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 speedup 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 can be 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.

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), type inheritance, abstract data types, procedural fields, complex objects, set-valued attributes, and so forth. A good survey for many of these constructs is presented in [Hull87].

One research area that is likely to be promising is exploring ways in which these new modeling constructs can be used to enhance database performance. In this paper we describe how reference attributes can be used in an object-oriented DBMS[1] to specify data replication. The motivation for replicating data in this case is to speedup 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 can be specified at the data model level and we also present storage-level mechanisms to efficiently support it.

The remainder of the paper describes field replication in detail. Although much of the discussion is based on the EXTRA data model of the EXODUS project [Care88], the ideas presented here can also be extended to other data models that support reference attributes or referential integrity facilities of the sort 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 gives examples of how it can be used. In Sections 4 and 5 we describe the two replication strategies: in-place replication and separate replication. Then in Section 6 an analytical cost model is presented in which the two replication strategies are compared to no replication. Finally, in Section 7 we discuss related work, and in Section 8 our conclusions are presented.

2. AN EMPLOYEE DATABASE EXAMPLE

2.1. The Example

All the examples given in this paper will 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 [Care88]. We do not expect the reader to be intimately familiar with the EXTRA data model, but we do assume that the reader is familiar with the notion of reference attributes [Zani83].

Three types have been defined in the schema: 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.

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.

