Finding Data in the Neighborhood

André Eickler   Alfons Kemper   Donald Kossmann

Universität Passau
Fakultät für Mathematik und Informatik
D-94030 Passau, Germany

⟨*last name*⟩@db.fmi.uni-passau.de
http://www.db.fmi.uni-passau.de/

Abstract

In this paper, we present and evaluate alternative techniques to effect the use of location-independent identifiers in distributed database systems. Location-independent identifiers are important to take full advantage of migration and replication as they allow accessing objects without visiting the servers that created the objects. We will show how a distributed index structure can be used for this purpose, we will present a simple, yet effective replication strategy for the nodes of the index, and we will present alternative strategies to traverse the index in order to dereference identifiers (i.e., find a copy of an object given its identifier). Furthermore, we will discuss the results of performance experiments that show some tradeoffs of the proposed replication and traversal strategies and compare our techniques to an approach that uses location-dependent identifiers like many systems today.

1 Introduction

Large distributed systems are beginning to play a dominant role in the information market-place. Companies are building so-called Intranets to provide access to their data from offices all around the world, and the WWW is attracting an ever growing number of users and providing access to an ever growing amount of data.

To achieve acceptable performance in large distributed information systems, it is important to replicate and to move (migrate) data close to sites where the data is frequently used. Both migration and replication can significantly reduce communication costs and help load-balance a system by storing frequently accessed objects on different servers. Replication can, in addition, improve the fault tolerance of a system since copies of replicated objects are available even if certain servers are down or unreachable.

The benefits of migration and replication can, however, only be exploited if the system provides powerful facilities to find all copies of an object given the object's identifier. Principally, a distributed database can be organized in two different ways in order to find objects: it can use location-dependent or location-independent identifiers. Location-dependent identifiers contain the address of a server that stores a copy of the object and/or the addresses of all other servers that have a copy of the object; identifiers can, thus, be dereferenced by visiting that server. One big advantage of this approach is its simplicity. One big disadvantage of this approach is that it cannot take advantage of migration and replication in many situations; for example, if a copy of an object is located in Munich (Germany) and the identifier of the object points to a server in San José (USA), then a request from Passau (Germany) to read the object will be directed (at potentially high cost) to San José before the object can (cheaply) be retrieved from Munich.

The benefits of migration and replication can only be fully exploited if location-independent identifiers are used. To dereference location-independent identifiers, the system typically maintains some sort of distributed index which maps every identifier to a list of addresses of servers that have a copy of the referenced object. Ideally, the nodes of the index structure are stored on servers in such a way that objects can be retrieved from the neighborhood without visiting distant servers. While this approach is effective to support migration and replication, it incurs additional cost for probing the distributed index whenever an identifier of a remote object is dereferenced. (Of course, local objects can be accessed without probing the distributed index.) In this paper, we will show how location-independent identifiers can be implemented in a distributed system, and we will focus especially on techniques to minimize the cost to probe the distributed index by means of caching, replication and specialized traversal strategies. In addition to these developments, we will present the results of performance experiments that evaluate some tradeoffs of the proposed techniques and compare these techniques to an approach that uses location-dependent identifiers.

Throughout this paper, we will concentrate on techniques to find copies of objects as efficiently as possible in the presence of migration and replication; we will not discuss strategies that actually decide which objects to replicate and migrate, or protocols that keep all the replicas of

Proceedings of the 23rd VLDB Conference
Athens, Greece, 1997

an object consistent. We recognize that the choice for one such strategy and protocol depends on the target application, and therefore designed our techniques to be applicable by any kind of system.

Both location-dependent and independent identifiers are used in existing systems. URLs (uniform resource locators) in the WWW are probably the most prominent examples for location-dependent identifiers; among distributed database systems such as SHORE [CDF$^+$94] many other examples can be found. Location-independent identifiers have been proposed for the domain name service [Moc87] and the X.500 directory [CCI89]—though in a very different application area. In the research literature, location-independent identifiers were used for mobile telecommunication systems [AHMW94, JLS$^+$95]. We will discuss the differences to our approach in detail towards the end of this paper after having fully presented our techniques.

The remainder of this paper is organized as follows. Section 2 shows how distributed databases can be organized using location-dependent and independent identifiers. The following three sections contain all the details of our approach and describe alternative ways to implement and optimize the use of location-independent identifiers: Section 3 describes the distributed index structure, Section 4 shows how caching and replication of index nodes can be exploited, and Section 5 presents alternative strategies to probe and traverse the distributed index. Section 6 summarizes the results of performance experiments obtained using a simulation model. Section 7 discusses related work and Section 8 concludes this paper.

2 Object Identification in Distributed Databases

In this section we will give an overview of how identifiers can be managed in a distributed system. As stated in the introduction, one of the main design choices is whether location-dependent or location-independent identifiers are to be used. While location-dependent identifiers are straightforward to apply in practice, there is a large design space of alternative techniques to effect and tune the use of location-independent identifiers. We will present that design space in the following sections and focus on the fundamental ideas here.

2.1 Location-Dependent Identifiers

In this approach, every object identifier contains the address of the *home server* of the object, i.e., the server that created the object. The home server stores a copy of the object and/or a collection of *forwards* for that object; a forward is a simple data structure that contains the address of (another) server that stores that object. Using location-dependent identifiers, thus, an object can be accessed by simply contacting the object's home server. The home server will reply by either sending a copy of the object, if it stores one, or by returning the collection of forwards. Migration and replication of an object involves updating the home server's collection of forwards for that object; if, for example, an object is replicated, the home server establishes a new forward for the new replica. In order to generate unique object identifiers for new objects, servers maintain counters and record their values in the *unique* fields of new object identifiers or apply one of the techniques described in [EGK95], just as in a centralized system.

