conference on Principles and Knowledge Representation and Reasoning* (*KR-02*), Fensel D, Giunchiglia F, McGuinness D, Williams M-A (eds.). Morgan Kaufmann Publishers: San Francisco, CA, April 2002; 482–496.
26. Constantinescu I, Faltings B, Binder W. Large scale, type-compatible service composition. *IEEE International Conference on Web Services* (*ICWS-2004*), San Diego, CA, U.S.A., July 2004; 506–513.
27. Binder W, Constantinescu I, Faltings B, Haller K, Türker C. A multiagent system for the reliable execution of automatically composed ad-hoc processes. *Journal of Autonomous Agents and Multi-Agent Systems* 2006; **12**(2):219–237.
28. Constantinescu I, Binder W, Faltings B. Flexible and efficient matchmaking and ranking in service directories. *2005 IEEE International Conference on Web Services* (*ICWS-2005*), Florida, U.S.A., July 2005; 5–12.
29. Binder W, Constantinescu I, Faltings B. A flexible directory query language for the efficient processing of service composition queries. *International Journal of Web Services Research* 2007; **4**(1):59–79.
30. Feeley M. Polling efficiently on stock hardware. *The 1993 ACM SIGPLAN Conference on Functional Programming and Computer Architecture*, Copenhagen, Denmark, June 1993; 179–187.
31. American National Standards Institute. Information Systems—Database Language—SQL. *ANSI X3.135-1992*, November 1992.
32. Chan A, Fox S, Lin W-TK, Nori A, Ries DR. The implementation of an integrated concurrency control and recovery scheme. *SIGMOD Conference*, Orlando, FL, U.S.A., 1982; 184–191.
33. Hadzilacos T, Papadimitriou CH. Algorithmic aspects of multiversion concurrency control. *PODS '85*: *Proceedings of the Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*. ACM Press: New York, NY, U.S.A., 1985; 96–104.
34. Harris C, Reddy M, Woolf C. A high performance multiversion concurrency control protocol for object databases. *SIGMOD '92*: *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*. ACM Press: New York, NY, U.S.A., 1992; 320–320.
35. Mohan C, Pirahesh H, Lorie R. Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions. *SIGMOD '92*: *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*. ACM Press: New York, NY, U.S.A., 1992; 124–133.
36. Ammann P, Jajodia S. An efficient multiversion algorithm for secure servicing of transaction reads. *CCS '94*: *Proceedings of the 2nd ACM Conference on Computer and Communications Security*. ACM Press: New York, NY, U.S.A., 1994; 118–125.
37. Berenson H, Bernstein P, Gray J, Melton J, O'Neil E, O'Neil P. A critique of ansi sql isolation levels. *SIGMOD '95*: *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*. ACM Press: New York, NY, U.S.A., 1995; 1–10.
38. Pacitti E, Simon E. Update propagation strategies to improve freshness in lazy master replicated databases. *The VLDB Journal* 2000; **8**(3–4):305–318.
39. Elnikety S, Pedone F, Zwaenepoel W. *Generalized Snapshot Isolation and a Prefix-consistent Implementation*, 2004.
40. Fekete A, Liarokapis D, O'Neil E, O'Neil P, Shasha D. Making snapshot isolation serializable. *ACM Transactions on Database Systems* 2005; **30**(2):492–528.
41. LVM. Logical Volume Manager for Linux. http://www.tldp.org/HOWTO/LVM-HOWTO/ [16 September 2008].
42. CVS. Concurrent Versions System. http://www.nongnu.org/cvs/ [16 September 2008].
43. Subversion. http://subversion.tigris.org/ [16 September 2008].
44. Gaede V, Günther O. Multidimensional access methods. *ACM Computing Surveys* 1998; **30**(2):170–231.
45. Böhm C, Berchtold S, Keim DA. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys* 2001; **33**(3):322–373.