Multiversion Concurrency Control for Large-Scale Service Directories

Walter Binder, Ion Constantinescu, Boi Faltings, Samuel Spycher
Ecole Polytechnique Fédérale de Lausanne (EPFL)
Artificial Intelligence Laboratory
CH-1015 Lausanne, Switzerland
Email: firstname.lastname@epfl.ch

Abstract

In this paper we describe the implementation of multiversion concurrency control on the Generalized Search Tree (GiST), an index structure introduced by Hellerstein. For large-scale service directories, the need arises for a data storage system capable of handling substantial amounts of multidimensional data efficiently, as well as being able to support queries which are natural to the type of data stored in the directory. The GiST is an indexing structure that lends itself particularly well to this type of application. However, the solutions that have been proposed to address concurrency control on the GiST do not meet the requirements of large-scale service directories. The solution proposed here optimizes towards highly concurrent read accesses that are far more frequent than updates to the stored data.[1]

Keywords: *Service-oriented computing, web services, service directories, concurrency control, service composition*

1. Introduction

Web services use a collection of protocols and standards to communicate with each other over the web. In correspondence to the dynamic nature of the web, web services are entities acting in an open, dynamically changing environment without a central authority.

A service description is a formal specification of the functionality of a service. Through this specification, a software agent can either request or advertise a specific functionality. Service descriptions based on WSDL [17] define the input/output behaviour and grounding of services, whereas OWL-S [14] or WSMO [18] descriptions also specify the service semantics in logic-based formalisms (e.g., preconditions and effects of services).

The process of matching service advertisements and service requests is called *matchmaking*, and is widely considered a central issue in the context of developing useful and efficient web service systems. On a higher level, matchmaking allows for *web service composition* by coupling the functionalities of existing services to provide a new service [16, 13, 19, 9].

Due to the open and changing environment, matchmaking has to operate without any specific prior knowledge of existing services. Services are therefore indexed in directories, and the main goals for the implementation of these directories and the matchmaking algorithms are to maximize the success rate as well as the efficiency of queries, which is crucial within the context of composition on a large number of available services. The directory presented in [8, 5] indexes service descriptions according to their input/output behaviour, which allows the efficient access to those service descriptions that are relevant to a particular service composition problem [9]. The input/output characterization of services is considered multidimensional data, and the index structure used by the directory is based on the Generalized Search Tree (GiST), introduced as an unifying framework to build balanced search trees by Hellerstein [10].

In previous work we found that incremental retrieval of query results that are obtained by a best-first search of the directory tree greatly improves the efficiency of service composition [6, 3, 7], since the service composition algorithm does not have to wait for the directory to compute the complete result. 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, since it may not have to compute the complete result set (if the service composition client is satisfied with the retrieved partial results, it may close the result set, hence avoiding the computation of the complete result set).

[1]The work presented in this paper was partly carried out in the framework of the EPFL Center for Global Computing and supported by the Swiss National Funding Agency OFES as part of the European projects KnowledgeWeb (FP6-507482), DIP (FP6-507483), and CASCOM (FP6-511632).

Incremental retrieval of the results of a directory query and interleaving service composition with directory accesses requires the directory to support so-called "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 [7] relies on an unchanged directory index structure during the whole query. For this reason, our directory supports *read sessions*, which guarantee the client a consistent view of the directory data [5].

In this paper we introduce a novel concurrency control scheme that fits particularly well to our service directory. The scheme builds on the ideas of multiversion concurrency control [1, 2]. Unlike traditional database systems which use locks for concurrency control, multiversion concurrency control maintains data consistency by using a multiversion model. This means that while querying a database, each transaction sees a snapshot of data (a database version) as it was some time ago, regardless of the current state of the underlying data. This protects the transaction from viewing inconsistent data that could be caused by (other) concurrent transaction updates on the same data, providing transaction isolation for each database session. The main advantage of using the multiversion model of concurrency control rather than locking is that reading never blocks writing and writing never blocks reading, because the two operate on separate data versions. Nowadays multiversion concurrency control is used by many commercial database systems.

In contrast to traditional multiversion concurrency control, our scheme is not limited to maintaining multiple versions of stored data (i.e., service descriptions), but we provide multiple versions of the whole index structure (the GiST mentioned above). We implemented a generic extension to the GiST framework, which we call Multiversion GiST (MVGiST). A MVGiST consists of a *write tree*, which corresponds to the original GiST and is used for updates, and a set of *read trees*, which provide read-only snapshots of the tree at different times. The MVGiST is a generic Java package that can be reused in different contexts, it is not limited to our service directory.

