Performance Enhancement Through Replication in an Object-Oriented DBMS

by
Eugene J. Shekita
Michael J. Carey

Computer Sciences Technical Report #817
January 1989

Performance Enhancement Through Replication in an Object-Oriented DBMS

Eugene J. Shekita
Michael J. Carey

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

This research was partially supported by the Defense Advanced Research Projects Agency under contract N00014-85-K-0788, by the National Science Foundation under grant IRI-8657323, by DEC through its Incentives for Excellence program, and by donations from Apple Corporation, GTE Laboratories, the Microelectronics and Computer Technology Corporation (MCC), and Texas Instruments.

Performance Enhancement Through Replication in an Object-Oriented DBMS

Eugene J. Shekita
Michael J. Carey

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

ABSTRACT

In this paper we describe how replicated data can be used to speed-up query processing in an object-oriented database system. The general idea is to use replicated data to eliminate some of the functional joins that would otherwise be required for query processing. We refer to our technique for replicating data as *field replication* because it allows individual data fields to be selectively replicated. In the paper we describe how field replication is specified at the data model level and we present storage-level mechanisms to efficiently support it. We also present an analytical cost model to give some feel for how beneficial field replication can be and the circumstances under which it breaks down. While field replication is a relatively simple notion, the analysis shows that it can provide significant performance gains in many situations. Much of the discussion is based on the EXTRA data model, but the ideas presented here can also be extended to other data models that support reference attributes or referential integrity facilities.

1. INTRODUCTION

In recent years, a number of new data models have been proposed. Naturally, there is no agreement on exactly what the "right" data model is, but by now it is clear that there are many applications for which the relational model is inadequate, at least in its pure form. Recent data models have addressed this issue by providing new constructs that offer more modeling power. Included among the new constructs are support for reference attributes (or object-valued attributes) [Ship81, Zani83, Cope84, Bane87, Fish87, Khos87, Care88a], type inheritance [Ship81, Cope84, Bane87, Fish87, Rowe87, Care88a], abstract data types [Ston86a, Schw86, Care88a], procedural fields [Ston86c, Ston87], complex objects [Sche86, Schw86, Horn87, Kim87, Care88a], set-valued attributes [Ship81, Zani83, Cope84, Rowe87, Care88a], and so forth. A good survey of many of the above constructs is presented in [Hull87].

One area of research that is likely to be promising is exploring ways in which these new modeling constructs can be used to enhance database performance. There has already been a fair amount of research in this area. For example, the results of executing procedural fields can be cached to speed up future accesses [Rowe87, Ston87]; information about complex objects can be used to make clustering decisions [Kim87, Horn87]; reference attributes can be used to replace costly value-based joins with less-expensive functional joins [Ship81, Zani83, Cope84, Bane87, Fish87, Care88a], and so on. Many other examples could be cited as well.

In this paper we describe how replicated data can be used to speed-up query processing in an object-oriented

DBMS.[1] In particular, we describe how reference attributes can be used to specify data replication. The motivation for replicating data in this case is to speed-up query processing. Data values that would normally be accessed through a functional join are replicated so the join can be eliminated or at least made less costly. We refer to our technique for replicating data as *field replication* because it allows individual data fields to be selectively replicated. In the paper we describe how field replication is specified at the data model level and we also present storage-level mechanisms to efficiently support it.

The remainder of this paper describes field replication in detail. Although much of the discussion is based on the EXTRA data model of the EXODUS project [Care88b], the ideas presented here can be extended to other data models that support reference attributes or referential integrity facilities of the type discussed in [Date87]. The rest of the paper is organized as follows: In Section 2 we present a simple employee database that is used in examples throughout the paper. Section 3 provides an introduction to replication in EXTRA, and describes several examples where replication is useful. In Section 4, we describe the first of our two basic replication strategies, in-place replication. In Section 5, we describe the second of our two basic replication strategies, separate replication. Section 6 presents an analytical cost model in which no replication, in-place replication, and separate replication are compared. In Section 7 we discuss related work, and in Section 8 our conclusions are presented. The reader may have noticed that the section on related work is presented towards the end of the paper rather than at the beginning; this is done so that field replication can be compared with related work.

2. AN EMPLOYEE DATABASE EXAMPLE

2.1. The Example

In order to make the discussion that follows concrete, we will often refer to the employee database pictured in Figure 1. As shown, the database is modeling the organization of a company. Each *organization* is made up of one or more *departments*, and each department is made up of one or more *employees*. The database is obviously too simple to be considered realistic, but it will serve the purposes of this paper. The schema of our database has been described in the syntax of the EXTRA data model [Care88a]. We will not assume that the reader is intimately familiar with the EXTRA data model. We will assume, however, that the reader is familiar with the notion of reference attributes [Zani83].

In the schema, three types have been defined: the type ORG, which defines the structure of organization objects, the type DEPT, which defines the structure of department objects, and the type EMP, which defines the structure of employee objects. Note that ORG, DEPT, and EMP have been capitalized to distinguish them as type definitions. Using the type definitions, four sets have been created. The set Org, the set Dept, and the sets Emp1 and Emp2. Two sets of employees have been included for reasons that will become clear later. In the EXTRA data model, the notation **own ref** indicates ownership or an existence dependency. Therefore, if Emp1 is deleted, all the EMP objects in Emp1 are also deleted. Note, however, that any DEPT objects which are referenced by Emp1 via the **ref** attribute 'dept' are not deleted when Emp1 is deleted.

[1] We use the term "object-oriented DBMS" here in reference to the kind of DBMS that [Ditt86] calls *structurally object-oriented.*

```
define type ORG          define type DEPT           define type EMP
(                        (                          (
    name:     char[],        name:     char[],          name:     char[],
    budget:   int            budget:   int,             age:      int,
)                            org:      ref ORG          salary:   int,
                         )                              dept:     ref DEPT
                                                    )

                    create Org:    {own ref ORG}
                    create Dept:   {own ref DEPT}
                    create Empl:   {own ref EMP}
                    create Emp2:   {own ref EMP}
```

Figure 1: The Employee Database

2.2. The Physical Representation of Sets and Objects

Before continuing, it is important to clearly state our assumptions about the way in which sets and objects are represented on disk. Our assumptions are relatively straightforward and coincide with what is being considered in the EXODUS project [Care88b]. First, we will assume that top-level (or named) sets are stored as disk files. Second, we will assume that objects with a simple, unnested structure are stored as single, contiguous objects on disk. Third, we will assume that every object contains a *type-tag*, which identifies the object's type. Finally, we will assume that object identifiers (OIDs) are used to implement reference attributes. In our employee database, for example, the set Empl would be stored as a disk file, and the pages in that disk file would contain only the EMP objects belonging to Empl. Each object E in Empl would have its type-tag set to EMP and would be stored as a single, contiguous object. The values of E's attributes would be stored in E itself (not as separate objects) and the reference attribute E.dept would contain the OID of a DEPT object.

Note that our assumptions do not say anything about the physical representation of more complicated objects, such as objects with nested sets. The physical representation of such objects is likely to be less straightforward. In this paper, we sidestep these issues by considering only objects that have a simple physical representation.

3. AN INTRODUCTION TO FIELD REPLICATION

3.1. A Simple Example

Although functional joins are generally much cheaper than value-based joins, they can still be expensive to execute. For example, if E is an EMP object, then dereferencing E.dept.name will generally cause two I/Os, one to retrieve E and another to retrieve the DEPT object that is referenced by E. With field replication, the second I/O can often be eliminated. As its name implies, this is accomplished by replicating data. If it is known that a particular reference path will be frequently accessed, then the data at the end of that path is replicated so that a separate I/O does not have to be performed to retrieve it. For example, we could add the following item to our schema:

```
replicate Emp1.dept.name
```

Here, replication is being specified along the reference path Emp1.dept.name. The replicate statement specifies that the values for dept.name will be replicated in objects belonging to Emp1. In other words, objects in Emp1 can be thought of as having a "hidden" field in which a replicated value for dept.name is stored[2]. By hidden, we mean that the replicated value will not be visible to users at the query language level for either updates or retrievals. Query processing, on the other hand, *will* know about field replication and exploit it whenever possible to avoid functional joins. For example, consider the following query, which retrieves the name, salary, and department of each employee who makes more than $100,000:

```
retrieve (Emp1.name, Emp1.salary, Emp1.dept.name)
where Emp1.salary > 100000
```

With Emp1.dept.name replicated as above, the query can be executed without performing a functional join.

Clearly, field replication will help speed up queries that would otherwise require a functional join. The only question is at what cost. First, there is the extra disk space that is required to store replicated values, and second, there is the cost of propagating updates to replicated values. As far as disk space goes, we assume that the speed-up in query processing is considered worth the extra space. With the cost of disk space decreasing, and with the gap between disk speed and memory speed growing all the time, this is probably a reasonable assumption to make. Propagating updates presents more of a problem. If there are a large number of replicated values, and updates to replicated values are frequent, then obviously, field replication will not work. Propagating updates would be so time consuming that query processing as a whole would slow down. Our assumption here is that the DBA who defines the data model is knowledgeable enough to realize that replication should only be specified on reference paths that are frequently accessed and, at the same time, infrequently updated. Under these circumstances, field replication will generally be beneficial. Later on, an analytical cost model will be developed to give some feel for the circumstances under which field replication is beneficial and when it breaks down.

3.2. Field Replication is Associated with Instance

One thing to notice about the example given earlier (where Emp1.dept.name was replicated) is that field replication is associated with instance (the set Emp1) rather than type (the type EMP). In general, we feel that associating field replication with instance provides more modeling power. (Note that this is only an issue in data models that separate type definition from type instantiation.) For example, it allows us to replicate Emp1.dept.name and, at the same time, *not* replicate Emp2.dept.name. This would be impossible if replication was associated with type. Associating field replication with instance rather than type also allows us to avoid certain issues related to type inheritance, such as whether field replication should be an inherited property. In this paper we will assume that field replication is associated *only* with instance and *never* with type.

[2]This is slightly misleading because later on we will describe a replication strategy in which values for dept.name are stored in separate objects. For now, however, it is easiest to think of objects in the Emp1 as having a hidden dept.name field.

3.3. More Examples of Field Replication

3.3.1. Full Object Replication

In addition to allowing individual fields to be selectively replicated, field replication can also be used to specify full object replication. For example, we could define a replication path on Empl.dept.all. This would cause all the information about an employee's department to be replicated and would allow any information about an employee's department to be obtained without a functional join.

3.3.2. Field Replication on N-Level Reference Paths

Until now, all our examples have involved replication on *1-level paths*; that is, reference paths that require only one functional join. One of the important uses of field replication is in reference paths of two or more levels because it allows more than one functional join to be eliminated. For example, we could define a replication path on Empl.dept.org.name, which is an example of *2-level replication* because it specifies replication on a 2-level path.

3.3.3. Using Field Replication to Collapse N-Level Paths

Another use of field replication is in collapsing n-level paths to n−1 levels or less. For example, we could define a replication path on Empl.dept.org. This effectively allows any information about an employee's organization to be obtained with just one functional join rather than two. The 2-level path from objects in Empl to objects in Org has been effectively collapsed into a 1-level path.