2.2 Location-Independent Identifiers

Using location-independent identifiers, objects can be accessed, migrated, and replicated independently from where they were born. The goal is to avoid any permanent dependency to one specific server (such as the home server) in order to be able to access the object if that server is down or to reduce cost if that server is, say, heavily loaded. This goal is achieved by maintaining a distributed tree-structured index.

The nodes of this index are stored on a number of dedicated servers organized in a hierarchy. The hierarchy reflects the topology of the network; typically, the communication between a name server and its child or parent is cheap (one hop) whereas the communication with a name server in a different branch of the hierarchy is expensive (several hops). The number of servers in the hierarchy and the height of the hierarchy depend on the size of the database, the workload, and the structure of the network. (See [AHMW94] for a detailed discussion.)

The nodes of the index are stored at servers of the hierarchy as follows: the root node is stored by the root server of the hierarchy, intermediate nodes are stored by intermediate servers and leaf nodes are stored by leaf servers. Probing the index to find the location(s) of an object can be carried out in a number of ways (see Section 5) and is influenced by caching and replication (see Section 4), but the basic pattern is always the same. First, the server checks, if it has, say, a copy of the root of the index. If not, the server asks its parent server in the server hierarchy for help. If the parent server is down or cannot help, the server will ask the grandparent server and so on, until (in the worst case) the server will ask the root server which definitely has a copy of the root of the index. Once the root has been found, the pointers stored in the root node can be followed to the intermediate nodes and so forth until a leaf node is reached. When the leaf node is reached, the information about the locations of the object can be extracted and the cheapest copy of the object can be retrieved.

2.3 Discussion

In the following, we will briefly compare the tradeoffs of the two approaches described above. First of all, both approaches perform equally well if a server creates new objects or accesses locally available objects because neither approach requires interaction with other servers: as we will see, the distributed index is constructed in such a way that it need not record new objects as long as they are not migrated or replicated, and in any case, local copies of objects can be read without asking other servers. Creating new objects and accessing local copies of objects at no additional cost is a very important property—we expect that many applications will predominantly work with local objects.

The differences between the two approaches become apparent if a server intends to access remote data. In this case, the use of location-dependent identifiers involves visiting the home server of the object. If the home server is down, heavily loaded or "far away" and a copy of the object is available at a cheaper or "nearby" server, the use of location-dependent identifiers misses a nice chance to benefit from replication (or migration). Only the use of location-independent identifiers makes it possible to benefit from migration and replication in the best possible way because objects can often be found at cheap servers without visiting their expensive home servers. On the negative side, accessing remote data with location-independent identifiers always requires the traversal of a distributed index, thereby possibly visiting several servers at high cost. Therefore, it is crucial to minimize the cost of such traversals—this is the main focus of this paper.

The two approaches also potentially differ in the costs to migrate and replicate objects: updating a distributed index vs. updating collections of forwards. We expect these costs to be of minor importance because migrations and replications are rarely performed compared to object reads and modifications.

3 The Distributed Index

In this section, we will describe our distributed index for location-independent identifiers. Our index is adapted from the B-link tree [LY81] and its distributed derivatives [JK93, Lom96] with slight modifications to the layout of the nodes, the way pointers between parent and child nodes are implemented and the initialization of the index. We will first describe the structure of the index and its nodes, then index maintenance (initialization, splitting of nodes, reorganizations), and finally how objects can be found in the presence of concurrent migrations.

3.1 Structure of the Index and Its Nodes

Figure 1 shows an example index. The figure shows six servers of a hierarchy with three levels, three index nodes, and several objects with their (location-independent) identifiers. The servers and their interconnects are represented in light gray; the nodes of the index and the objects (denoted by ovals) are printed in black. For ease of presentation and without loss of generality, we will assume throughout this paper that all objects of the database are stored at leaf servers of the server hierarchy.

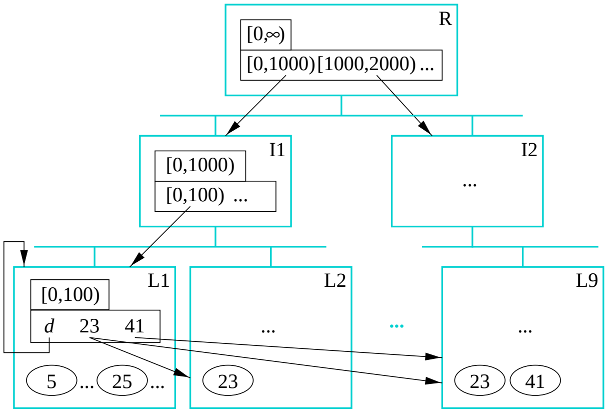

Figure 1: Example Server Hierarchy

From the figure, it becomes immediately apparent that root and intermediate nodes of our index contain pointers to their children just as in any conventional tree-index. There are two differences to conventional trees. The first difference is the structure of the leaf nodes of the index, and the second difference is the way that the pointers in the index are implemented.

The leaf nodes of our index have the following structure. Every leaf node stores exactly one entry to record the so-called *default route* and a list of so-called *exceptions* for migrated and replicated objects. The default entry contains the address of the server that generated or would generate all objects with identifiers that fall into the range of identifiers covered by the leaf node; for example, if identifiers 0 to 99 are reserved for objects created at Server L1, then the default entry of a leaf node for the range [0,100) or any subrange hereof would point to Server L1; this situation is shown in Figure 1 where the default entry (labeled as *d*) of the leaf node stored by Server L1 points to Server L1. Storing default routes is an effective technique to reduce the size of the distributed index: we expect that many objects of a distributed database system are never migrated or replicated. These objects need not be registered individually in the distributed index as they are implicitly registered in the default route so that space in the index is only invested for the important cases of migrated and replicated objects.