In our directory implementation, which relies on our MVGiST package, a read session (used for service composition) is confined to a single read tree (the most recent one upon initiation of the read session) and all read accesses are non-blocking, boosting scalability. Periodically, the write tree is copied into a new read tree, which will be used by new read sessions. The copying algorithm avoids replicating tree nodes that have not changed since the last copying, thus preserving memory and speeding up the copying process.

The main contribution of this paper is the presentation of the MVGiST, an extension of the GiST framework to support concurrency. The rest of this paper is structured as follows: Section 2 reviews our implementation of the GiST framework. Section 3 introduces the MVGiST, gives an overview of its API, explains how it can be extended and customized, and discusses the creation and management of multiple GiST versions. Section 4 outlines how our service directory exploits the MVGiST. Finally, Section 5 discusses related work on concurrency control for the GiST and Section 6 concludes this paper.

2. Generalized Search Tree (GiST)

This section gives an overview of the GiST. Our extension, the MVGiST, is introduced in the next section.

2.1. Search Trees

A search tree is a balanced tree with (usually) high fanout. The internal nodes are used as directories, and the leaf nodes contain the actual data. Every internal node has a series of keys and pointers 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). But essentially a search key may be any 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 rebalancing methods to the user, arbitrary search trees may be constructed, which is exactly what is accomplished with the GiST.

As the name implicates, the Generalized Search Tree is a search tree that is independent of the data types it indexes, it provides basic search tree logic and supports queries which 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.

2.2. Structure

A GiST is a balanced search tree of variable fanout between kM and M, where M is the maximum number of child nodes and $2/M \leq k \leq 1/2$, 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 tuple referenced by the leaves of the tree can instantiate all the variables.

2.3. Properties

These 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 indicated tuple.
- 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. with R-trees, for any entry (p', ptr') reachable from ptr, p' can express a predicate that is entirely orthogonal to p. This aspect is interesting with respect to the classification of data 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. This can, but need not be the logical union of these pointers.
- **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 $\pi =$ `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. The penalty is typically some metric for the increase in size of p_1 to union($E_1.p_1, E_2.p_2$).
- **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.

Note that in our implementation we omit `compress(E)` and `decompress(E)`. This is because for our purposes speed is more important than size of the tree. Also, 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.

2.5. Implementation

The implementation presented here is the version of the GiST that served as a basis for the realization of Multiversion Concurrency Control on the GiST as described in detail in Section 3. It is implemented in Java and makes use of abstract classes to allow for user-implementation of the key methods described above. See Figure 4 for a class diagram of the GiST. The classes `And`-, `True`-, `Not`-, and `OrPredicate` extend `Predicate` and exist simply to allow more complex predicate classes to be built easily. The `GiSTList` class is a doubly linked list containing `GiSTListNode` and is used to store GiST nodes.

3. Multiversion Generalized Search Tree (MVGiST)

3.1. Multiversion Concurrency Control

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 the GiST, 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 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.

Multiversion concurrency control 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. Since writes do not overwrite each other, this gives greater flexibility to the database system in its ordering of conflicting operations. [2] gives an in-depth view of multiversion concurrency control for database systems.

3.2. Requirements for the MVGiST

Because a directory query in our system may retrieve a large number of matching entries, it is important to support incremental access to the results to avoid wasting network bandwidth. We therefore offer *read sessions* to allow a user to issue queries and retrieve the results in chunks of limited size. The read session has to offer a consistent view of the directory, i.e., concurrent updates (service registration, update and removal) of the directory must be invisible to a read session. Proposed solutions to concurrency control in multidimensional index structures [11, 15] synchronize individual operations on the tree, whereas our directory supports long-lasting read sessions which require unchanging structure and content of the whole tree. The following assumptions underly the design of our concurrency control mechanism:

1. Read accesses (i.e., queries within read sessions and the incremental retrieval of results) will be 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 directory data; they have to be isolated from concurrent updates of the directory data and index structure.[2]
4. Read accesses shall not be delayed.
5. Updates may become visible with a significant delay, but feedback concerning the update (success/failure) shall be returned immediately. This implies that a read session may see an outdated (but consistent) version of the directory data. However, as there is no guarantee that a retrieved service description remains up-to-date until invocation of the corresponding service, the potentially outdated directory data does not introduce any new problems in practice. Failure upon service invocation is always possible.
6. The duration of a read session can be limited (timeout).

3.3. Design

The indicated requirements for the MVGiST with respect to concurrent access leads us to a design in which the readers access a tree which is separate from the write 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.