Of course, the same thing can be accomplished without replication. by adding an 'org' field to the type definition for EMP, but this could lead to problems with referential integrity. For example, if the 'org' field of a DEPT object D is changed from X to Y, then all the EMP objects that reference D would have to have their 'org' field explicitly changed from X to Y. In contrast, with replication, the required changes would take place automatically, and referential integrity could never be violated.

3.3.4. Indexing on an N-Level Path

Our final example shows how field replication can be used in indexing. There is basically no reason why an index cannot be built on replicated data, and by allowing indexes to be built on replicated data, new indexing opportunities are created. For example, suppose we had:

> **replicate** Empl.dept.org.name
> **build btree on** Empl.dept.org.name

Because of replication, an index for the path Empl.dept.org.name can be built on the replicated values that are stored in Empl. The index would map organization names *directly* to objects in Empl, and could efficiently support queries that require an associative lookup on the path Empl.dept.org.name. Other techniques [Maie86a] have been described for maintaining indexes on paths such as Empl.dept.org.name, but they are not likely to be as efficient (at least for lookups). For example, an associative lookup on Empl.dept.org.name using the techniques described in [Maie86a] would involve traversing three B^+ tree indexes. We will say more about the techniques described in [Maie86a] in the section on related work.

4. THE IN-PLACE REPLICATION STRATEGY

This section describes the first of our two basic field replication strategies, namely, *in-place replication*. In-place replication is analogous to the *cache-in-tuple* strategy of POSTGRES [Ston87, Jhin88], which is described in the section on related work. In-place replication gets its name from the fact that replicated values are stored directly in the objects that cause replication to take place. For example, if we defined a replication path on Empl.dept.name, then an extra field would be added to the objects in Empl for storing the replicated value dept.name. Of course, this means that structural changes will have to be made to the objects in Empl, but such changes are easily handled through subtyping. For the remainder of the paper, it can be assumed that all the structural changes that are required by replication are handled through subtyping.

4.1. Propagating Updates in In-Place Replication

Replicated values are kept consistent using what we refer to as an *inverted path*[3]. To demonstrate how inverted paths are used, suppose a replication path was defined on Empl.dept.name. In order to keep replicated values consistent, we create the inverted path Empl.dept^{-1}, which maps DEPT objects to the objects in Empl that reference them. If the 'name' field in a DEPT object D is updated, the inverted path Empl.dept^{-1} is traversed to propagate that update to the objects in Empl that reference D. Note that the inverted path for Empl.dept.name is Empl.dept^{-1} rather than $\text{Empl.dept.name}^{-1}$. This reflects the fact that in our physical representation of objects, the 'name' field of a DEPT object is stored in the object itself, not as a separate object. The 'name' field does not play a part in the mapping of DEPT objects to Empl objects, so it is dropped in the inverted path. It does play a part in determining what updates need to be propagated, but that issue will be addressed later.

Figure 2 illustrates what the inverted path Empl.dept^{-1} looks like. Basically, inverted paths are broken down in to a series of *links*. For example, the inverted path $P_1.P_2.P_3 \cdots P_n^{-1}$ would be broken down into the links $P_1.P_2^{-1}$, $P_2.P_3^{-1}$, $P_3.P_4^{-1}$, etc. *Link objects*, which implement an inverse mapping, are then created for each link in an inverted path. Strung together end-to-end, these link objects form the inverted path. In our example, the link objects for the Dept set map each DEPT object D to the objects in Empl that reference D. Collectively, the link objects for Dept implement the link Empl.dept^{-1}.

As Figure 2 suggests, each link object contains little more than a collection of OIDs. The OIDs that appear in a link object are kept in sorted order so that, if necessary, a particular OID can be found and deleted using a binary search. Keeping OIDs in sorted order also allows us to propagate updates in clustered order if OIDs are physically based, as they are in EXODUS.

Before continuing, several comments about Figure 2 are in order. First, it is important to notice that the link objects for Dept have been stored in a separate set. In general, each link object can contain a large number of OIDs,

[3] The authors of [Maie86a] also examined how to implement inverted paths, although they did not refer to them as such. As will be noted in the section on related work, our implementation of an inverted path is quite different from the one described in [Maie86a].

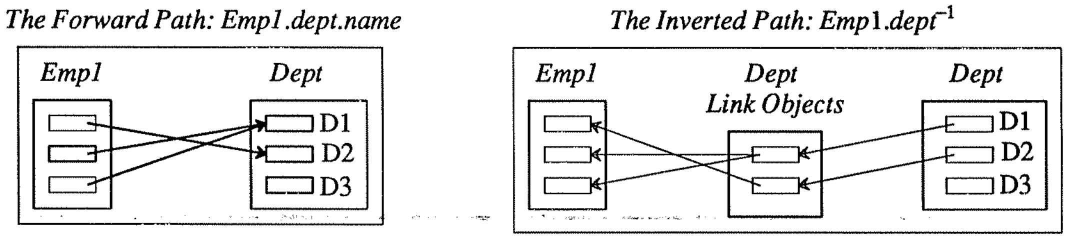

The Forward Path: Empl.dept.name
Empl
Dept
D1
D2
D3
The Inverted Path: Empl.dept^{-1}
Empl
Dept Link Objects
Dept
D1
D2
D3

Figure 2: The Inverted Path Empl.dept^{-1}

and can be quite large as a result. Consequently, the link objects are stored in a separate set so that the clustering of objects in Dept is not disrupted. Second, notice that the link objects for Dept are stored in the same physical order as the objects in Dept which reference them. This is important because in propagating updates, we want to access link objects in clustered order to ensure that as little I/O as possible is generated. Third, notice that only two objects in Dept have link objects, namely, D1 and D2. This is due to the fact that only D1 and D2 are actually referenced by Empl. If D3.name is updated, that updated does not have to be propagated. The way we determine when an update needs to be propagated will be discussed shortly. Finally, it should be noted that each object D in Dept can lie on more than one replication path (although this is not shown in Figure 2). If that were the case, then more than one link object would be generated for D. The way multiple paths are handled will also be discussed shortly.

4.1.1. Maintenance of a 1-Level Path

Once it is created, maintenance of an inverted path is straightforward. Continuing our example, let E be an object in Empl and let E.dept = D. The operations that affect the inverted path are: insert E (inserting E into Empl), delete E (deleting E from Empl), and update E.dept (updating the reference attribute E.dept). The following paragraphs summarize the modifications to the inverted path that take place as a result of these operations:

insert E:	A link object is created for D if there is none. E's OID is then added to D's link object, after which E.dept.name is retrieved and stored in E.
delete E:	E's OID is deleted from D's link object. If there are no longer any OIDs in the link object, it is deleted.
update E.dept:	The actions under *delete E* are executed with D set to the old value of E.dept, and then the actions under *insert E* are executed with D set to the new value of E.dept.

Note that deleting D does not affect the inverted path. The assumption here is that D can be deleted only when it is not referenced by any object in Empl. This implies that D cannot be part of the inverted path when it is deleted, and therefore its deletion cannot affect the inverted path.

4.1.2. Handling N-Level Replication Paths

Replication paths with two or more levels are handled in much the same way that 1-level paths are handled. It is mostly a matter of adding more links to the inverted path. For example, suppose a 2-level replication path was defined on Empl.dept.org.name. Figure 3 illustrates what the inverted path Empl.dept.org^{-1} looks like. As shown, the inverted path has been broken down into two links, Empl.dept^{-1} and dept.org^{-1}. The link objects for Dept implement Empl.dept^{-1}, while the link objects for Org implement dept.org^{-1}. From the figure, it should be clear how updates are propagated. If the 'name' field of an ORG object O is updated, then dept.org^{-1} and Empl.dept^{-1} are traversed to propagate that update to the objects in Empl that reference O.

Maintenance of an inverted path with n levels is mostly just an extension of maintenance on a 1-level inverted path. The main difference is that the effects of insertions and deletions can ripple through n levels of the inverted path rather than just one. For example, suppose that we have the replication path defined above and that E is an object in Empl such that E.dept = D, and D.org = O. If E is inserted into Empl, then a link object object may have to be created for not just D, but O, too. And if E is deleted from Empl, then both D's link object and O's link object may end up being deleted if they become empty. The only other real difference is that updates to reference attributes of intermediate objects such as D can cause more than one update to be propagated. For example, if D.org is changed from O to X, then X.name will have to replace O.name in all of the objects in Empl that reference D.

As the above paragraphs demonstrate, maintenance of an inverted path can be costly. In general, though, replication will be used on only on static reference paths; that is, paths in which object-to-object references are fairly static. Consequently, the cost of maintaining an inverted path consists primarily of the one-time cost to build it.

4.1.3. Determining How and When to Propagate an Update

To determine how and when to propagate an update, we store something called *link identifiers* (link IDs) in each object O on a replication path. The link ID(s) stored in O identify the link(s) of the replication path to which O belongs. As such, they can be used to determine which updates to O need to be propagated and also how to propagate those updates. The association between link IDs and links is obtained from *link sequences*, which are generated by the database. A link sequence is essentially just a sequence of link IDs that identify the links in a replication path, as illustrated by the following example:

replicate Empl.dept.org.name   link sequence = (1,2)

As shown, the replication path Empl.dept.org.name has been assigned the link sequence (1,2). (Note that the syntax

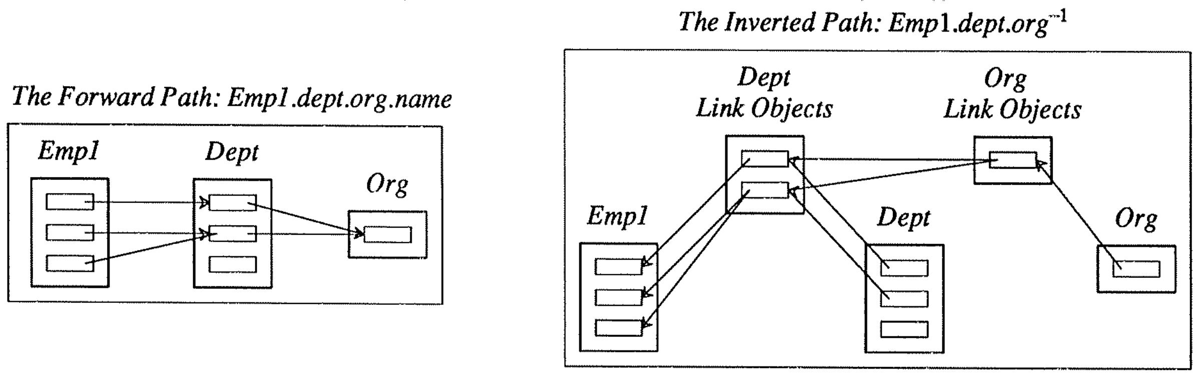

The Forward Path: Empl.dept.org.name
Empl
Dept
Org
The Inverted Path: Empl.dept.org^{-1}
Dept Link Objects
Org Link Objects
Empl
Dept
Org

Figure 3: The Inverted Path Empl.dept.org^{-1}

'link sequence (1,2)' would not appear in the schema definition and has only been shown here for illustrative purposes.) In this example, there are two links in the replication path, and consequently there are two link IDs in the path's link sequence. Link ID 1 corresponds to the link Empl.dept, and link ID 2 corresponds to the link dept.org. The association between link IDs, links, and replication paths would presumably be stored in the system catalog.