Pointers in the distributed index are implemented as plain server addresses rather than storing server *plus* disk addresses. This is true for the pointers of the default route and the exceptions stored in a leaf as well as for the parent-child pointers of the root and intermediate nodes. In addition to the global distributed index, every server must, therefore, maintain two local indexes in order to map identifiers (for objects) and key ranges (for index nodes) to disk addresses, and we have a two-step process: (1) use our global distributed index to find the right server, and (2) use the server's local indexes which can be ordinary B$^+$-trees to find the object or node at that server. Obviously, this approach can result in additional disk I/O to probe a server's local index, but it also has important advantages: (1) it reduces the size and height of the global index as more entries can be stored per node, and (2) it increases the local autonomy of servers. As we will see in Section 5, this approach has one more crucial advantage: it allows to start probing the global index at every of its nodes rather than being restricted to starting traversals at the root.

3.2 Initialization, Splitting, Reorganization

At the beginning when no objects have yet been created, our distributed index is initialized bottom-up. First, every leaf server of the server hierarchy stores exactly one leaf node. The range of the leaf node established at a server is identical with the range of identifiers pre-allocated for the objects to be created at that server, the default route of the leaf node is set to point to the server itself, and the list of exceptions is naturally empty to begin with. Based on these leaf nodes, the intermediate nodes and the root are constructed: the root is stored at the root server, and intermediate nodes are stored at every intermediate server in such a way that the intermediate nodes stored at a server contain pointers to all index nodes stored at child servers of that server. This approach is very intuitive, and its effects can be seen in Figure 1 in which the intermediate node stored at Server I1 is responsible for the leaf node stored at Server L1. This initial state of the index is not changed even if objects are created at the servers; migration and replication are the only operations that trigger changes.

When an object is migrated or replicated, an exception containing the object's identifier and its new locations is established in the leaf node responsible for the object. Index nodes are handled in the same way as nodes in a B-link tree. Thus, excessive migration and replication can result in the usual bottom-up splitting procedure. As in a B-link tree, no locks on the parent node need to be acquired because all the nodes of the same level are chained (including the two nodes that result from a split) so that even clients that read an out-of-date parent node (before the split) can find the correct sibling node (after the split) by following the pointers of the chain. (Details of this approach are described in [LY81, JK93, Lom96].) It should be noted that if a leaf node is split, both resulting nodes inherit the default route of the original node. Accordingly, two leaf nodes can only be coalesced if they have the same default route. With the exception of this constraint to coalesce nodes, all conceivable reorganization techniques are possible in our distributed index; for example, moving index nodes from one server to another.

3.3 Concurrency Control

The big advantage of B-link trees is that they allow to safely split a node without holding a lock on the parent node; that is, searches can be completed by navigating along sibling links even if an out-of-date parent node was read. But, what if an object is migrated and the lookup is based on an out-of-date leaf node? At this level the chaining mechanism does not work so that another solution must be found. To see why, consider the following situation: a client reads a leaf node which indicates that Object 5 is stored by Server L1. Now, Object 5 migrates from Server L1 to Server L9, thereby deleting Server L1's copy of Object 5 and also updating the distributed index. Then, the client visits Server L1 based on its inconsistent index information and erroneously assumes that Object 5 has been deleted. To handle these race conditions we adopt a simple technique: After Object 5 has migrated from Server L1 to Server L9, Server L1 keeps temporarily a forward for Object 5 to Server L9. These forwards are only accessed by lookups that were already past the leaf node when the modification took place. Therefore, these forwards can automatically be discarded after a short period of time.

4 Caching and Replication

We now turn to the question how the performance of our distributed index can be improved by the means of caching and replicating index nodes. Without caching and replication, it is easily possible that several different servers need to be visited at potentially high cost in order to probe the index. Both caching and replication reduce the number of visits to remote servers, and they are particularly effective in this environment because the nodes of our index are very rarely modified. In this section, we will discuss situations in which caching is especially effective, and we will present a simple replication strategy that improves performance in situations in which caching does not help. At the end of the section, we will briefly discuss how copies of index nodes can be kept consistent.

4.1 Caching

To see how caching works, consider that a client of Server L2 wants to read Object 5 in Figure 1. To do so, it will read the root of the index from Server R, then it will read the intermediate node from Server I1 and finally the leaf node from Server L1. The kick of caching is that after these requests have been processed, copies of the three index nodes are available at Server L2 so that a subsequent request of the client to, say, Object 25 can be processed without visiting Servers R and I1. The copies of the index are kept at the client until they are found to be no longer useful for clients of Server L2 so that they are replaced by other, more frequently used index nodes in the cache of Server L2.

This example already demonstrates the two major reasons why caching is particularly effective in our environment: (1) The root and intermediate nodes of higher levels of the index are used very often. These nodes are, therefore, likely to be cached by almost every server so that interaction with the top-level servers of the server hierarchy, which are potential bottlenecks, are rare. (2) When a client accesses

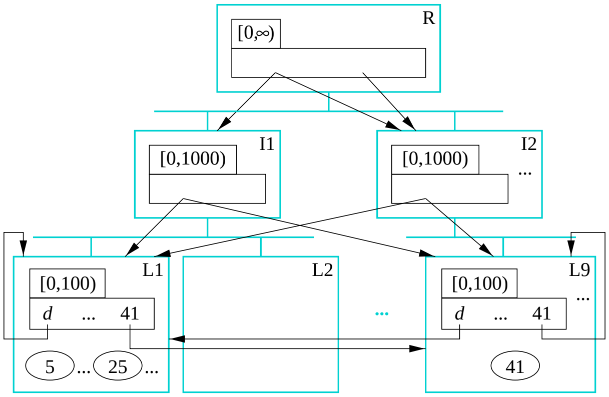

Figure 2: The replication scheme

an object from a remote server, it is likely that the client will also access other, logically related objects from that server. As a result, it is often also beneficial to cache, at least for a while, leaf nodes of the index.