The following is a generic (independent of service directories) outline of the algorithm for read tree management: After instantiation of the MVGiST, the write tree is populated with existing data (e.g., the service descriptions may be read from durable storage). Then a `createNewVersion` procedure is called. This procedure recurses down the tree, instantiates a read node for every write node whose read twin reference is null (all nodes, when `createNewVersion` is first called), and places an additional reference to it in the array of child nodes of the new read node's parent read node, thus creating a complete read tree. The new read tree root is inserted into a list of read tree roots. 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. 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.

After a certain number of write tree modifications, the `createNewVersion` 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 `createNewVersion` call. The new read root is now added to the list of read roots, and incoming readers now receive the new read root to access the read tree. The MVGiST keeps track of the number of readers accessing a read tree, and when all readers have left the oldest read tree, the reference to its root is nullified. The read nodes that are not referenced by other read tree versions will now gradually get garbage collected. Figure 1 visualizes the process of read tree node creation. Bear in mind that this diagram is only schematic, and does not mean that a new read tree is created after every write node update.

Note that upward navigation within the read trees is impossible, because this could lead a reader to outdated versions of the tree.

3.4. Properties

In addition to the properties stated for the GiST, the MVGiST has the following invariant property:

[2] As described in [7], our directory implements a best-first search algorithm that maintains a heap of references to tree nodes. These nodes are processed by a tree traversal in the order of their estimated relevance. The traversal can be a long lasting operation, because the result set is generated lazily: Once a result has been returned to the client, the traversal is suspended until the client asks for more results. The correctness of this search algorithm relies on an unchanged tree structure during the whole traversal. The design of the MVGiST was motivated particularly by this use case.