Figure 4 illustrates how link IDs would be used in this example. (Note that link-OID denotes the OID of a link object.) The fact that link ID 2 appears in O indicates that O belongs to the replication path(s) that contain the digit 2 in their link sequence, which in this case is the replication path Empl.dept.org.name. Link ID 2 also indicates that O lies at the end of the link associated with link ID 2, namely, dept.org of Empl.dept.org.name. Based on this information, we can tell that updates to O.name need to be propagated and also how to propagate those updates. Similarly, the presence of link ID 1 in D tells us that that updates to D.org need to be propagated.

In the above example, it is important to recognize that simple bit-masks identifying the fields whose updates need to be propagated cannot be used in place of link IDs. For updates to reference attributes such as D.org, more information is needed than a bit-mask can provide, and that is why link IDs are used. For example, if D.org is changed, then in order to propagate updates correctly and to carry out the necessary changes to the inverted path Empl.dept^{-1}, we need to know that D appears in the replication path Empl.dept.org.name and also that D lies at the end of the first link in that replication path. The link ID stored in D provides this information, whereas a simple bit-mask would not provide enough information.

4.1.4. Handling Multiple Paths

To handle multiple replication paths we need to consider two cases. The first case we need to consider is when replication paths share a common prefix. For example, suppose we had:

replicate Empl.dept.budget	link sequence = (1)
replicate Empl.dept.name	link sequence = (1)
replicate Empl.dept.org.name	link sequence = (1,2)

In this example, the fact that each path emanates from Empl means that the mapping defined by Empl.dept (and Empl.dept^{-1}) is the same in each path. As a result, link ID 1 appears in all three link sequences, indicating that not only is the first link of each path the same, but also that each path can share the link Empl.dept^{-1}. In terms of updates, the

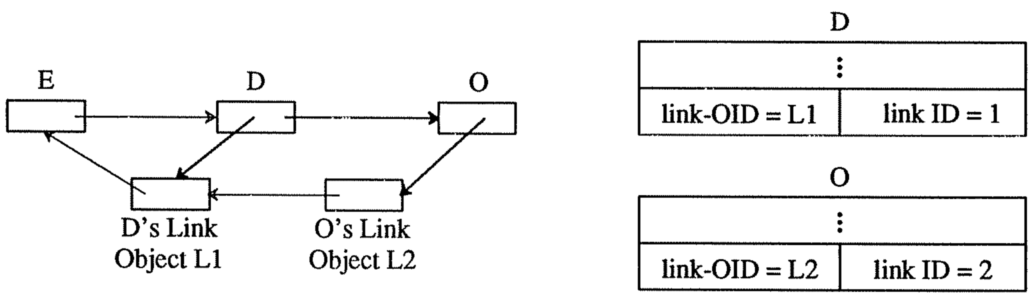


Figure 4: Determining when to Propagate an Update

presence of link ID 1 in a DEPT object D indicates that D lies on all three replication paths, and therefore if either D.budget, D.name, or D.org is updated, that update has to be propagated. In general, if two replication paths A and B share a common prefix $P_1.P_2.P_3 \cdots P_n$, then A and B would share the links $P_1.P_2^{-1}$, $P_2.P_3^{-1}$,..., and $P_{n-1}.P_n^{-1}$ in their inverted paths, and link IDs would be assigned in a manner that reflects this sharing.

The other case we need to consider is when replication paths do not share a common prefix. For example, suppose we have:

replicate Emp1.dept.budget	link sequence = (1)
replicate Emp1.dept.name	link sequence = (1)
replicate Emp1.dept.org.name	link sequence = (1,2)
replicate Emp2.dept.org	link sequence = (3)

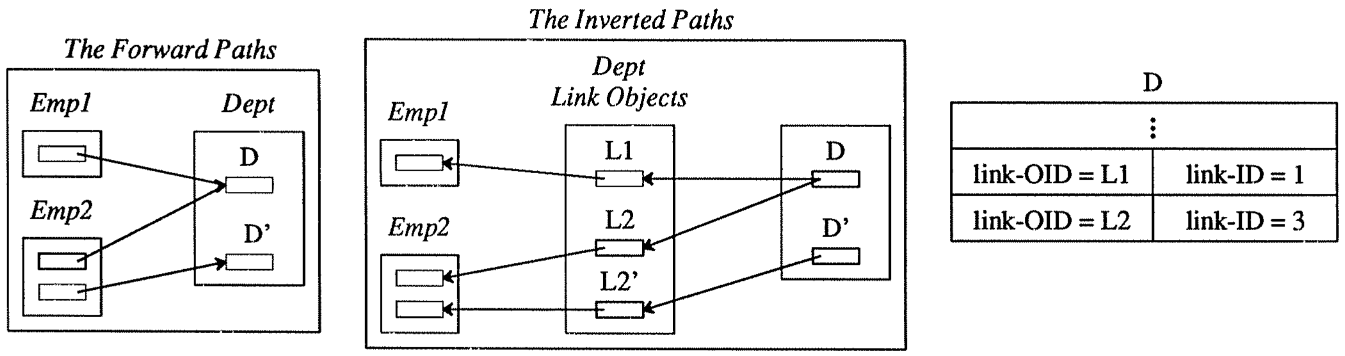
In this example, the first three paths share a common prefix, which is different from the fourth path's prefix. Therefore, a new link ID is generated for the forth path to reflect the fact that its first link is different from the first link of the other three paths and also to reflect the fact that the link Emp2.dept^{-1} cannot be shared by the other three paths.

Figure 5 illustrates the way the above replication paths are handled. Based on the preceding discussion, most of Figure 5 should be clear. The key thing to observe about Figure 5 is that only one link object (L1) is used to propagate updates in the first three replication paths. L1 and Emp1.dept^{-1} are one in the same in this example. Another thing to observe about Figure 5 is the way two link objects have been generated for D. L1 is needed to propagate updates in the first three replication paths, while L2 is needed to propagate updates in the fourth replication path.

Based on Figure 5, it should be clear how multiple replication paths are handled in the general case. It basically comes down to sharing links in inverted paths whenever possible and generating a link object for an object O whenever O enters a link in which it does not already appear.

4.2. The Space Overhead of In-Place Replication

One of the concerns with in-place replication is the amount of extra space that is required to support it. The very fact that a replication path has been created indicates that we are willing to tolerate the extra space required to store



⋮	
link-OID = L1	link-ID = 1
link-OID = L2	link-ID = 3

Figure 5: Determining when to Propagate an Update

replicated data. However, for objects along the replication path (as opposed to the objects in which replicated values are stored), we still need to worry about the space required for (link–OID, link–ID) pairs.

Fortunately, there are three mitigating factors that come into play here. First, the size of (link–OID, link–ID) pairs do not have to be large. This is mainly due to the fact that link IDs which are not in use can be reused and can probably kept as small as 8 bits. Second, in a typical environment, we would expect only a few replication paths per set (much like there are typically only a few indexes built per set), and therefore it is unlikely that an object will lie on several replication paths at once. Finally, we can assume that the DBA will make reasonable modeling decisions. Creating several replication paths on very small objects would obviously be a poor modeling decision unless the performance gains made the storage overhead worthwhile.

4.3. Optimizing Inverted Paths

Before moving on to our next replication strategy, it is important to note that our storage structures for inverted paths can be optimized in a number of ways. This section takes a cursory look at some of the optimizations that are possible.

4.3.1. Eliminating Link Objects when Possible

One way our inverted path structure can be optimized is to eliminate link objects when they contain a small number of OIDs. For example, suppose a link object L has only the OID x stored in it. If that is the case, then L can be eliminated, and x can be stored directly in the object(s) that reference L. The result is that L would no longer need to be retrieved to propagate an update. The space required to store L's OID is the same as the space required to store x, so there is no reason *not* to make this optimization. In general, it may even be worthwhile to eliminate L if it has less than n OIDs stored it, where n is on the order of two or three, or perhaps even larger. Using physical schema information about object size, it should be possible to determine a reasonable value for n on a type-by-type basis.

4.3.2. Clustering Related Link Objects in N-Level Paths

Another way our inverted path structure can be optimized is to cluster related link objects in an N-Level path. For example, suppose a 2-level replication path was defined on Emp1.dept.org.name. Further, suppose that E is an object in Emp1, E.dept = D, D.org = O, L_D is D's link object in this path, and L_O is O's link object in this path. The problem with our current scheme is that an update to O.name will require both L_O and L_D to be retrieved. Since L_O and L_D are stored in different sets, two I/Os would result.

One way to avoid two I/Os is to cluster L_O and L_D together on the same page, and in general, the same idea could be applied to replication paths with three or more levels. The only drawback of this scheme is that clustering goals can conflict. For example, supposed we add the replication path Emp1.dept.name. For this replication path, we want L_D to be clustered with the other link objects that make up Emp1.dept^{-1} so that updates to department names can be propagated with a minimal amount of I/O. Unfortunately, this means that clustering L_D with L_O would conflict with the clustering of Emp1.dept^{-1}. The way such conflicts might be resolved is left for future study.

4.3.3. Collapsing N-Level Inverted Paths to 1-Level Inverted Paths

The final optimization we consider is collapsing inverted paths with two or more levels into inverted paths with just 1-level. To demonstrate how this is done, suppose we take our previous example, where Empl.dept.org.name is replicated. To collapse the inverted path, we essentially combine the links Empl.dept^{-1} and dept.org^{-1} to form the collapsed link Empl.org^{-1}.

Figure 6 illustrates the way this would work. In the collapsed version of the inverted path, updates to O can be propagated directly to Empl via the link Empl.org^{-1}. It is important to note that the OIDs for E1, E2, and E3 that appear in O's link object would have to be tagged in some way to indicate their association with D. The tags would be needed to handle updates to D.org. For example, if D.org is set to some other object in Org, say X, then the OIDs of E1, E2, and E3 will have to be moved from O's link object to X's link object. Without the tags, there would be no way to know that the OIDs of E1, E2, and E3 need to be moved.

Although it would appear that it is always a good idea to collapse inverted paths, there are two reasons why this is not necessarily so. First, a collapsed path is more costly to maintain. For example, when D.org is updated, the OIDs of E1, E2, and E3 have to be moved. In contrast, in the uncollapsed version, only the OID of D would have to be moved. Second, collapsed paths prohibit the sharing of some links. For example, in the collapsed version, the link object of D cannot be shared due to the fact that it no longer exists. Even with these problems, though, collapsed paths may still prove useful in many situations, particularly when reference paths are static and when there is little opportunity for sharing links.

5. THE SEPARATE REPLICATION STRATEGY

In situations where sharing is heavy and where there is a moderate to high probability that replicated data will be updated, in-place replication is likely to perform poorly. For example, suppose replication paths were defined on Empl.dept.name and Empl.dept.budget, and suppose that departments tend to be large, consisting of, say, a thousand employees. If the name or budget of a DEPT object D is updated, then with in-place replication that update would have to be propagated to every object in Empl that references D. Clearly, if D.name or D.budget is updated with even moderate frequency, the cost of propagating updates will be so great that field replication would provide little benefit, if

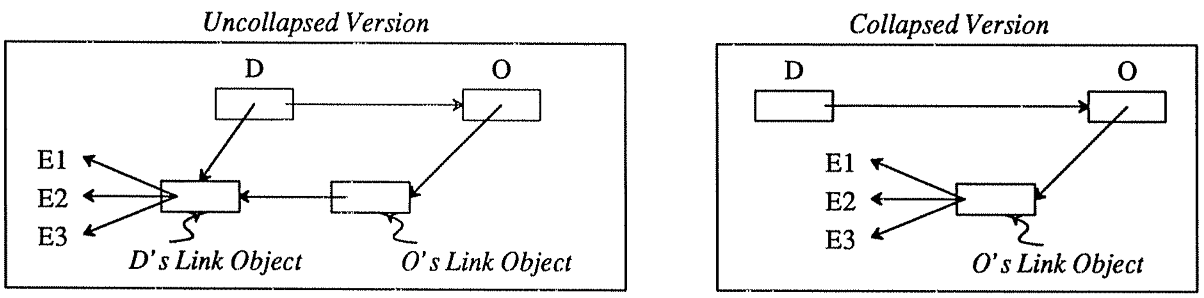


Figure 6: Collapsing a 2-Level Inverted Path

any.