4.2 Replication

Caching is not the solution to all performance problems. Consider again Figure 1. The figure shows that Object 41 has been migrated to Server L9—probably because it is frequently used by Server L9 or clients that are run on servers in Server L9's neighborhood; say, Servers L10, L11, . . . Now if a client of Server L10 tries to read Object 41, it will have to visit (distant) Server L1 at high cost in order to find out that Object 41 is stored at (nearby) Server L9; likewise, clients of Servers L11, L12, . . . would have to visit Server L1 at high cost before fetching Object 41.

In this case, caching does not help because it cannot take advantage of *geographical locality*; this observation has, for example, also been made for WWW pages in [GS94]. Thus, we need to *replicate* the index nodes of migrated and replicated objects as shown in Figure 2. When Object 41 is migrated from Server L1 to Server L9, the corresponding leaf node is replicated at Server L9, the parent of this leaf is replicated at the parent server of Server L9, and so on until a common ancestor of Servers L1 and L9 is reached (in this case Server R, the root). This way, requests of clients at Servers L10, L11,. . . to access Object 41 can be processed at low cost using Server I2's copy of the intermediate node and Server L9's copy of the leaf node and without visiting the distant Server L1.

An interesting case occurs if a replicated index node is split. Consider, for example, that the leaf node [0,100) of Figure 2 needs to be split into two new leaf nodes for [0,50) and [50,100). After this split only a replica of the [0,50) node needs to be stored at Server L9 because only this node contains information about objects that are stored at Server L9. Furthermore it should be noted that there is another important technical difference between caching and replication: As shown in Figure 2, replication actually changes the structure of the index; for example the root of the index was updated as a result of replicating the intermediate node; the root would not have been updated if the intermediate node had been cached at Server I2. As a result, the size of the index grows, and it is even possible that the height of the index increases due to node replication.

4.3 Consistency

Of course, we have to maintain a level of consistency that guarantees that every lookup operation finds the referenced object (if it exists) or determines its non-existence (if it never existed or if it was deleted). Fortunately, because our index is based on a B-link tree, we can use all the techniques devised in the literature for maintaining replicas of the root and intermediate nodes of our index (e.g., [JK93, Lom96]).

We need a different approach for replicas of leaf nodes. We propose to use a *master copy* concept for leaf nodes. That is, all updates to leaf nodes are first performed on the master copies which, of course, are freely distributed over the network (one could, for example, declare the copy of a leaf at its default server as master). From these master copies the updates are asynchronously propagated to the replicas. Now, if a lookup of an object fails this can have two reasons: either the object does not exist or the lookup was navigated to the wrong server due to an out-of-date index node replica. Therefore, every lookup failure has to be verified by visiting the master copy. However, such failures can be expected to be rare because they can only happen if the referenced object doesn't exist (i.e., a referential integrity violation in the database occurred) or if a (copy of an) object was migrated and this migration was not yet propagated to the replicas of the corresponding leaf node.

5 Alternative Search Strategies

In this section, we will complete our discussion of how location-independent identifiers can be implemented by presenting alternative strategies to probe our distributed index in order to dereference a location-independent identifier. We will classify such strategies along two dimensions: Policies of the first dimension specify how the *distributed index* is traversed, and policies of the second dimension specify how the *server hierarchy* is traversed.

5.1 Traversing the Distributed Index: Full Traversals and Shortcuts

The standard way to probe an index is to read the root of the index first, follow the appropriate pointer to an intermediate node, and go further down from node to node until the right leaf is found; we will call this strategy *Full Traversal*. A potential improvement to this strategy is to start at an intermediate node or, if available, even directly at the right leaf of the index; we call such an approach a *Shortcut*. To illustrate the benefits of Shortcuts, let's go back again to Figure 1 and consider a request of a client at Server L2 to access Object 5. Due to a Shortcut, the client can search for Object 5 using immediately the intermediate node stored at Server I1 and without visiting Server R for the root. Shortcuts are possible because every server maintains a local B$^+$-tree for nodes of the distributed index (see Section 3.1), and this local B$^+$-tree makes it possible to probe for relevant index nodes for an object. We define the Shortcut policy as a search strategy that starts an index traversal at the bottom-most of these relevant index nodes; i.e., the index node with the narrowest key range. (Of course, if no relevant index node is found at a server, the server must ask its parent server for help using the server-hierarchy protocol described in Section 2.2.)

The benefits of Shortcuts are not very pronounced in some situations in which the root and high-level index nodes are cached. If, for example, the root is cached and no other node on the path to the right leaf node can be found, the Full Traversal and the Shortcut strategies are identical, since both strategies will use the cached copy of the root as entry point for the index traversal. Using Shortcuts can, however, improve performance if caching is not very effective, say, because the index has many levels and not all high-level nodes of the index can be cached at every server.

5.2 Traversing the Server Hierarchy: Iterative and Recursive Search

For the second dimension of possible search strategies, we propose two approaches which we call *Iterative* and *Recursive* search. Figure 3 gives an example for an Iterative search. A client of Server L1 requests to access an object stored at Server L9. After finding out that neither the object itself nor the root or any other relevant index nodes are available at Server L1, the client asks Server I1, the parent server, for help. Server I1 replies that it has no useful information either. So, the client asks Server R, the grandparent server, for help; Server R returns the root of the index. Then the client visits Server I2 and finally Server L9 in order to fetch the leaf node and the object.