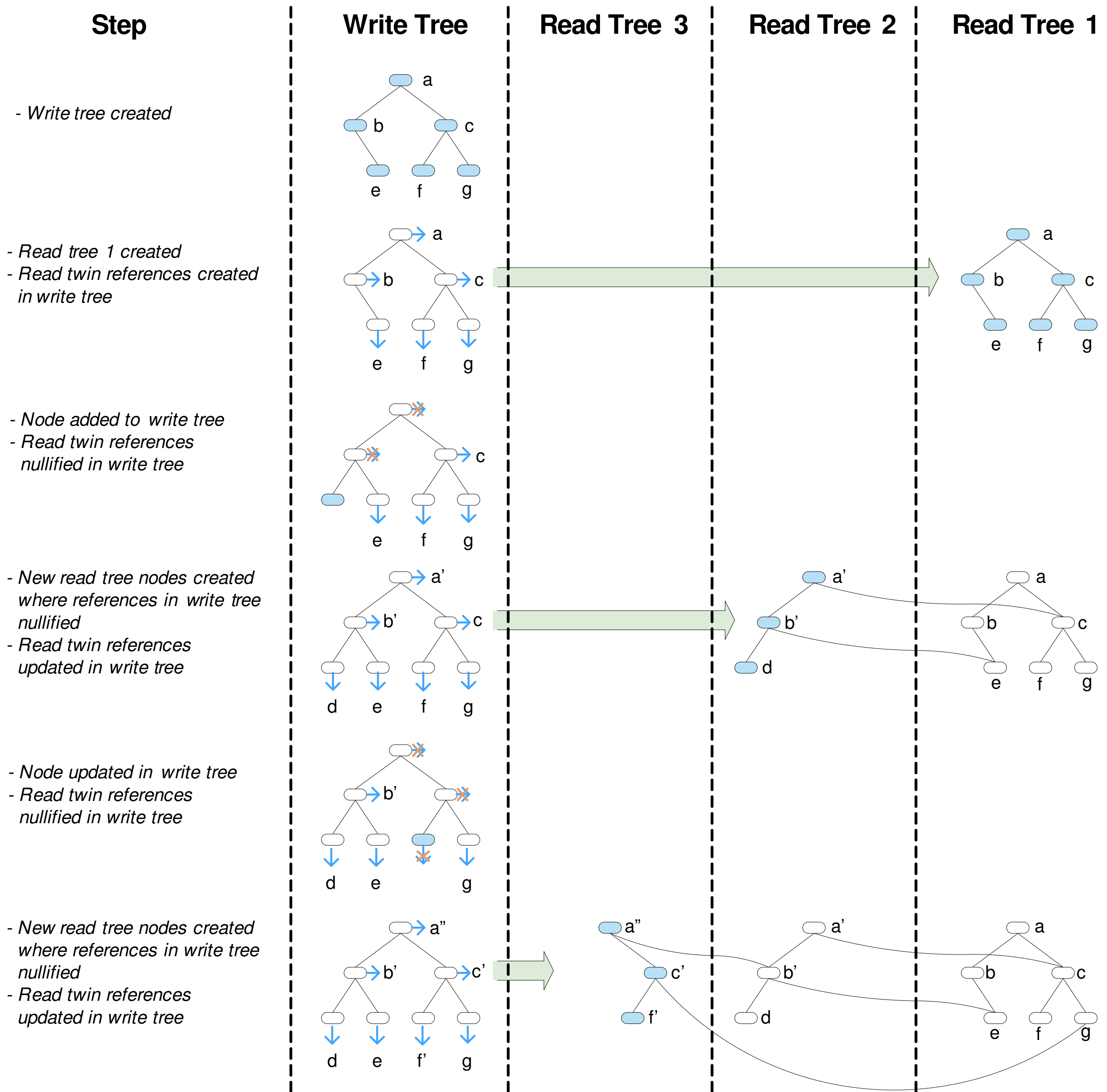

Figure 1. The MVGiST read tree creation process.

- Within a read session, a reader's view of the tree never changes.

3.5. Use Cases

Back within the context of service directories, we can distinguish two use cases: read access and write access on the MVGiST. These make reference to a control layer, the session manager (see Section 4).

Use Case 1: Read Access (see Figure 2)

1. The reader requests a read session from the service directory. The session API creates a new instance of a read session, and passes it a reference to the newest version of the read tree root node.

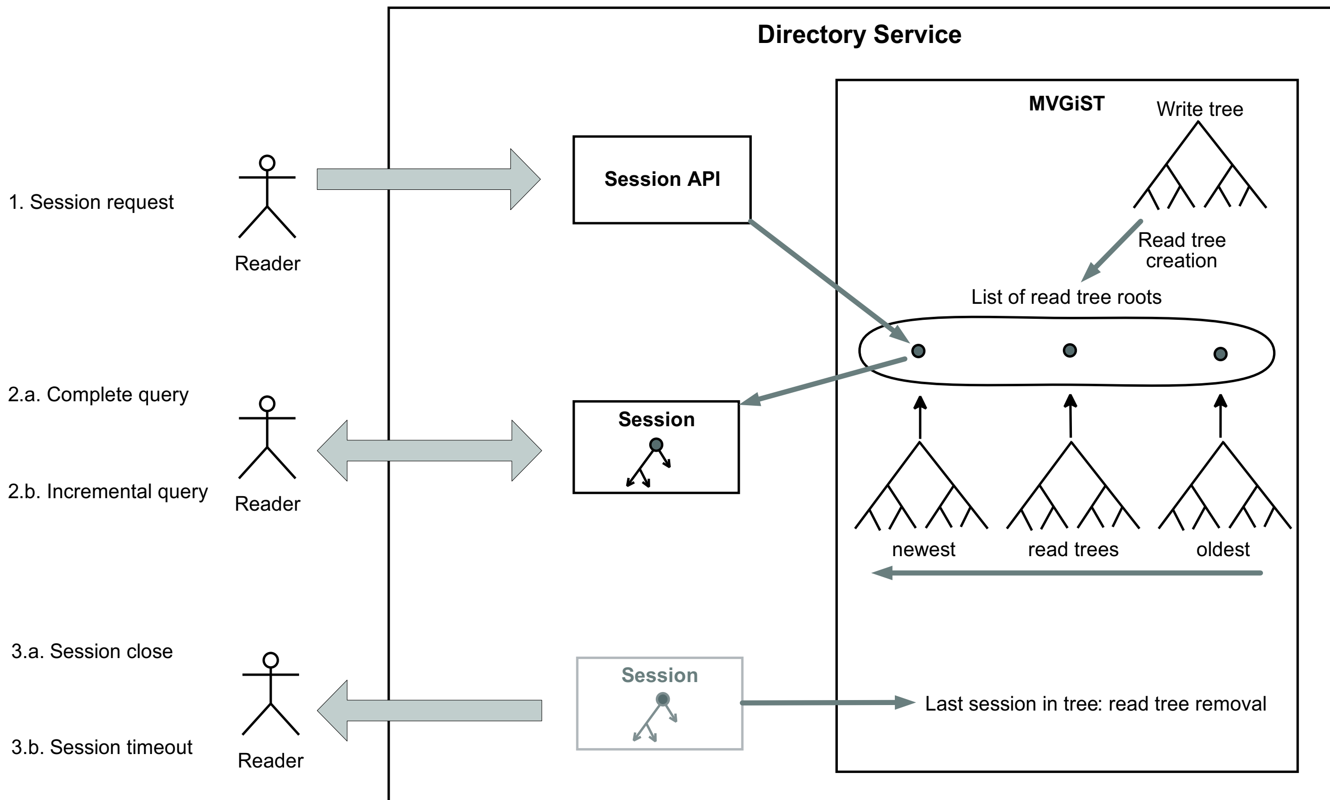

Figure 2. Read access on the MVGiST.