Separate replication eliminates this problem by reducing the cost of propagating updates. This is achieved by storing replicated values in separate objects which are shared within a set. Separate replication is analogous to the *cache separately* strategy of POSTGRES [Jhin88], which is described in the section on related work. Figure 7 illustrates the way separate replication would work for our example. As shown, the replicated values for Empl.dept.name and Empl.dept.budget are stored in separate objects, which are shared. (Note that the replicated values for D1.name and D1.budget are stored together in *one* object, and similarly for D2.) Because replicated values are shared, propagating updates is less costly. For example, if D1.name is updated, then that update only has to be propagated to the object associated with D1.name. In contrast, with in-place replication, the update would have to be propagated to E1 and E2. The way updates are propagated with separate replication will be described shortly. In Figure 7, it is important to notice that the objects in which replicated data is stored are kept in same order as the corresponding objects in Dept. This is done to ensure that propagating updates generates as little I/O as possible.

One thing that needs to be emphasized is that replicated values are *not* shared between sets with separate replication. For example, suppose replication paths were defined on Empl.dept.name and Emp2.dept.name. In this case, two sets would be generated for replicated values. One set would be used to store the replicated values for Empl.dept.name, while the other set would be used to store the replicated values for Emp2.dept.name; the two sets would not be shared. For reasons that will become clear shortly, we want to cluster the replicated values of a particular set as tightly as possible. By maintaining separate sets, we ensure that the clustering of one set's replicated values does not interfere with the clustering of another set's replicated values.

Separate replication can also be used for replication paths with two or more levels. Replicated values are stored the same way in n-level paths that they are in 1-level paths, as illustrated in Figure 8, which will be discussed in more detail shortly.

5.1. Why Separate Replication will do Better than No Replication

At first glance, it may appear that separate replication provides no cost benefit for retrievals (at least in 1-level paths). After all, a functional join still has to be performed to access replicated values. While this may be true for single-object retrievals, for retrievals that access many objects in the same set (a set scan, for example) it should be possible to physically cluster replicated values so tightly that relatively few I/Os would be required to retrieve all replicated

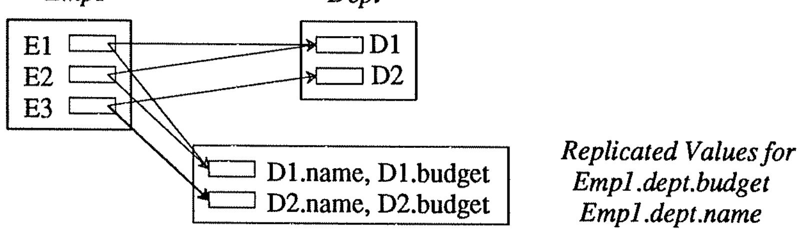
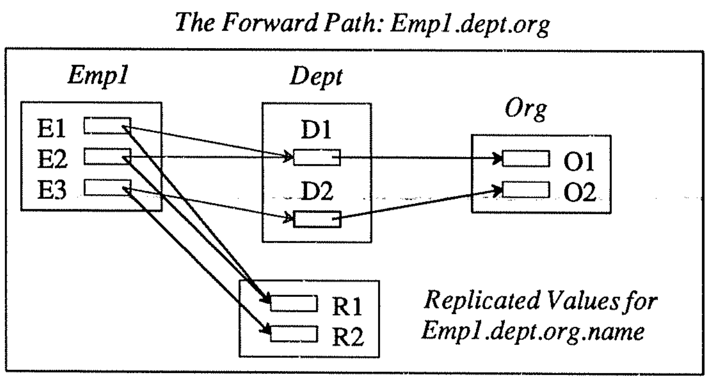


Figure 7: Separate Replication for Empl.dept.name, Empl.dept.budget



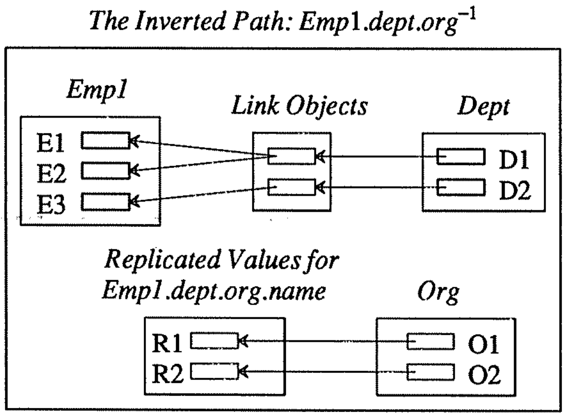


Figure 8: Inverted Paths in Separate Replication

values, particularly if replicated values are small. As a result, for retrievals such as scans, separate replication should outperform no replication. Later in the paper we present results which suggest that this is indeed the case. Another situation where separate replication should outperform no replication is in n-level references. This follows because separate replication effectively reduces an n-level reference to a 1-level reference.

5.2. Propagating Updates in Separate Replication

With separate replication, updates to data fields (as opposed to reference attributes) are always handled in the same manner, regardless of whether a 1-level or an n-level replication path has been defined. This is illustrated in the right hand side of Figure 8. As shown, the objects in Org point directly to the objects containing replicated values (R1 and R2). Taking the object O1 as an example, updates to O1.name are propagated by simply retrieving the object R1 and updating it. Although it is not shown, O1 contains R1's OID, a reference count for R1, and a tag of some sort to indicate which fields have been replicated in R1 so we know which updates to propagate to R1.

When reference attributes such as E1.dept are updated, things get slightly more complicated. As Figure 8 indicates, for replication paths with two or more levels, an inverted path needs to be maintained. The inverted path is the same in all respects as the one used with in-place replication, except that there is one less level. That is, with in-place replication, an n-level replication path requires an n-level inverted path to be maintained. In contrast, with separate replication, an n-level replication path requires an (n-1)-level inverted path to be maintained. The reason why an inverted path still needs to be kept is to propagate updates to reference attributes. In the replication path Empl.dept.org.name, for example, the link Empl.dept^{-1} is needed to handle updates to the reference attribute 'org'. If D2.org is changed from O2 to O1, then E3 must be updated so that it references R1, rather than R2. The link Empl.dept^{-1} is needed to propagate the update to E3.

In general, everything that was said earlier about inverted paths in the section on in-place replication, including space issues, links IDs, and optimizations, applies to separate replication as well, so we will not discuss the matter further. The maintenance of an inverted path is the same in almost all respects. The only exception is that updates to reference attributes are handled somewhat differently with separate replication. Although the previous example only demonstrated how they are handled in a 2-level path, it should be clear how the general case is handled.

5.3. Supporting both In-place and Separate Replication

Before leaving this section, it should be noted that in-place and separate replication can both be supported at the same time. The two strategies do not conflict with each other, and in fact, links can even be shared by the two strategies. We view this as important because there will undoubtedly be applications where a mix of in-place and separate replication provides the best performance.

6. COMPARING THE REPLICATION STRATEGIES

In this section an analytical cost model is developed to compare no replication, in-place replication, and separate replication. Only queries with 1-level functional joins are considered. Although the model considers only one class of queries, it does give some feel for how beneficial replication can be and under what circumstances it breaks down, at least for simple 1-level queries. The cost model to be described is based on the following schema:

```
define type RTYPE              define type STYPE
(                              (
    sref: ref STYPE,               repfield:SCALAR-TYPE
    various fields...              various fields...
)                              )

             create R: {own ref RTYPE}
             create S: {own ref STYPE}
             replicate R.sref.repfield
```

As shown, there are two sets in the model, R and S, with objects in R referencing objects in S. For both in-place and separate replication, a replication path has been defined on R.sref.repfield, where repfield is a scalar field that appears in members of S. Two types of queries are considered in the model:

Read Query: **retrieve** (R.fields, R.sref.repfield)
where... some clause on a scalar field R.field$_r$

Update Query: **replace** (S.fields = newvalues, S.repfield = newvalue)
where... some clause on a scalar field S.field$_s$

Read queries read data from R and also along the path R.sref.repfield, while update queries modify data in S, including the replicated field, repfield. Therefore, read queries model those queries that read replicated data, while update queries model those queries that update replicated data. Note that the clause in read queries is on the scalar field field$_r$. We will assume that this field is indexed by a B^+ tree. The same assumption is made for the field field$_s$, which appears in update queries.

In order to compare replication strategies, an expected I/O cost function, C_{total} is computed for each strategy. Let:

C_{read} = the I/O cost of processing a read query.
C_{update} = the I/O cost of processing update query.
P_{update} = the probability that an update query will be executed.

For a given query mix, C_{total} is then defined as follows:

$$C_{total} = (1 - P_{update})C_{read} + P_{update}C_{update}$$

In the analysis that follows, P_{update} is varied from 0 to 1, and the resulting values for C_{total} are used to compare the different replication strategies.

6.1. File Structure in the Model

Before describing the cost model in more detail, it is instructive to consider Figure 9, which illustrates the files that are involved in the model. Here we are assuming that each object set is stored as a single disk file. As shown, the number of files differs with each replication strategy. With no replication, there are only the files for storing R and S. The arc from R to S in the figure reflects the fact that objects in R reference objects in S. This notation is used for all the file structures. With in-place replication, the file structure is slightly more complex. There are three files: R and S as before, and the file L, which is used to store links objects for the inverted path $R.sref^{-1}$. With separate replication there are also three files, namely, R, S, and S'. The file S' is used to store replicated values for the path R.sref.repfield.

One thing to notice in Figure 10 is that the objects in L and S' are assumed to be stored in the same order as the objects in S which reference them. By ordering L and S' this way, updates are less expensive to propagate, as explained in the sections which introduced in-place and separate replication.