In an Iterative search, a client keeps control of the entire search. In a Recursive search, on the other hand, clients delegate control to other servers—this is illustrated in Figure 4. After finding that no useful information is available at Server L1, the client asks Server I1 for help, just as before. Rather than replying, however, Server I1 forwards the client's request to Server R, its own parent, after finding that it has not got any useful information either. Server R has the root of the index and, therefore, knows that the search should be continued at Server I2; it forwards the request to Server I2, and Server I2 forwards the client's request to Server L9. Finally, Server L9 ships the object to Server L1.

Comparing the Iterative and Recursive approaches in Figures 3 and 4, it can be seen that the recursive approach requires less messages: 5 instead of 10 for the Iterative approach. Even more important, the messages of the Recursive approach are cheaper: in all, the Recursive approach requires 7 hops through the network (four on the way to Server L9, and three on the way back) whereas the Iterative approach requires 16 hops.

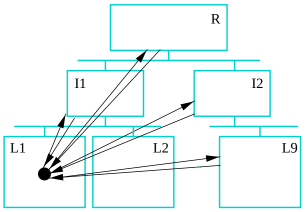

Figure 3: Iterative Search

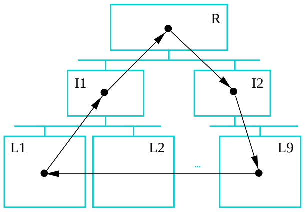

Figure 4: Recursive Search

The *basic* Recursive technique has, however, one crucial disadvantage: it does not support caching of index nodes. After the request has been processed, no index nodes are cached at Sever L1. In order to take advantage of caching, we propose the use of a *cache-enhanced* Recursive strategy in one of the following two ways. (1) While delegating the search from server to server, all index nodes that were used are collected. When the search arrives at the target server (L9 in the example), not only the object is returned but also the collection of all used index nodes. Obviously, this causes rather large messages to be sent over the network. (2) Alternatively, the index nodes could be sent back asynchronously to the server that started the search as soon as the nodes are visited. In the example, Server R would send the root node to Server L1, Server I2 would send the intermediate node to L1, and Server L9 would send the leaf node to L1. The disadvantage of this approach is that it causes additional messages.

5.3 Summary

Combining the two traversal methods for the distributed index and the four traversal methods for the server hierarchy (Iterative and the three choices for Recursive), we get eight conceivable strategies. Each of these strategies can be used with and without node replication, giving us a total of sixteen different ways to implement location-independent identifiers in a distributed database system.

6 Performance Experiments and Results

In this section, we study the tradeoffs of all the different ways to implement location-independent identifiers. As a baseline for our comparisons, we also show the performance of an approach that uses location-dependent identifiers as described in Section 2.1.

6.1 Simulation Model and Parameters

For all our experiments, we used a simulator. The simulator allowed us to study different server hierarchies and network topologies and get reproducible results for wide-area networks. With the simulator, we could generate a

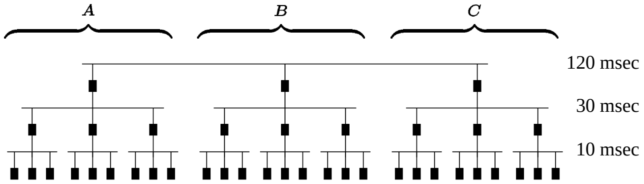

Figure 5: Server Hierarchy and Network Topology

distributed database, maintain the distributed index in the case of location-independent identifiers and handle the forward references in the case of location-dependent identifiers. Each server in the simulation had a single disk (average latency 15 msec, 8K page size) for the local B$^+$-trees and a main-memory buffer pool for caching local and remote objects, nodes of the distributed index and nodes of the local B$^+$-trees. When using location-independent identifiers, the buffer pool was split in two halves of equal size: one half to cache nodes of the distributed index and one half to cache objects of the database.

The server hierarchy and the network topology used in all experiments reported here is shown in Figure 5. In fact, our server hierarchy consists of three separate server hierarchies (A, B, and C in Figure 5), each with its own root, each with three intermediate servers, and each with nine leaf servers. The roots of the three hierarchies are connected by a wide-area network with 120 msecs latency. Within a server hierarchy, intermediate servers and the root are connected by a metro-area network with 30 msecs latency, and leaf nodes of the same subnet and the "gateway" intermediate node of the subnet are connected by a local-area network with 10 msecs latency. (All latency times were determined in separate experiments with *ping* on the Internet.)

6.2 Database and Workloads

As mentioned earlier, we only consider cases in which objects are stored at leaf servers of the server hierarchy. In our experiments, every leaf server stored exactly 10,000 objects, so the whole database had 270,000 objects. Every object was 8K large; this is the page size and also the size of local B$^+$-tree nodes and nodes of our distributed index. We established an initial distributed index for this database as described in Section 3.2 with one exception: the root of the index was stored on all three root servers of the three server hierarchies. We carried out some experiments on this *Initial* database. Then we migrated objects and carried out experiments on the resulting *Migration* database.

The Migration database was generated by migrating all objects from servers in A to servers in B, migrating all objects from B to C, and migrating all objects from C to A. After migration, every leaf server stored again exactly 10,000 objects. As a result of migration, forwards had to be established if location-dependent identifiers were used, and the distributed index for location-independent identifiers had to be updated. Also, index nodes were replicated if node replication was activated as defined in Section 4.2.