2.a. Complete query: the reader can now start a query that retrieves all leaf nodes consistent with the query predicate on its version of the directory.

2.b. Incremental query: the reader retrieves a subset of the consistent leaf nodes in rank-ordered chunks of limited size.

3.a. The reader closes its read session and leaves the service directory.

3.b. If the reader should remain too long in the directory (unsuccessful matching or crash), the read session eventually times out. The session API decrements the number of readers present in that specific read tree, and if there are none left, initiates removal of the read tree by the MVGiST.

Use Case 2: Write Access

1. The writer requests a write session from the service directory. The session manager creates a new instance of a write session, and passes it a reference to the write tree root node.

2.a. The writer performs the insertion of a new leaf node containing a specific service according to a certain predicate in the write tree.

2.b. The writer searches for an existing leaf node according to a specific predicate, and deletes it from the write tree.

2.c. The writer deletes a set of nodes according to a certain predicate.

3. The writer closes its write session and leaves the service directory.

Of course the writer can combine insertion and deletion to modify an existing node; if the predicate is modified, this may change the position of the node in the tree.

3.6. Implementation

The implementation of the MVGiST is similar to the original GiST, with the addition of all classes and methods necessary for read tree creation, access, and removal. Figure 5 shows a class diagram of the MVGiST package.

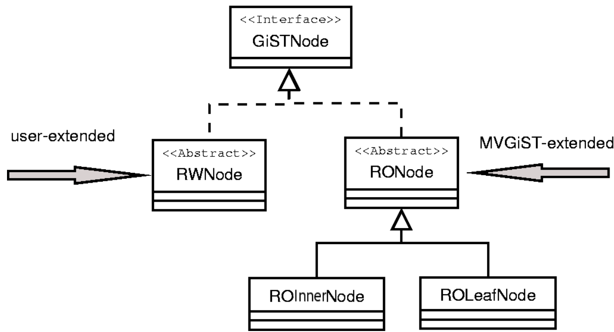

Figure 3. The node inheritance model.

Changes to the original GiST The following are the principal implementation differences to the original GiST:

- Because the domain to be indexed (the domain of services) in the MVGiST has no linear ordering, the corresponding methods for searching (`findMin` and `next`) have been omitted.
- There are now two versions of the tree, one write tree which is similar to the GiST, and a read-only tree which does not contain the user-implementable methods and other methods related to updating the tree (read tree node classes therefore do not need to be extended).
- The MVGiST class contains additional methods for read tree creation, removal, as well as read root storage and retrieval.
- In view of the fact that the directory must contain a large number of services, some effort was taken to reduce the size of nodes. The original GiST nodes contain child reference arrays of type `GiSTEntry`, which contains `GiSTKey` and the reference to the child node. In the MVGiST, the `GiSTEntry` class was removed, and nodes now directly contain arrays of child nodes and their own keys. Also, for read nodes, the inner nodes do not contain an empty data object, and leaf nodes do not contain an empty array of child nodes, as was the case with the original GiST.
- Because upwards navigation in read trees is impossible, there is no parent node reference in a read node.

Tree Nodes The existing commonalites 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 nodes.

The `GiSTNode` interface defines aspects that all nodes contain. Two abstract classes implement this interface: `RONode` (nodes belonging to the read tree) and `RWNode` (nodes belonging to the write tree). The `RONode` class is extended by `ROInnerNode` and `ROLeafNode` for the above mentioned reason (Section 3.6). The `RWNode` 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).

4. Synchronization and Session Management

This section presents the general design of the session management API. It is here that the MVGiST interfaces are integrated with matchmaking intelligence and the larger web service architecture. Here we discuss only part of the design, as far as the MVGiST is concerned.

4.1. Synchronization

With the implementation of MVGiST, we now have a service directory in which readers can concurrently access the data without any danger of incoherence due to writers simultaneously accessing the tree. For every query, a separate thread can simply be generated to access the latest version of the read tree.

However, the write tree is still exposed to the same concurrent access problems as the GiST. As mentioned, previous research work has addressed the issue of concurrency control in generalized search trees [11]. The proposed solution is a hybrid locking mechanism involving predicate-based locking and two-phased locking. While probably a powerful solution to write access on our service directory, it is not necessary to implement this rather complex mechanism. Since directory modification is anticipated to be far less frequent than read access, a simple solution is to serialize the update tasks on the write tree and the read tree creation task. With the advent of Java 1.5, predefined concurrency control mechanisms are at our disposal: a straightforward solution is to share a *fair* (FIFO) semaphore as a mutual exclusion lock between the accessing threads on the root of the write tree. The `Semaphore` class in `java.util.concurrent` allows for FIFO ordering of threads that try to acquire a lock, this in contrast to the classical locking mechanism of Java.