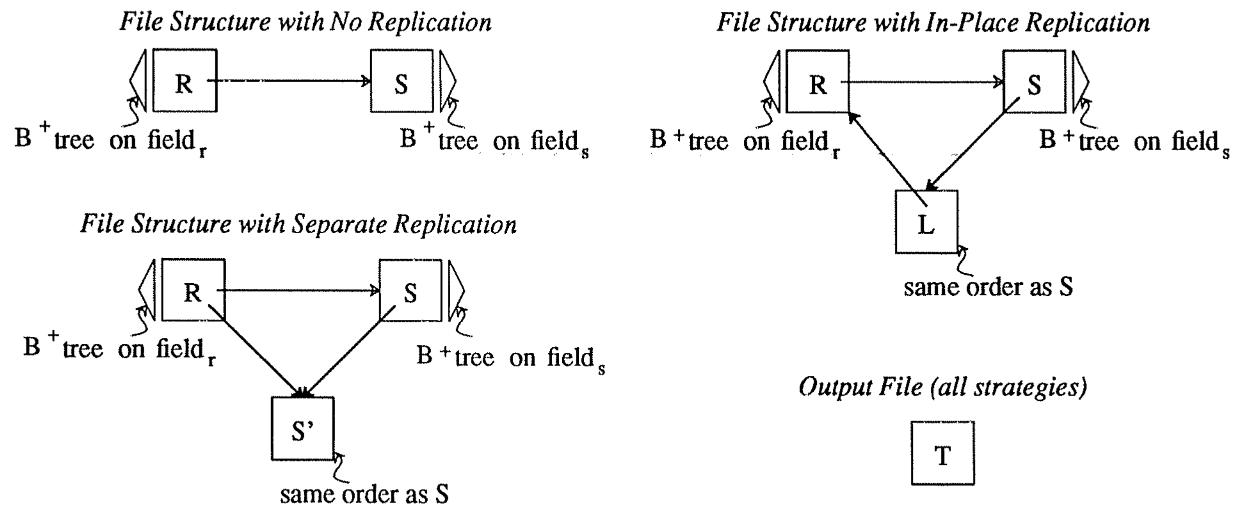

File Structure with No Replication
R
S
B+tree on field_r
B+tree on field_s
File Structure with In-Place Replication
R
S
L
B+tree on field_r
B+tree on field_s
same order as S
File Structure with Separate Replication
R
S
S'
B+tree on field_r
B+tree on field_s
same order as S
Output File (all strategies)
T

Figure 9: The File Structure with each Replication Strategy

6.2. Assumptions in the Model

Several assumptions are required in the model. The first and most important assumption we make is that R and S are *relatively unclustered*; that is, we assume that objects in R are *not* ordered by their references to S. (With separate replication, S' is kept in the same order as S; therefore, R and S' are also assumed to be relatively unclustered.) We make this assumption because we feel it represents the most typical case; the objects in R would typically be ordered by the value of some data field, not by their references to S. Bear in mind that this is a key assumption and has a considerable impact on the analysis, as it increases the cost of performing the functional join between R and S that is required by read queries. If R and S were relatively clustered, n consecutive objects in R would reference m objects in S that are clustered together on one or more contiguous disk pages. In contrast, when R and S are relatively unclustered, those m objects in S will generally be scattered over m non-contiguous disk pages (or at least close to m non-contiguous disk pages). Therefore, the amount of I/O required to join a subset of R to its corresponding subset in S will be significantly higher when R and S are relatively unclustered.

Although the analysis is not carried out here, it should be clear that replication will be less beneficial when R and S are relatively clustered. This is because replication works by lowering or eliminating the cost of performing the functional join between R and S. When R and S are relatively clustered, the cost of performing the functional join will be lower, and therefore, replication cannot offer as much savings.

The next assumption we make is that functional joins are always performed in an optimal way in the sense that if a page is required in a functional join, then that page will be read only once in performing the join. For example, suppose there are n objects $r_1, r_2, r_3, \dots r_n$ in R that reference the object s in S. Further suppose that $r_1, r_2, r_3, \dots r_n$ are scattered over n non-contiguous pages in R (which is possible because R and S are assumed to be relatively unclustered), and that s is stored on page p_s. What our assumption says is that when R and S are joined, only $n + 1$ pages will be read to join $r_1, r_2, r_3, \dots r_n$ with s. Of course, for large joins, this is an overly optimistic assumption. If a naive join algorithm is used (for example, one that joins each object in R with S independently) then p_s may be paged out of memory before $r_1, r_2, r_3, \dots r_n$ have all been joined to s. If so, then p_s will have to be read more than once.

The assumption about optimal joins is made largely because it simplifies the analysis. It allows us to ignore buffering and the exact details of how an efficient join algorithm might operate. Keep in mind, though, that if anything, the assumption about optimal joins will tend to *bias the results in favor of no replication.* Without replication, more data is joined in read queries, and as a result, it is less likely that functional joins can be performed in an optimal way. Later, our results will show that replication outperforms no replication by a large margin in many instances. If we had assumed a realistic join algorithm, however, our results for replication would have been even better, due to the large amount of data being joined in some cases.

The last assumption we make is that read and update queries always access R and S through the indexes on field_r and field_s, respectively. This assumption is made because we feel that it most accurately models the "typical" database query, since most queries will make use of some index. It also allows us to treat the accesses to R and S uniformly in our analysis.

6.3. The Parameters of the Cost Model

The parameters of the cost model are listed in Figure 10. There are a large number of parameters, but only a few of them are actually varied here. Moreover, most of the parameters are not really parameters per se, but rather functions of a small set of "core" parameters. Defaults are only listed for the core set of parameters. To insure meaningful results, the values for B, h, $sizeof\,(type-tag)$, and $sizeof\,(OID)$ were taken from the EXODUS Storage Manager [Care86].

From Figure 10, the meaning of most parameters should be clear; f, f_r and f_s require further explanation, however. The parameter f denotes the sharing level of objects in S. In the model, we are assuming that every object in S is referenced (or shared) by f objects in R. The parameters f_r and f_s denote the selectivity of read and update queries, respectively. Each read query reads $f_r\,|R|$ objects in R, and each update query updates $f_s\,|S|$ objects in S.

One thing that should be noted about the defaults is that the values for r and s represent the size of objects in R and S with no replication. Consequently, with in-place or separate replication, r and s need to be adjusted. For example, with in-place replication, r must be increased by k to account for the replicated data. Rather than introduce more

Parameter	**Definition**	**Default (for core parameters)**
B	number of bytes in a disk page available for user data.	4056 bytes
h	storage overhead per object (i.e., object header).	20 bytes
m	B^+ tree fanout.	350
$\lvert S \rvert$	number of objects in S.	10,000
f	sharing level of objects in S.	1 (varied)
f_r	selectivity of the clause in read queries.	.001 (varied)
f_s	selectivity of the clause in update queries.	.001
$sizeof\,(OID)$	size of OIDs.	8 bytes
$sizeof\,(link-ID)$	size of link IDs.	1 byte
$sizeof\,(type-tag)$	size of type-tags.	2 bytes
k	size of the replicated field, repfield.	20 bytes
r	size of objects in R (varies with replication strategy).	100 bytes
s	size of objects in S (varies with replication strategy).	200 bytes
t	size of objects in T.	100 bytes
$\lvert R \rvert$	number of objects in R ($\lvert R \rvert = f\,\lvert S \rvert$).	
s'	size of objects in S' ($s' = k + sizeof\,(type-tag)$).	
l	size of objects in L ($l = 1 + sizeof\,(type-tag) + f sizeof\,(OID)$).	
O_r	objects per page in R ($O_r = \lfloor B\,/\,(h + r) \rfloor$).	
O_s	objects per page in S ($O_s = \lfloor B\,/\,(h + s) \rfloor$).	
$O_{s'}$	objects per page in S' ($O_{s'} = \lfloor B\,/\,(h + s') \rfloor$).	
O_l	objects per page in L ($O_d = \lfloor B\,/\,(h + l) \rfloor$).	
O_t	objects per page in T ($O_t = \lfloor B\,/\,(h + t) \rfloor$).	
P_r	pages in R ($P_r = \lceil \lvert R \rvert \,/\, O_r \rceil$).	
P_s	pages in S ($P_s = \lceil \lvert S \rvert \,/\, O_s \rceil$).	
$P_{s'}$	pages in S' ($P_{s'} = \lceil \lvert S \rvert \,/\, O_{s'} \rceil$).	
P_l	pages in L ($P_l = \lceil \lvert S \rvert \,/\, O_l \rceil$).	
P_t	pages in T ($P_t = \lceil f\,\lvert R \rvert \,/\, O_t \rceil$).	

Figure 10: The Parameters of the Cost Model

notation, the cost equations that follow will tacitly assume that r and s (and the parameters that depend on r and s) reflect these adjustments. Therefore, the values actually used for r and s will differ from strategy to strategy, even though the same symbols will be used in the equations. The same will be true of the other parameters that depend on r and s.

6.4. The Setting for the Analysis

The model that has been described will be analyzed in two settings. In the first setting, we will assume that the B^+ trees on field_r and field_s are both unclustered indexes. In the second setting, we will assume that both indexes are clustered indexes. The reason for considering these particular settings is because they represent opposite ends of the performance spectrum. When both indexes are unclustered, more overall I/O is generated and as a result, the savings in I/O due to replication will be smaller as a percentage of the total I/O. Conversely, when both indexes are clustered, less overall I/O will be generated and therefore the savings in I/O due to replication will be larger on a percentage basis.

6.5. Cost Analysis when Unclustered Indexes are Used

In this section, cost equations are derived for read and update queries under the assumption that unclustered indexes are used. Throughout the section we shall use C_{read} to denote the net cost of a read query and C_{update} to denote the net cost of an update query. We shall also use $C_{action/X}$ to denote the cost of performing *action* on file X, where *action* is *read*, *update*, or *generate* in the case of the output file T.

In the cost equations that follow, it should be noted that no distinction is made between sequential I/O and random I/O. We initially distinguished between the two, but found that it did not significantly change our results. This is due to the fact that our results are presented in terms of relative cost rather than absolute cost, and all the strategies we examine receive roughly the same benefit from sequential I/O.

6.5.1. Read Queries with No Replication

In terms of I/O, processing a read query with no replication consists of reading the index on field_r, reading R, reading S (to join R and S), and generating the output file T. Reading the index on field_r consists of descending the B^+ tree to a leaf, then scanning across the leaves to obtain the OIDs of the $f_r\,|R|$ objects in R that satisfy the clause of the read query. The cost to descend the index to the first leaf page is:

$$\lceil \log_m |R| \rceil$$

And the cost to read the remaining leaf pages is:

$$\left\lceil \frac{f_r\,|R|}{m} - 1 \right\rceil$$

Therefore, the net cost to read the index is:

$$C_{read/index_r} = \lceil \log_m |R| \rceil + \left\lceil \frac{f_r\,|R|}{m} - 1 \right\rceil$$

To calculate the cost to read R, we first consider the probability that a given page P_i in R is *not* read by a read query. Since our access to R is unclustered, we can assume that any given subset of $f_r\,|R|$ objects is just as likely to be read by a read query as any other subset. Therefore, the probability that page P_i is not read is equal to the probability of choosing a subset of $f_r\,|R|$ objects from R such that the chosen subset contains no object from page P_i. This is equal to:

$$Prob(page\ P_i\ in\ R\ is\ not\ read) = \frac{\binom{|R| - O_r}{f_r\,|R|}}{\binom{|R|}{f_r\,|R|}}$$

The expected number of pages that are read in R is therefore:

$$expected\ pages\ read\ in\ R = P_r \left[1 - \frac{\binom{|R| - O_r}{f_r\,|R|}}{\binom{|R|}{f_r\,|R|}} \right]$$