	Initial	Migr.
LocDep	21.30	49.24
LocIndep	21.60	49.24
LocIndep + index node repl.	21.60	108.94

Table 1: Space Required for Forwards and Index (in MB)
Space Requirements for Local B$^+$-trees are Included

We also created databases with objects that were *replicated* rather than migrated. We do not show the results of experiments with those databases here due to space constraints; the effects were, however, essentially the same as with the Migration database.

To get a feeling for the storage space consumed by the individual approaches, Table 1 lists the space requirements of forwards (location-dependent identifiers) and for the distributed index with and without node replication (location-independent identifiers); all numbers of Table 1 include the space required to maintain local B$^+$-trees. It becomes apparent that compared to forwards, the additional space used by an entire distributed index is marginal if the nodes of the distributed index are not replicated. Node replication using our schema of Section 4.2 inflated the size of the index by a factor of 2 if objects are migrated or replicated. (In the initial state, no nodes are replicated according to our scheme.)

The workloads we used are very simple. One client was run on every leaf server, and every client requested objects with a fixed frequency of about 20 objects per 10 seconds. The objects read by a client were chosen randomly using a Uniform distribution; we measured workloads in which a client only requested objects stored in its neighborhood (i.e., stored on one of the other 8 servers in the same server hierarchy) or from distant servers (i.e., servers of a different server hierarchy). To reduce the effects of randomness, we repeated all experiments so that the 90% confidence intervals (computed using batch means) were within ±5%.

6.3 Experiment 1: Initial Database

In the first experiment, we study the response time of requests to read objects in the Initial database; i.e., before any objects are migrated. Here, location-dependent identifiers always show the lowest response time because they directly visit the home server of an object and fetch the object from there. We can thus use location-dependent identifiers in this experiment as a baseline to measure the overhead of our techniques to implement location-independent identifiers.

Figure 6 shows the response time of requests to read an object from a nearby server (i.e., clients from A read objects in A). Location-independent identifiers with an Iterative and Full Traversal strategy (Full-Iter) perform only good in the case when the cache of a server is able to hold the root and all relevant intermediate and leaf nodes. This is the case when the cache of every server has 40 or more pages. Shortcuts (Shortcut-Iter) show better performance than Full

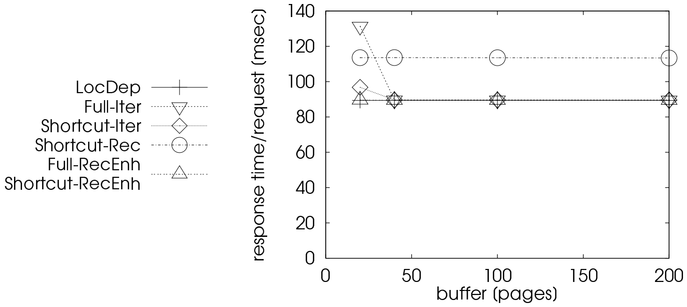

Figure 6: Resp. Time, Vary Cache
Initial DB: Access to Nearby Servers

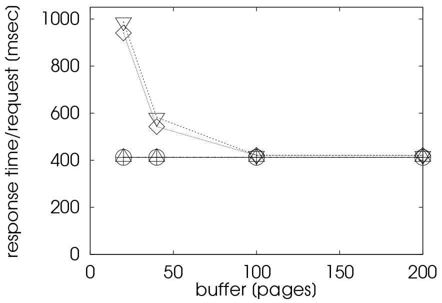

Figure 7: Resp. Time, Vary Cache
Initial DB: Access to Distant Servers

Traversals if the cache is small because Shortcuts only require the right leaf nodes rather than all relevant intermediate nodes to be cached in order to get good performance.

If location-independent identifiers are used with the basic Recursive strategy, no index nodes can be cached and it is always necessary to visit an intermediate server and in some cases the root server. If Shortcuts are used, the response time of basic Recursive searching is 26% higher than that of LocDep (the Shortcut-Rec graph in Figure 6), and if Full Traversals are used, the performance penalty is 46%. (We do not show the Full-Rec graph in Figure 6 to improve readability; the graph would be higher up and parallel to the Shortcut-Rec graph at about 130 msecs.)

The cache-enhanced Recursive strategies get the best of caching and of cheap Recursive traversals. In this experiment, these strategies, therefore, show almost the same (best) performance as LocDep regardless of whether Full Traversals or Shortcuts are used and regardless of the technique used to effect cache enhancement. (As a representative, we use "asynchronous messages" in all experiments reported here.)

Figure 7 shows the response time of requests to read an object from a distant server (i.e., clients in A read objects from B and C). Obviously, the cost to read an object from a distant server is much higher than from a nearby server. Comparing the results of Figures 6 and 7, we can observe the following differences. (1) The Iterative approaches need more cache than in Figure 6, since there are more distant objects than neighborhood objects and, thus, more relevant index nodes need to be cached. (2) All Recursive strategies show the same performance as LocDep because all requests to remote servers need to be routed over the root of the server hierarchy regardless of which strategy is used. As a result, Shortcut-Rec (and Full-Rec) can find an entry point to the index (i.e., the root) at no additional cost.

6.4 Experiment 2: Migration Database

In the second set of experiments, we study the effects of object migration. We will examine two different scenarios: (1) a client reads objects that were migrated from a distant to a nearby server (e.g., a client from A reads objects that were migrated from C to A), and (2) a client reads objects that were migrated from distant servers to other distant servers (e.g., a client from A reads objects that were migrated from B to C). There is a third conceivable scenario: clients read objects that were migrated from a nearby to a distant server. We will not show the results of this scenario for space limitations; in this third scenario, all techniques for location-independent identifiers outperform LocDep, but they never do so by more than 25%. Furthermore, we will not discuss the results for Full Traversals because Shortcuts always perform at least as good.