4.2. Session Management

As far as the MVGiST is concerned, session management involves the creation of read and write sessions, assignment of IDs to these sessions, and their destruction on timeout. Based on the session management, the service directory decides when to create a new read tree (depending

on the activity of writers), and when to destroy the oldest read tree (after the last read session on that tree either leaves or times out). The sessions themselves incorporate methods for accessing the tree, the read sessions have fields and methods for complete queries and incremental rank-based queries, directly accessing the `getChildAt`, `getData`, and consistency checking methods on the tree. All tree accesses spin off into separate threads, which are enqueued if they access the write tree.

4.3. Tree Replication

An important issue in the design of the control layer is the desired freshness of the most recent read tree (differences between write and read tree). The parameters involved are the frequency of tree modification by writers and the frequency of read tree creation. The freshness of the most recent read tree R is indirectly proportional to the number of changes in the write tree since the creation of R. In our implementation, the number of modifications can be easily tracked by counting the number of (non-null) read twins that are nullified. This metric may be used to decide when to create a new read tree.

Directly connected to this is the problem of memory usage, which corresponds (appoximatively) to the number of read trees in existence plus the write tree itself. Duration of reader timeout t and time between tree replication calls d control how many read trees remain in existence; the number of read trees in memory is $1 + \lceil \frac{t}{d} \rceil$. A very approximate measure for the memory used by the system is $S_{write} + S_{read} * (1 + \lceil \frac{t}{d} \rceil)$, where S_{write} is the size of the write tree and S_{read} is the average read tree size. In the above formulae, the parameter to be defined by the control layer is d (and to a lesser extent t, the definition of which is probably constrained within a certain range). Compromising between memory usage and read tree freshness is therefore a central point of focus for the control layer designer. In order to minimize memory consumption, d and t should be chosen so that $t \leq d$. In this case, only two read trees have to be kept in memory.

Figure 6 gives a partial class diagram of the session management API, this diagram only incorporates the MVGiST-related activities of the service directory.

5. Related Work

Below we summarize prevailing approaches to add concurrency control to the GiST [10] or to multidimensional index structures in general.

In [11] the authors present a concurrency control protocol for the GiST that is based on an extension of the link technique originally developed for B-trees [12]. This approach avoids holding node locks during I/O and achieves repeatable read isolation with a combination of predicate locks and two-phase locking of data records.

An alternative concurrency control mechanism for the GiST is discussed in [4], 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 [4] the authors conclude that while granular locking causes lower lock overhead than predicate locking, it may reduce concurrency.

In [15] the authors describe an enhanced concurrency control algorithm that reduces blocking time during split operations. 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, our 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. Though, our tree replication algorithm minimizes the allocation of nodes and allows to share unchanged subtrees between different read trees. To the best of our knowledge, the MVGiST is the first multiversion concurrency control scheme for the GiST.

6. Conclusion

In this paper we presented the Multiversion Generalised Search Tree (MVGiST), an extension of the well established Generalised Search Tree (GiST) that adds a novel multiversion concurrency control mechanism to the GiST. Thanks to the concurrency control scheme of the MVGiST, it is possible to support “long reads” that guarantee a consistent tree structure, despite of potentially concurrent updates to the MVGiST. Moreover, read access is non-blocking, which is essential to achieve high scalability.

The MVGiST is an ideal data structure to index web services based on their input/output behaviour, as it is required by some recent service composition algorithms. The implemented multiversion concurrency control scheme ensures that complex directory queries, such as queries issued by service composition algorithms, have a consistent view of the directory index. Because reads are much more frequent than writes in the case of a service directory, the support of non-blocking reads is key to efficiently supporting a large number of concurrent requests.

While we developed the MVGiST as index structure for our directory, we designed it as a generic, reusable package that can be extended and customized for other (database) applications as well.

References

[1] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, 1981.

[2] P. A. Bernstein and N. Goodman. Multiversion concurrency control – theory and algorithms. *TODS*, 8(4):465–483, Dec. 1983.

[3] W. Binder, I. Constantinescu, and B. Faltings. A directory for web service integration supporting custom query pruning and ranking. In *European Conference on Web Services (ECOWS 2004)*, pages 87–101, Erfurt, Germany, Sept. 2004.