This same quantity was derived in another context [Yao77]. If we let:

$$y(a, b, c) = 1 - \frac{\binom{a - b}{c}}{\binom{a}{c}}$$

then the cost to read R is:

$$C_{read/R} = P_r\, y(|R|, O_r, f_r\,|R|)$$

The function $y()$ has been introduced because it will be needed again shortly. To calculate the cost to read S, we use more or less the same technique that we used with R. In this case, however, the probability that a given page P_i in S is not read is equal to the probability of choosing a subset of $f_r\,|R|$ objects from R such that the chosen subset contains no object that references an object in page P_i. Since there are fO_s objects in R that reference page P_i, the probability that page P_i is not read is:

$$Prob(page\ P_i\ in\ S\ is\ not\ read) = \frac{\binom{|R| - fO_s}{f_r\,|R|}}{\binom{|R|}{f_r\,|R|}}$$

Therefore, the cost to read S is:

$$C_{read/S} = P_s\, y(|R|, fO_s, f_r\,|R|)$$

Finally, the cost to generate the output file T is:

$$C_{generate/T} = P_t$$

Summing it up, the net cost to process a read query with no replication is:

$$\begin{aligned} C_{read} &= C_{read/index_r} + C_{read/R} + C_{read/S} + C_{generate/T} \\ &= \lceil \log_m |R| \rceil + \left\lceil \frac{f_r |R|}{m} - 1 \right\rceil + P_r y(|R|, O_r, f_r |R|) + P_s y(|R|, fO_s, f_r |R|) + P_t \end{aligned}$$

6.5.2. Update Queries with No Replication

Processing an update query with no replication consists of reading the index on field$_s$ and updating S. (We are assuming that update queries do not cause the index on field$_s$ to be modified.) The cost equation for reading the index on field$_s$ is similar to the equation for $C_{read/index_r}$. Updating S consists of reading pages in S, updating them, and then writing them to disk. The expected number of pages read and written in S can be calculated in the same manner that was used to calculate the expected number of pages in R that are read by a read query. Based on this observation, the cost to update S is:

$$C_{update/S} = 2P_s y(|S|, O_s, f_s |S|)$$

The net cost to process an update query with no replication is therefore:

$$\begin{aligned} C_{update} &= C_{read/index_s} + C_{update/S} \\ &= \lceil \log_m |S| \rceil + \left\lceil \frac{f_s |S|}{m} - 1 \right\rceil + 2P_s y(|S|, O_s, f_s |S|) \end{aligned}$$

6.5.3. Read Queries with In-Place Replication

Processing a read query with in-place replication consists of reading the index on field$_r$, reading R, and generating the output file T. No join between R and S is required because sref.repfield is replicated in R. The cost equations for processing a read query with in-place replication are basically the same[4] as with no replication except that the cost to read S is no longer included. Consequently, the net cost to process a read query with in-place replication is:

$$\begin{aligned} C_{read} &= C_{read/index_r} + C_{read/R} + C_{generate/T} \\ &= \lceil \log_m |R| \rceil + \left\lceil \frac{f_r |R|}{m} - 1 \right\rceil + P_r y(|R|, O_r, f_r |R|) + P_t \end{aligned}$$

[4] In the equations, it is important to remember that the values for parameters such as O_r and O_s will differ from strategy to strategy, even though the same symbols are used. For example, O_r is smaller here than with no replication because of replicated data.

6.5.4. Update Queries with In-Place Replication

Processing an update query with in-place replication consists of reading the index on field$_s$, updating S, reading L to propagate the updates in S to R, and updating R. The cost equations for reading the index on field$_s$ and for updating S are the same as with no replication. The cost equation for reading L can be derived in the same manner that $C_{read/R}$ was derived with no replication.

The cost equation for updating R is similar to the equation for $C_{update/S}$ with no replication. To derive the equation, we first observe that, because of replicated values, each update to an object in S has to be propagated to f objects in R. Therefore, a total of $f_s f\,|S|$ objects in R are updated by an update query. Since R and S are relatively unclustered, we can assume that any given subset of $f_s f\,|S|$ objects in R is just as likely to be updated as any other subset. Based on this observation, and following the reasoning that was used to derive $C_{update/S}$ with no replication, the cost of updating R is:

$$C_{update/R} = 2P_r y(|R|, O_r, f_s f\,|S|)$$

Since $|R| = f\,|S|$, this can be rewritten as:

$$C_{update/R} = 2P_r y(|R|, O_r, f_s\,|R|)$$

Adding in the other terms, the net cost to process an update query with in-place replication is therefore:

$$\begin{aligned} C_{update} &= C_{read/index_s} + C_{update/S} + C_{read/L} + C_{update/R} \\ &= \lceil \log_m |S| \rceil + \left\lceil \frac{f_s\,|S|}{m} - 1 \right\rceil + 2P_s y(|S|, O_s, f_s\,|S|) + P_l y(|S|, O_l, f_s\,|S|) \\ &\quad + 2P_r y(|R|, O_r, f_s\,|R|) \end{aligned}$$

6.5.5. Read Queries with Separate Replication

Processing a read query with separate replication is the same as with no replication except that R is joined with S' rather than S. Consequently, the cost equations for separate replication are obtained by simply substituting S' for S in the cost equations for in-place replication. Doing this, the net cost to process a read with separate replication is:

$$\begin{aligned} C_{read} &= C_{read/index_r} + C_{read/R} + C_{read/S'} + C_{generate/T} \\ &= \lceil \log_m |R| \rceil + \left\lceil \frac{f_r\,|R|}{m} - 1 \right\rceil + P_r y(|R|, O_r, f_r\,|R|) + P_{s'} y(|R|, f O_{s'}, f_r\,|R|) + P_t \end{aligned}$$

6.5.6. Update Queries with Separate Replication

Processing an update query with separate replication consists of reading the index on field$_s$, updating S, and updating S'. The cost equations for reading the index on field$_s$ and for updating S are the same as the equations for $C_{update/index_s}$ and $C_{update/S}$ with no replication, respectively. The cost equation for updating S' follows directly from the equation for $C_{update/S}$. The net cost to process an update query with separate replication is therefore:

$$\begin{aligned} C_{update} &= C_{read/index_s} + C_{update/S} + C_{update/S'} \\ &= \lceil \log_m |S| \rceil + \left\lceil \frac{f_s |S|}{m} - 1 \right\rceil + 2P_s y(|S|, O_s, f_s |S|) + 2P_{s'} y(|S|, O_{s'}, f_s |S|) \end{aligned}$$

6.6. Performance Results when Unclustered Indexes are Used

Figure 11 presents the results for unclustered indexes. The graphs were obtained by computing C_{total} which was described earlier, for each replication strategy, with P_{update} being varied from 0 to 1. For in-place and separate replication, the values for C_{total} were compared to the corresponding values for C_{total} with no replication, and the percentage difference in C_{total} was plotted. The horizontal line in the graphs therefore represents no replication. (Note that the vertical axes of the graphs were arbitrarily cutoff at 50% so that all the graphs could be placed on one page.)

Four graphs are shown in Figure 11, each one for a different sharing level f. In all the graphs, $|S|$ was fixed at 10,000 objects, and f_s at 0.001; therefore, an update query always updated 10 objects. The value of f was set at 1, 10, 20, and 50. Consequently, $|R|$ varied from 10,000 to 500,000 objects. In each graph, three lines have been drawn for both in-place and separate replication, corresponding to the read selectivity f_r being set at 0.001, 0.002, and 0.005. The size of objects in R was fixed at 100 bytes, those in S at 200 bytes, and the replicated field at 20 bytes.

Looking at the graphs, it is clear that replication can be beneficial. As expected, replication is particularly useful when the probability of an update query is low. The graphs show that in-place replication performs its best for small update probabilities and for small values of f. Its performance decreases for large values of f because the cost to propagate updates is higher in that case. (Recall that with inplace-replication, each update to an object in S has to be propagated to f objects in R.) In contrast to in-place replication, separate replication performs its best for large values of f. This is because the size advantage provided by S' in joins becomes more pronounced as f increases. By size advantage we are referring to fact that there are more disk pages in S than there are in S', which means that it costs more to join R with S than it does to join R with S' (especially if the join is large). As f increases, the size of the join in read queries also increases, so this effect becomes more pronounced. The graphs also show that, compared to in-place replication, separate replication breaks down less rapidly as the update probability is increased. This is because the cost of propagating updates is much higher with in-place replication.

It is important to note that for $f = 1$, separate replication provides almost no benefit. In this case, separate replication is essentially the same as no replication, except that the cost of processing an update query increases. For small joins such as this, S' provides virtually no size advantage over S, and functional joins between R and S' are almost as expensive as joins between R and S. In contrast, in-place replication performs extremely well under these circumstances, as the cost of propagating an update is small.

In terms of numbers, the graphs show that in-place replication always outperforms separate replication when the probability of an update query is less than roughly 0.15. For update probabilities in that range, in-place replication reduces I/O costs by approximately 15 to 45 percent. In contrast, if we exclude the case where $f = 1$, then separate replication always outperforms in-place replication when the probability of an update query exceeds roughly 0.35. For

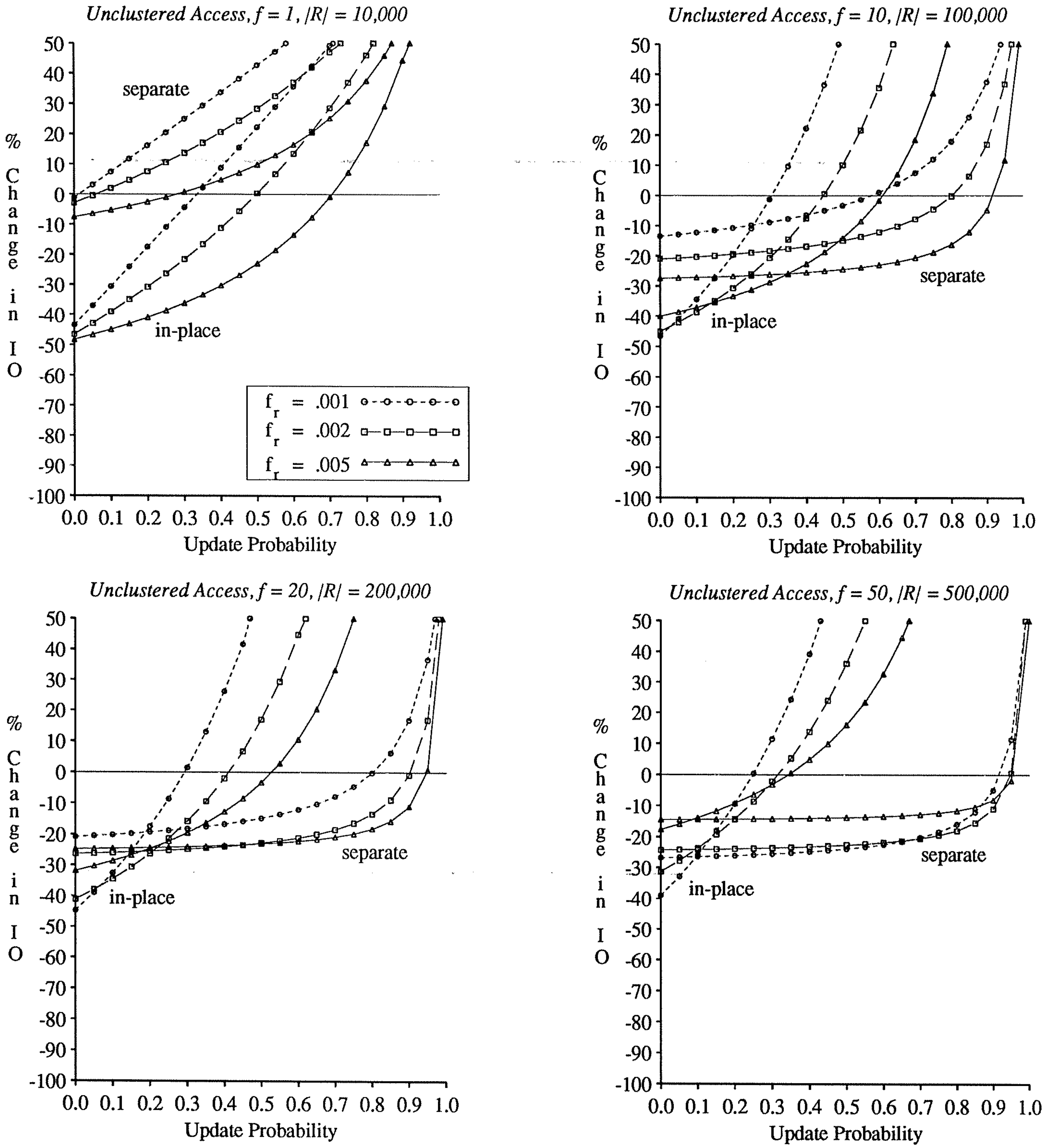


Figure 11: Results for Unclustered Indexes

a wide range of update probabilities, separate replication reduces I/O costs by approximately 10 to 30 percent. What may be surprising is the large range of update probabilities for which separate replication outperforms no replication. The reason this happens is because update queries only update 10 objects in S; since each update to an object in S only has to be propagated to one object in S', the cost of propagating updates never becomes much of a factor with separate replication.