Figure 8 shows the results for the first scenario. We expect this scenario to be very important for applications in which objects are processed by different clients at different points in their life-cycle. In this experiment, LocDep is outperformed by any of the strategies for location-independent identifiers if the cache of a server has more than 40 pages because it must always visit the distant home server of an object before it can fetch the object from the neighborhood.

As in the first experiment, the performance of location-independent identifiers with an Iterative strategy depends on the size of the cache. If the cache is small, Iterative shows poor performance for repeatedly visiting distant servers, but if the cache is large, Iterative can often completely avoid visiting distant servers resulting in significantly better performance than LocDep. The basic Recursive strategy must always visit a distant server in order to read the right leaf node; therefore, its performance is overall poor in this experiment. The cache-enhanced Recursive strategies again get the best of caching and Recursive searching, and thus, show better performance than basic Recursive and Iterative in all cases. Best performance in this scenario can, however, only be achieved if our simple node replication strategy is in effect: in this case, distant servers never need to be visited because all relevant index nodes are available locally or on nearby servers. This is true regardless of which search strategy is used, so that all three search strategies have almost the same performance if index nodes are replicated.

Figure 9 shows the results of the experiments in the second scenario in which all clients read objects that were migrated from distant servers to other distant servers. LocDep has, again, the overall highest response time. LocDep first

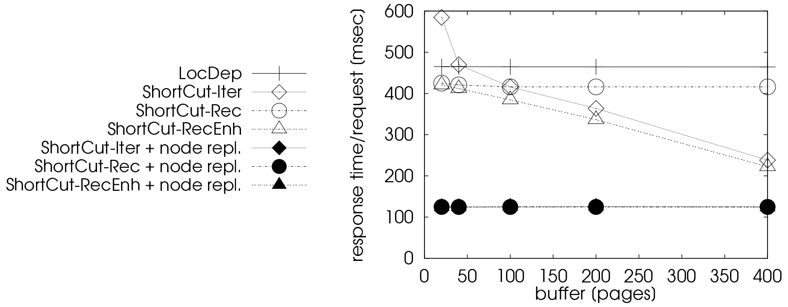

Figure 8: Resp. Time, Vary Cache
Migration DB: Distant to Nearby

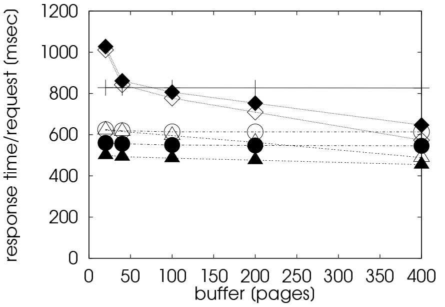

Figure 9: Resp. Time, Vary Cache
Migration DB: Distant to Distant

visits the distant home server of the object, and afterwards LocDep visits the distant server that actually stores the object. In all, LocDep pays the price for four expensive messages across the WAN.

Location-independent identifiers with an Iterative strategy are, again, sensitive to the amount of buffers available at the servers. Interestingly, node-replication is detrimental to the performance of the Iterative strategies in this experiment because (1) node replication does not help in this case since reading an index node is always expensive in an Iterative search if the node is not cached, regardless of whether the original or a replica of the node is read; and (2) node replication significantly inflates the size of the distributed index (Table 1), and therefore, caching of index nodes becomes less effective.

Using location-independent identifiers with a Recursive search strategy (basic or cache-enhanced), only three expensive messages need to be sent over the WAN: A client in A would, for example, first search the right leaf node in B. Then, the server in B would forward the request to the server in C that stores the object. Finally, the server in C would return the object to the client in A. With a Recursive search strategy, node replication can sometimes be helpful because with node replication the object in C could be read without visiting any servers in B; thus, avoiding one hop through the WAN. It must be noted, however, that there is an indeterminism in the Recursive strategies which makes node replication less effective in this case. When inspecting the root node of the index, the search could be continued using the original copy of the intermediate node stored in B or using the replica in C: the root node contains pointers to both and, at that point, it is not possible to see which one is the better to use. If the original copy is used, no advantage of node replication is taken, and three hops through the WAN are required to access an object that was migrated to C. If the replica is used only two hops through the WAN would be required to access an object in C; however, three hops would be required this way if the target of the search is an object that has not been migrated (i.e., is still stored in B). For our experiments, we implemented this indeterminism by using the original copy and the replica in 50% of the searches each.

7 Related Work

Our work has been influenced by various developments on index structures for distributed databases, wireless communication systems and name services in operating systems.

Data structures and algorithms for implementing distributed hash tables and trees are described in [JK93, KW94, LNS94, Lom96]. Issues such as concurrency control and recovery are thoroughly covered there, and the results are directly applicable to our approach. The main focus of these papers is to balance the load of a system by distributing and replicating parts of the indexes on several servers. The issue of finding neighboring copies of replicated data and the idea of finding shortcuts in a distributed index, however, have not yet been discussed in that work.

In wireless communication systems, profiles of mobile users need to be retrieved with as little communication cost as possible. [AHMW94, JLS$^+$95] discuss location-dependent as well as location-independent schemes for maintaining identifiers (phone numbers) for this task. Pleiades [JLS$^+$95], for example, uses a hierarchy of servers where the profile information is stored in the leaves and higher level servers store routing information for every profile in all servers lower in the hierarchy. Thus, creating new objects is much more expensive than in our approach, because in Pleiades all servers up to the root must be informed. In addition, Pleiades does not support the caching of indexing information.