[4] K. Chakrabarti and S. Mehrotra. Efficient concurrency control in multidimensional access methods. In *SIGMOD '99: Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 25–36, New York, NY, USA, 1999. ACM Press.

[5] I. Constantinescu, W. Binder, and B. Faltings. Directory services for incremental service integration. In *First European Semantic Web Symposium (ESWS-2004)*, pages 254–268, Heraklion, Greece, May 2004.

[6] I. Constantinescu, W. Binder, and B. Faltings. An extensible directory enabling efficient semantic web service integration. In *3rd International Semantic Web Conference (ISWC 2004)*, Hiroshima, Japan, Nov. 2004.

[7] I. Constantinescu, W. Binder, and B. Faltings. Flexible and efficient matchmaking and ranking in service directories. In *2005 IEEE International Conference on Web Services (ICWS-2005)*, Florida, July 2005.

[8] I. Constantinescu and B. Faltings. Efficient matchmaking and directory services. In *The 2003 IEEE/WIC International Conference on Web Intelligence*, pages 75–81, 2003.

[9] I. Constantinescu, B. Faltings, and W. Binder. Large scale, type-compatible service composition. In *IEEE International Conference on Web Services (ICWS-2004)*, pages 506–513, San Diego, CA, USA, July 2004.

[10] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In U. Dayal, P. M. D. Gray, and S. Nishio, editors, *Proc. 21st Int. Conf. Very Large Data Bases, VLDB*, pages 562–573. Morgan Kaufmann, 1995.

[11] M. Kornacker, C. Mohan, and J. M. Hellerstein. Concurrency and recovery in generalized search trees. In J. M. Peckman, editor, *Proceedings, ACM SIGMOD International Conference on Management of Data: SIGMOD 1997: May 13–15, 1997, Tucson, Arizona, USA*, 1997.

[12] P. L. Lehman and s. Bing Yao. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst.*, 6(4):650–670, 1981.

[13] S. A. McIlraith and T. C. Son. Adapting Golog for composition of semantic web services. In D. Fensel, F. Giunchiglia, D. McGuinness, and M.-A. Williams, editors, *Proceedings of the 8th International Conference on Principles and Knowledge Representation and Reasoning (KR-02)*, pages 482–496, San Francisco, CA, Apr. 22–25 2002. Morgan Kaufmann Publishers.

[14] OWL-S. DAML Services, http://www.daml.org/services/owl-s/.

[15] S. I. Song, Y. H. Kim, and J. S. Yoo. An enhanced concurrency control scheme for multidimensional index structures. *IEEE Transactions on Knowledge and Data Engineering*, 16(1):97–111, 2004.

[16] S. Thakkar, C. A. Knoblock, J. L. Ambite, and C. Shahabi. Dynamically composing web services from on-line sources. In *Proceeding of the AAAI-2002 Workshop on Intelligent Service Integration*, pages 1–7, Edmonton, Alberta, Canada, July 2002.

[17] W3C. Web services description language (WSDL) version 1.2, http://www.w3.org/tr/wsdl12.

[18] WSMO. Web Service Modeling Ontology, http://www.wsmo.org/.

[19] Wu, Dan and Parsia, Bijan and Sirin, Evren and Hendler, James and Nau, Dana. Automating DAML-S Web Services Composition Using SHOP2. In *Proceedings of 2nd International Semantic Web Conference (ISWC2003)*, 2003.

«interface»
GiSTconstants
CONST_MaximumEntries: int
CONST_MinimumEntries: int
CONST_MaxPenalty: int

GiST
isOrdered: boolean
treeModelListeners: Vector
GiST()
getAllNodesNo()
getLeafNodesNo()
CreateNode()
getNode()
findMin()
nextOnLevel()
prevOnLevel()
next()
search()
delete()
condenseTree()
getGiSTRoot()
insert()
adjustKeys()
chooseSubtree()
searchMinPenalty()
split()
getChild()
getChildCount()
getIndexOfChild()
isLeaf()

GiSTpenalty
amount: double
GiSTpenalty()
lessThan()

GiSTnode
level: int
numEntries: int
minEntries: int
maxEntries: int
GiSTnode()
getMinimum()
isUnderfull()
tooFew()
getPosition()
getEntry()
setEntry()
insert()
search()
union()
pickSplit()
newNode()

parent 0..1
root 0..1
ptr 0..1
entries *

GiSTentry
level: int
data: Object
GiSTentry()
Penalty()
newEntry()
compare()

key 0..1
GiSTkey

andPredicate
andPredicate()
consistent()
searchDone()

truePredicate
truePredicate()
consistent()
searchDone()

notPredicate
notPredicate()
consistent()
searchDone()

orPredicate
orPredicate()
consistent()
searchDone()

q1 0..1
q2 0..1
q 0..1

GiSTpredicate
consistent()
searchDone()
searchChilds()

GiSTlist
GiSTlist()
append()
prepend()
join()
getFirst()
getNumEntries()
getEntry()
isFull()
isEmpty()
isNotEmpty()

front 0..1
rear 0..1

GiSTlistNode
entry: Object
GiSTlistNode()

next 0..1
prev 0..1

Figure 4. Class diagram of the Java GiST package.