One aspect of the graphs worth noting is that, beyond a certain point, reading more data only reduces the relative effectiveness of replication. For example, for $f = 10$, separate replication performs the best when $f_r = 0.005$, which corresponds to the case where read queries access the most data. (Recall that $|R| = f\ |S|$ and f_r is the selectivity of read queries.) By the time $f = 50$, however, $f_r = 0.005$ results in the worst performance for separate replication, and instead $f_r = 0.001$, which corresponds to the case where read queries access the least data, results in the best performance. This effect can be observed by noting the way the lines for $f_r = 0.001$ and $f_r = 0.005$ "flip" as f is increased from 10 to 50 in the graphs for separate replication. The reason why this occurs is fairly simple. As f and f_r increase, read queries read more and more data in R. At some point, so much data is accessed in R that the cost of reading R overwhelms all other costs. Since replication does not reduce the cost of reading R, the savings in I/O due to replication start to decrease at that point on a percentage basis.

To provide further insight into why graphs in Figure 11 appear as they do, selected values for C_{read} and C_{update} are presented in Figure 12. (Recall that C_{read} is the cost of a read query and C_{update} is the cost of an update query.) Rather than present values for each combination of f and f_r, only two sets of values are shown in Figure 12. The left-hand side of the table lists C_{read} and C_{update} for $f = 1$ and $f_r = 0.002$ and is representative of how C_{read} and C_{update} differ from strategy to strategy when $f = 1$. The right-hand side of the table lists C_{read} and C_{update} for $f = 20$ and $f_r = 0.002$ and is representative of how C_{read} and C_{update} differ from strategy to strategy when $f > 1$. In the table, fractional values were rounded up to the nearest unit.

Looking at the table, the reason for the shape of the graphs in Figure 12 should be clearer. By eliminating the join between R and S, in-place replication significantly reduces the cost of read queries for both $f = 1$ and $f = 20$. The cost of an update query, however, increases by roughly $2f \cdot f_s\ |S|$ compared to no replication. This is because with in-place replication each update to an object in S (of which there $f_s\ |S|$ or 10 per update query) has to be propagated to f objects in R. Like in-place replication, separate replication also reduces the cost of read queries, but only marginally when $f = 1$. As mentioned earlier, there is little cost reduction for $f = 1$ because S' provides virtually no size

	$f = 1, f_r = .002$		$f = 20, f_r = .002$	
Strategy	C_{read}	C_{update}	C_{read}	C_{update}
no replication	43	22	691	22
in-place replication	23	42	407	427
separate replication	41	42	509	42

Figure 12: Selected Values for C_{read} and C_{update} (for Unclustered Access)

advantage over S in small joins. In contrast to in-place replication, the cost of an update query in separate replication is unaffected by the value of f. Regardless of the value of f, the cost of an update query is roughly double the cost of an update query with no replication. The factor of two arises because with separate replication, each update to an object in S has to propagated to an object in S'.

6.7. Cost Analysis when Clustered Indexes are Used

The cost equations for clustered indexes are similar to the ones for unclustered indexes. The only differences are that R is accessed in clustered order in read queries, and S and S' are accessed in clustered order in update queries. Based on this observation, and referring to the cost equations for unclustered indexes, the cost equations for clustered indexes are relatively straightforward to derive. Without further explanation, they are:

No Replication:

$$\begin{aligned} C_{read} &= C_{read/index_r} + C_{read/R} + C_{read/S} + C_{generate/T} \\ &= \lceil \log_m |R| \rceil + \left\lceil \frac{f_r |R|}{m} - 1 \right\rceil + f_r P_r + P_s y(|R|, fO_s, f_r |R|) + P_t \\ C_{update} &= C_{read/index_s} + C_{update/S} \\ &= \lceil \log_m |S| \rceil + \left\lceil \frac{f_s |S|}{m} - 1 \right\rceil + 2 f_s P_s \end{aligned}$$

In-Place Replication:

$$\begin{aligned} C_{read} &= C_{read/index_r} + C_{read/R} + C_{generate/T} \\ &= \lceil \log_m |R| \rceil + \left\lceil \frac{f_r |R|}{m} - 1 \right\rceil + f_r P_r + P_t \\ C_{update} &= C_{read/index_s} + C_{update/S} + C_{read/L} + C_{update/R} \\ &= \lceil \log_m |S| \rceil + \left\lceil \frac{f_s |S|}{m} - 1 \right\rceil + 2 f_s P_s + + f_s P_l + 2 P_r y(|R|, O_r, f_s |R|) \end{aligned}$$

Separate Replication:

$$\begin{aligned} C_{read} &= C_{read/index_r} + C_{read/R} + C_{read/S'} + C_{generate/T} \\ &= \lceil \log_m |R| \rceil + \left\lceil \frac{f_r |R|}{m} - 1 \right\rceil + f_r P_r + P_{s'} y(|R|, fO_{s'}, f_r |R|) + P_t \\ C_{update} &= C_{read/index_s} + C_{update/S} + C_{update/S'} \\ &= \lceil \log_m |S| \rceil + \left\lceil \frac{f_s |S|}{m} - 1 \right\rceil + 2 f_s P_s + 2 f_s P_{s'} \end{aligned}$$

6.8. Performance Results when Clustered Indexes are Used

Figure 13 presents the results for clustered indexes. As mentioned earlier, when both indexes are clustered, less overall I/O is generated and therefore the savings in I/O due to replication is significantly larger on a percentage basis. This can be seen by comparing the graphs in Figure 13 to the ones in Figure 11. As expected, the graphs in Figure 13 follow the same general patterns that were seen earlier. Once again, in-place replication peforms its best for small values of f, while separate replication performs its best for large values of f. In-place replication is shown to be particularly effective when $f = 1$.

The surprising thing about these graphs is just how much replication reduces I/O costs when clustered indexes are used. For example, the graphs show that in-place replication again outperforms separate replication when the probability of an update query is less than roughly 0.15. In this case, however, in-place replication reduces I/O costs by approximately 55 to 90 percent over that range of update probabilities, whereas it only reduced I/O costs by approximately 15 to 45 percent before. Similarly, if we exclude the case where $f = 1$, then we see that separate replication again outperforms in-place replication when the probability of an update query exceeds 0.35. In this case, separate replication reduces I/O costs by approximately 25 to 70 percent over a wide range of update probabilities, whereas it only reduced I/O costs by approximately 10 to 30 percent before.

In Figure 14, selected values of C_{read} and C_{update} are presented to provide further insight into why the graphs in Figure 13 appear as they do. As expected, the values in Figure 14 follow roughly the same patterns that were observed in the table of Figure 12. In general, the costs of read and update queries are much smaller here because R and S are accessed in clustered order. The one exception is the cost of an update query with in-place replication, which remains large. It remains large because the cost of propagating updates from S to R with in-place replication does not change when clustered indexes are used.

7. COMPARISON WITH RELATED WORK

7.1. Caching in POSTGRES

Readers will notice that our work bears a strong similarity to the work that has been done in POSTGRES on caching the results of procedural fields [Hans87, Sell87, Ston87, Hans88, Jhin88]. Three basic caching strategies have been analyzed in POSTGRES [Hans87, Hans88, Jhin88]:

cache-in-tuple and invalidate

In this strategy, cached results are stored directly in tuples. Cached results are marked as invalid when they become out-of-date due to updates. Special locking mechanisms are used to detect when cached results become out-of-date.

cache-separately and invalidate

This is essentially the same as the cache-in-tuple strategy, except that cached results are stored in a separate *Cache* relation and shared when possible. Hashing is used to access the Cache relation.

update cache

This is a variant of the other two strategies in which cached results are kept up-to-date rather than being invalidated. Algebraic view maintenance algorithms [Blak86, Hans87, Hans88] are used to update cached results.

Based on the above description, it is clear that field replication and caching in POSTGRES share many things in common. Our two basic replication strategies were in fact directly motivated by the cache-separately and cache-in-