In operating systems, name services [Moc87, CCI89, Ter85] bind hierarchically organized names to hosts, mailboxes, etc. These names can be viewed as location-independent identifiers by which the objects are located. In contrast to our work, the name space is usually organizationally chosen (for example, the ".com" domain in DNS) and no index structures are involved (for example, in DNS the database is stored in flat text files maintained by administrators). [Ord93] contains a proposal to improve such name services using "flat" tables.

There are several distributed database systems that employ location-dependent identifiers. SHORE [CDF$^+$94] is a research prototype and Itasca a commercial system [Ita93] using a location-dependent naming scheme. In Itasca, if the birth site of an object is found to be unavailable, a broadcast message to all servers is sent which is not viable in large distributed systems. In the Thor project [DLMM93], lazy updating of location-dependent identifiers was discussed. The basic idea is to lazily change all references in the whole database that are pointing to old locations of a migrated object so that they contain the current location of the object. Objects can potentially leave several forwards behind and it may take multiple steps to resolve chains of forwards. Patankar et al. [PSS96] propose a directory service for CIM databases; their approach is based on hashing, and it works best in a LAN.

8 Conclusion

In this paper, we studied two alternative ways to identify objects in a distributed database: location-dependent and location-independent identifiers. Clearly, location-dependent identifiers show the best performance if objects are neither replicated nor migrated. However, in many distributed database applications there are (1) hot-spot objects which are frequently accessed by many different sites, and (2) there are objects whose access patterns shift in different phases of their life time. The solutions to these problems are (1) object replication and (2) object migration. In this paper we showed that only systems employing location-independent identifiers can fully profit from object replication and migration and that even in the absence of replication and migration, the overhead of using location-independent identifiers is very low in most situations. Purely local objects (i.e., objects that never move and are accessed only from their “birth site”) incur no additional cost in our approach because special (global) actions are only taken for objects that are migrated or replicated. As a result, we recommend to use location-independent identifiers.

We presented a number of different techniques to implement location-independent identifiers using a general-purpose distributed index with slight modifications to the structure of the nodes. The overall design choices were:

Index node replication vs. no node replication: Index node replication ensures good performance in one particular, very important case: copies of objects that were created by a distant server are available in the neighborhood. Of course, node replication does not come for free because additional storage space is required and replicated nodes must be kept consistent. In our particular environment, however, this additional cost is moderate.

Full Traversal vs. Shortcuts: Using Shortcuts is a no-loss game. Shortcuts are never worse than Full Traversal, but in many situations they do not provide significant performance benefits either.

Recursive vs. Iterative search: The clear winner in our experiments was cache-enhanced Recursive searching.

The messages required for returning index nodes in the cache-enhanced Recursive strategy cause costs (in terms of bandwidth usage) that cannot be realistically measured in a simulation. Therefore, we are currently working on a “real” implementation of our approaches in order to carry out experiments using the Internet and LANs. Based on this implementation, we are also going to specifically study the cost to migrate and replicate objects.

References

[AHMW94] V. Anantharam et al. Optimization of a database hierarchy for mobility tracking in a personal communications network. *Performance Evaluation*, 20, 1994.

[CCI89] CCITT. *Verzeichnis-Systeme*, Volume 8 of *CCITT-Empfehlungen der V-Serie und der X-Serie*. R. v. Decker's Verlag, Heidelberg, 1989.

[CDF+94] M. J. Carey et. al. Shoring up persistent applications. In *Proc. of the ACM SIGMOD Conf. on Management of Data* Minneapolis, 1994.

[DLMM93] M. Day et. al. References to remote mobile objects in Thor. *ACM Letters on Programming Languages and Systems*, 2(1–4), 1993.

[EGK95] A. Eickler, C. Gerlhof, and D. Kossmann. A performance evaluation of OID mapping techniques. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Zürich, 1995.

[GS94] J. Gwertzman and M. Seltzer. The case for geographical push-caching. Technical Report HU TR-34-94, Harvard University, Cambridge, 1994.

[Ita93] Itasca Systems Inc. Technical summary for release 2.2, 1993. 7850 Metro Drive, Mineapolis, MN 55425, USA.

[JK93] T. Johnson and P. Krishna. Lazy updates for distributed search structures. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, Washington, 1993.

[JLS+95] J. Jannink et al. Data management for user profiles in wireless communication systems. Technical report, Stanford University, Computer Science Dept., 1995.

[KW94] B. Kröll and P. Widmayer. Distributing a search tree among a growing number of processors. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, Minneapolis, 1994.

[LNS94] W. Litwin et al. RP*: A family of order preserving scalable distributed data structures. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Santiago, 1994.

[Lom96] D. Lomet. Replicated indexes for distributed data. In *Proc. of the Intl. IEEE Conf. on Parallel and Distributed Information Systems*, Miami Beach, 1996.

[LY81] P. Lehman and S. Yao. Efficient locking for concurrent operations on B-trees. *ACM Trans. Database Sys.*, 6(4), 1981.

[Moc87] P. Mockapetris. Domain names - concepts and facilities. RFC 1034, USC/Information Sciences Institute, 1987.

[Ord93] J. J. Ordille. *Descriptive Name Services for Large Internets*. PhD thesis, University of Wisconsin, Madison, 1993.

[PSS96] A. K. Patankar et al. A directory service for a federation of cim databases with migrating objects. In *Proc. IEEE Conf. on Data Engineering*, New Orleans, 1996.

[Ter85] D. B. Terry. *Distributed Name Servers: Naming and Caching in Large Distributed Computing Environments*. Technical report CSD-85-228, University of California, Berkeley, 1985.