Figure 5. Class diagram of the MVGiST package.

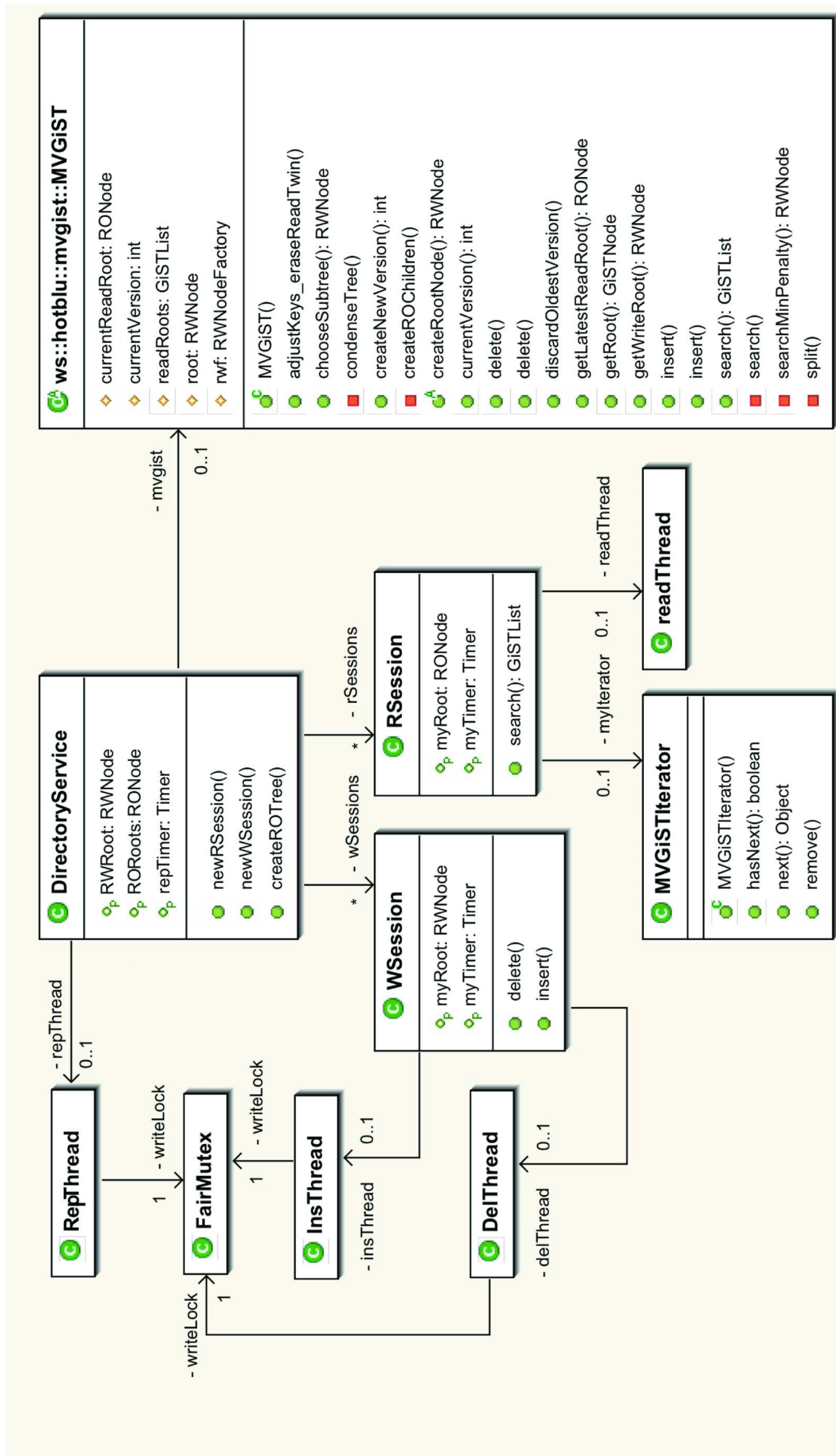

Figure 6. Partial class diagram of the session management API.