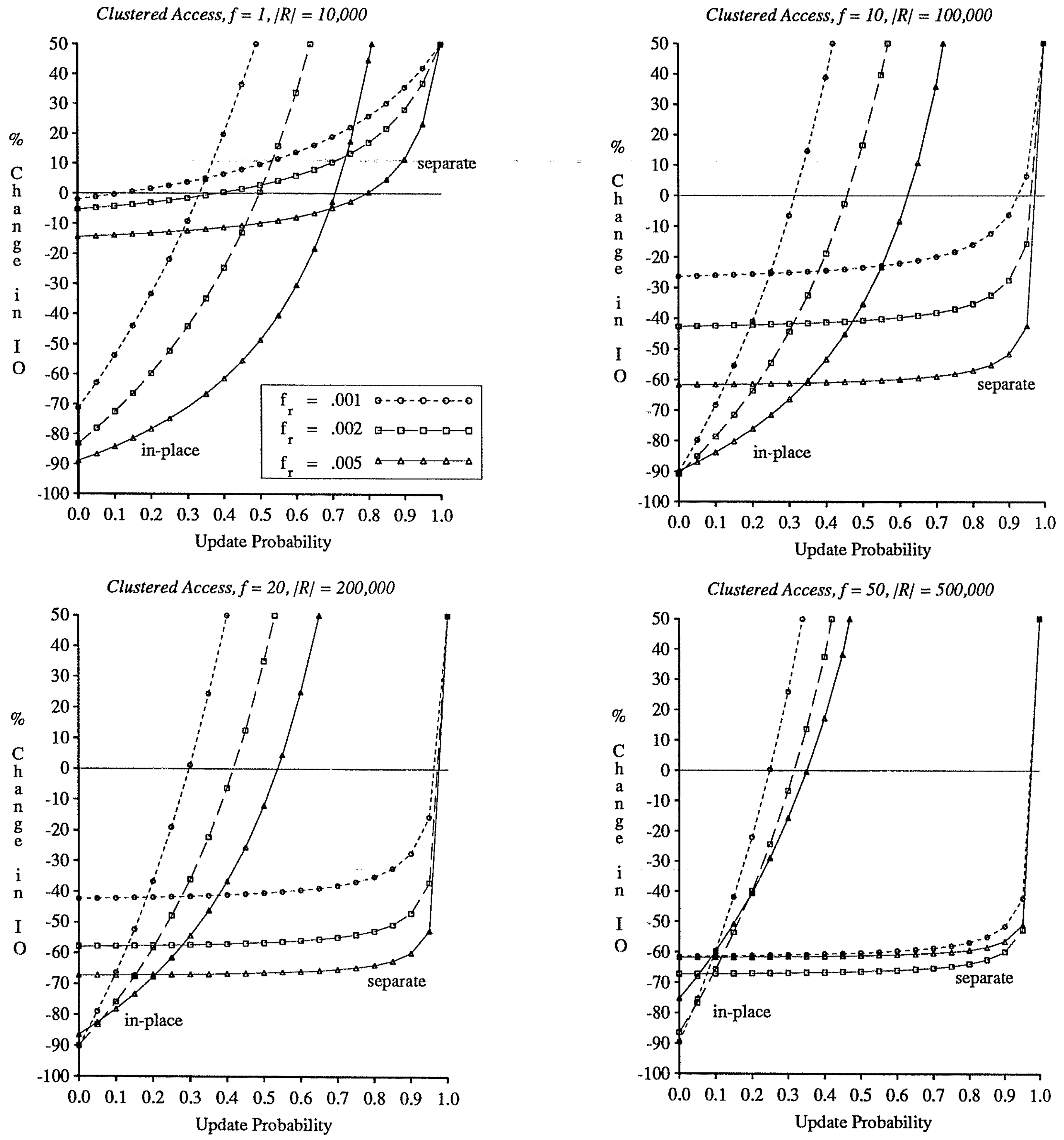


Figure 13: Results for Clustered Indexes

	$f = 1, f_r = .002$		$f = 20, f_r = .002$	
Strategy	C_{read}	C_{update}	C_{read}	C_{update}
no replication	24	4	316	4
in-place replication	4	24	32	400
separate replication	23	6	133	6

Figure 14: Selected Values for C_{read} and C_{update} (for Clustered Access)

tuple strategies of POSTGRES. In some respects, field replication can be viewed as a primitive form of caching in which only equijoins with projection are permitted in procedural fields. On closer inspection, though, field replication and caching are found to differ in three major ways. First, in contrast to POSTGRES, we are not working in the context of the relational model and consequently, our mechanisms for keeping replicated data consistent are very different. Second, because field replication is more primitive than caching, it should prove easier to implement and also more efficient. We view this as especially important because we feel that field replication will be able to handle many of the cases for which caching is useful. Field replication should prove more efficient than caching because, among other things, special locks do not have to be maintained to invalidate replicated data, nor do general view maintenance algorithms have to be used to keep replicated data up-to-date. Finally, query optimization should be easier with field replication. In caching, there is always the possibility that cached results may either be invalid or not present when needed. Query optimization would appear to be difficult in such a dynamic environment. In contrast, with field replication, replicated values are *always* guaranteed to exist and, moreover, are guaranteed to be up-to-date As a result, optimization techniques that use static analysis and the cost models described here can be applied.

7.2. Path Indexes in Gemstone

Our work also borrows from [Maie86], which described the design and implementation of *path indexes* in the Gemstone object-oriented database system. A path index is basically the same as a normal index, except that it is defined on a reference path. For example, a path index on Empl.dept.name would map department names to EMP objects in Empl. One of the key issues addressed by the Gemstone researchers was how to maintain an inverted path. In essence, maintaining an index on Empl.dept.name boils down to maintaining the inverted path Empl.dept.name^{-1}. In Gemstone, an inverted path is essentially broken down into a series of index components, which serve the same purpose as our links but are implemented with B$^+$ trees.

The obvious difference between our inverted path structure and the one used in Gemstone is that our inverted path structure provides a direct object-to-object mapping, whereas in Gemstone the mapping is indirect via B$^+$ tree components. The main advantage in using B$^+$ trees to implement inverted paths is that associative lookups are possible. For example, if the link Empl.dept^{-1} is implemented as a B$^+$ tree, we can ask whether the DEPT objects with OIDs x through y are referenced by Empl, and this can be done *without* accessing the Dept set. The main disadvantage in using B$^+$ trees to implement inverted paths is that, assuming B +trees are two levels, traversing an inverted path requires roughly twice as much I/O. Another disadvantage is that no clustering options are available. Since we were predominantly concerned about I/O costs and not interested in associative lookups, we chose to implement inverted paths via a direct object-to-object mapping.

8. CONCLUSION

This paper introduced the notion of field replication and then described various ways to implement it. Two basic replication strategies were discussed, namely, in-place and separate replication. For both of these strategies, we showed how inverted paths can be used to keep replicated data consistent. A significant part of the paper was devoted to describing how inverted paths can be efficiently implemented. Although much of the discussion was based on the EXTRA data model, the ideas presented here can also be extended to other data models that support reference attributes or referential integrity facilities of the type discussed in [Date87].

Finally, we developed an analytical cost model to give some feel for how beneficial field replication can be in executing simple queries. The model compared the I/O costs of executing simple queries with no replication, in-place replication, and separate replication, and an analysis was carried out in two settings, one with unclustered indexes another with clustered indexes. For unclustered indexes, if we exclude the case where the sharing level was equal to one, we found that in-place replication reduced I/O costs by 20 to 45 percent when the update probability was small (less than 0.2). Under the same conditions, but for a much wider range of update probabilities, we found that separate replication reduced I/O costs by 15 to 30 percent. For clustered indexes, the improvement was even more dramatic. There we found that in-place replication reduced I/O costs by 40 to 90 percent when the update probability was small (less than 0.2), while for separate replication, I/O costs were reduced by 25 to 70 percent over a wide range of update probabilities. Thus, while field replication is a relatively simple notion, the analysis showed that it can provide significant performance gains in many situations.

As far as future work goes, we are currently investigating replication techniques in which updates are not propagate until needed, indexes on replicated data, and ways in which inverted paths can be used for referential integrity and in implementing inverse functions (or bidirectional reference attributes). We also plan on using replication in our implementation of the data model and query language for EXODUS [Care88].

REFERENCES

[Bane87] J. Banerjee et al., "Data Model Issues for Object-Oriented Applications," *ACM Trans. on Office Info. Sys.* 5(1), Jan.1987.

[Blak86] J. Blakeley et al., "Efficiently Updating Materialized Views," *Proc. of the 1986 ACM-SIGMOD Conf.*, Washington DC, May 1986.

[Care86] M. Carey et al., "Objects and File Management in the EXODUS Extensible Database System," *Proc. of 1986 VLDB Conf.*, Kyoto, Japan, Aug. 1986.

[Care88a] M. Carey et al., "A Data Model and Query Language for EXODUS," *Proc. of the 1988 ACM-SIGMOD Conf.*, Chicago, Ill, 1988.

[Care88b] M. Carey and D. Dewitt, "The EXODUS Extensible DBMS Project: An Overview," Univ. of Wisconsin Tech Report #808.

[Cope84] G. Copeland and D. Maier, "Making Smalltalk a Database System," *ACM Trans. on Database Systems* 4(4), Dec. 1979.

[Date87] C. Date, "A Guide to THE SQL STANDARD," Ch. 11, pp. 113-120, Addison Wesley, Reading Mass. 1987.

[Ditt86] K. Dittrich, "Object-Oriented Database Systems: the Notion and the Issues," *Proc. of the 1st Int'l Workshop on Object-Oriented Database Systems*, Asilomar, CA, Sept. 1986.

[Fish87] D. Fishman et al. "Iris: An Object-Oriented Database Management System," *ACM Trans. on Office Info. Sys.* 5(1), Jan.1987.

[Hans87] E. Hanson, "A Performance Analysis of View Materialization Strategies", *Proc, of the 1987 ACM-SIGMOD Conf.*, San Francisco CA, May 1987.

[Hans88] E. Hanson, "Processing Queries Against Database Procedures, A Performance Analysis," *Proc. of the 1988 ACM-SIGMOD Conf.*, Chicago, Ill, 1988.

[Horn87] M. Hornick and S. Zdonik, "A Shared, Segmented Memory System for an Object-Oriented Database," *ACM Trans. on Office Info. Sys.* 5(1), Jan. 1987.

[Hull87] R. Hull and R. King, "Semantic Database Modeling: Survery , Applications, and Research Issues," *ACM Comput. Surveys* 19(3), Sept. 1987.

[Jhin88] A. Jhingran, "A Performance Study of Query Optimization Algorithms on a Database System Supporting Procedures," *Proc. of the 14th VLDB Conf.*, Los Angeles CA, 1988.

[Kim87] W. Kim et al., "Composite Object Support in an Object-Oriented Database System," *Proc. of the 2nd ACM OOPSLA Conf.*, Orlando, FL., 1987.

[Maie86a] D. Maier and J. Stein, "Indexing in an Object-Oriented DBMS," *Proc. of the Int'l Workshop on Object-Oriented Database Systems*, Pacific Grove, CA. Sept. 1986.

[Maie86b] D. Maier et al., "Development of an Object-Oriented DBMS," *Proc. of the 1st ACM OOPSLA Conf.*, Portland, OR, 1986.

[Rowe87] L. Rowe and M. Stonebraker, "The POSTGRES Data Model," *Proc. of the 13th VLDB Conf.*, Brighton, England, 1987.

[Sche86] H. Schek and M. School, "The Relational Model with Relation-Valued Attributes," *Information Sys.* 11(2), 1986.

[Schw86] P. Schwarz et al., "Extensibility in the Starburst Database System," *Proc. of the Int'l. Workshop on Object-Oriented Database Sys.*, Pacific Grove, CA, 1986.

[Sell87] T. Sellis, "Efficiently Supporting Procedures in Relational Database Systems," *Proc, of the 1987 ACM-SIGMOD Conf.*, San Francisco CA, May 1987.

[Ship81] D. Shipman, "The Functional Data Model and the Data Language DAPLEX," *ACM Trans. on Database Sys.* 6(1), Sept. 1987.

[Ston86a] M. Stonebraker, "Inclusion of New Types in Relational Database Systems," *Proc. of the 2nd Int'l. Conf. on Data Eng.*, Los Angeles, CA Feb 1986.

[Ston86b] M. Stonebraker et al., "Rule Indexing Implementations in Database Systems," *Proc. of the First Int'l Conference on Expert Database Systems*, Charleston SC, April 1986.

[Ston86c] M. Stonebraker and L. Rowe, "The Design of POSTGRES," *Proc. of the 1986 ACM-SIGMOD Conf.*, Washington DC, May 1986.

[Ston87] M. Stonebraker et al., "Extending a Database System with Procedures," *ACM Trans. on Database Sys.* 12(3), Sept. 1987.

[Yao77] S. Yao, "Approximating Block Accesses in Database Organizations," *Comm. of the ACM* 20(4), April 1977.

[Zani83] C. Zaniolo, "The Database Language GEM," *Proc. of the ACM-SIGMOD Conf.*, San Jose, CA, 1983.