[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
    (name: char[], budget: int)
define type DEPT
    (name: char[], budget: int, org: ref ORG)
define type EMP
    (name: char[], age: int, salary: int, dept: ref DEPT)
create Org:  {own ref ORG}
create Dept: {own ref DEPT}
create Emp1: {own ref EMP}
create Emp2: {own ref EMP}
```

Figure 1: The Employee Database

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. The reason for having two sets of employees will become clear later. In the EXTRA data model, the notation **own ref** indicates ownership or an existence dependency. So, for example, if Emp1 is deleted, then all the EMP objects in Emp1 would also be deleted. Note, however, that the DEPT objects referenced by Emp1 via the reference attribute 'dept' are not deleted when Emp1 is deleted.

2.2. The Physical Representation of Sets and Objects

Before continuing, we need 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 [Care89]. 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. And finally, we will assume that object identifiers (OIDs) are used to implement reference attributes.

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. Here, 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 can be replicated so that a separate I/O does not have to be performed to retrieve it.

The following example illustrates how replication is specified at the schema level:

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 will exploit it whenever possible to avoid functional joins. For example, consider the following query, which retrieves the name, salary, and department of each employee in Emp1 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 speedup in query processing is considered worth the extra space. Propagating updates presents more of a problem, particularly if there are a large number of replicated values and updates to replicated values are frequent. 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. 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 actually stored in separate objects. For now, though, it is easiest to think of the objects in 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, a replication path on Empl.dept.all 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. A replication path on Empl.dept.org.name is an example of *2-level replication* because it specifies replication on a 2-level path.

3.3.3. Collapsing N-Level Paths

Another use of field replication is in collapsing n-level paths to $n-1$ levels or less. For example, a replication path on Empl.dept.org 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 such an approach can lead to problems with referential integrity.

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 have:

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 support queries that require an associative lookup on the path Empl.dept.org.name.

Indexes on paths such as Empl.dept.org.name have been proposed before [Maie86], but they are not likely to be as efficient as using indexes on replicated data (at least for lookups). More will be said about [Maie86] in the section on related work. Indexes on replicated data [Sell87] have also been proposed before, but in a different context.

4. THE IN-PLACE REPLICATION STRATEGY

This section describes the first of our two field replication strategies: *in-place replication*. 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 a replication path is defined on Empl.dept.name, then with in-place replication, an extra field would be added to the objects in Empl for storing the replicated value dept.name. Of course, the addition of an extra field will cause the objects in Empl to undergo structural changes, but such changes are easily handled through subtyping.

4.1. Propagating Updates in In-Place Replication

With in-place replication, replicated values are kept consistent using what we refer to as *inverted paths*.[3] For example, suppose a replication path on Empl.dept.name is defined. In order to keep replicated values consistent, the inverted path Empl.dept^{-1} is created, mapping DEPT objects to the objects in Empl that reference them. Then 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 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; since the 'name' field does not play a part in the mapping of DEPT objects to Empl objects, it is dropped in the inverted path. Of course, the 'name' field 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*. *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.

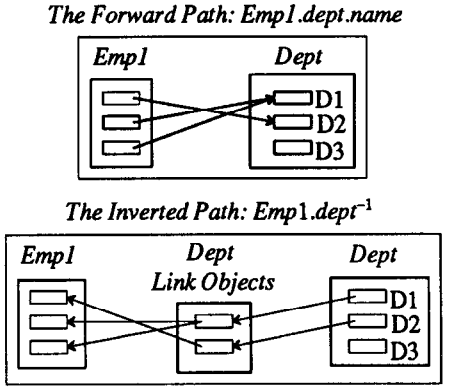

Figure 2: The Inverted Path Empl.dept^{-1}

[3] The authors of [Maie86] also examined how to implement inverted paths, although they did not refer to them as such. As noted in the section on related work, our implementation of an inverted path is quite different from the one described in [Maie86].

Before continuing, several comments about Figure 2 are in order. First, it is important to notice that the link objects have been stored in a separate set. In general, each link object can contain a large number of OIDs, 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 because only D1 and D2 are actually referenced by Empl. If D3.name is updated, that update 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 relatively 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 on Empl.dept.org.name is defined. Figure 3 illustrates what the resulting 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.

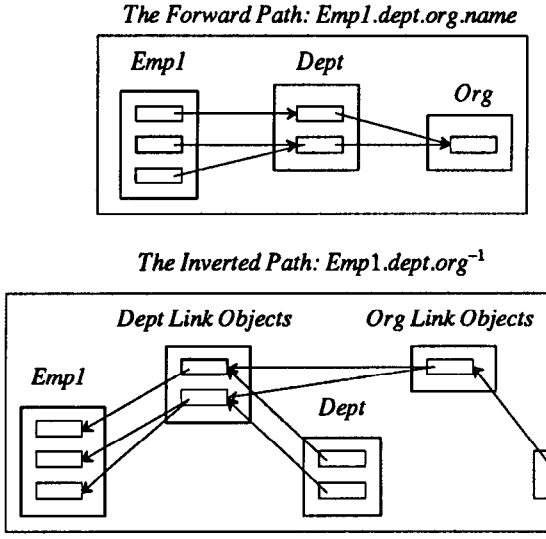

Figure 3: The Inverted Path Empl.dept.org^{-1}

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. A more thorough discussion of how this takes place can be found in [Shek89].

Although maintenance of an inverted path with more than one level can be costly, it must be remembered that replication will be generally be used on only 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, *link identifiers* (link IDs) are stored in each object O on a replication path. The link ID (or link IDs) 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 internally by the database system. A link sequence is 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). (The syntax 'link sequence = (1,2)' would not actually appear in the schema, and has only been shown here for illustrative purposes.) In the 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. (In Figure 4, 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 contains 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 know that updates to O.name need to be propagated and also how to propagate those updates (how many links to traverse, etc.). Similarly, the presence of link ID 1 in D tells us that that updates to D.org require the inverted path Empl.dept.org^{-1} to be updated.

4.1.4. Handling Multiple Paths

Unfortunately, space limitations prevent us from giving a detailed description of how multiple paths are handled (see [Shek89] for details). Basically, though, the idea is to share links in inverted paths whenever possible, as illustrated below:

replicate Empl.dept.name	link sequence = (1)
replicate Empl.dept.org.name	link sequence = (1,2)
replicate Emp2.dept.org.name	link sequence = (3,4)

The fact that the first two paths emanate 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 the link sequences of the first two paths, indicating that not only is the first link of both paths the same, but also that both paths can share the link Empl.dept^{-1}. In contrast, the third replication path does not share a common prefix with the first two paths. Consequently, it does not share links with the other paths, and this is reflected in the assignment of link IDs

4.2. Optimizing Inverted Paths

Before moving on to our next replication strategy, it is important to note that the 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. A more detailed discussion can be found in [Shek89].

One optimization is to eliminate link objects in an inverted path when they contain a small number of OIDs and instead store the OIDs directly in objects. This would shorten the length of some inverted paths and decrease the cost of propagating updates as a result.

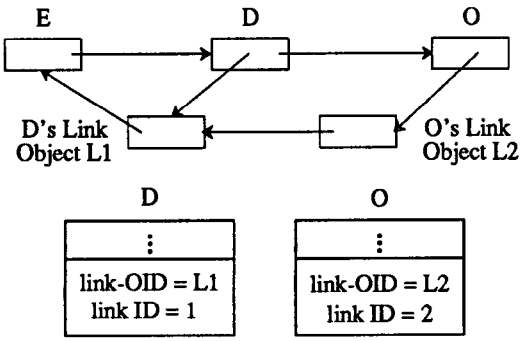

Figure 4: Determining when to Propagate an Update

Another optimization is to cluster related link objects in an n-Level path. For example, in a 2-level path, each link object in level 2 could be physically clustered with the link object(s) in level 1 that it references. This could effectively cut the cost of traversing the inverted path in half. In general, the same idea could be applied to any replication path with two or more levels.

The final optimization we consider is collapsing inverted paths with two or more levels into inverted paths with just one level. For example, in the inverted path Empl.dept.org^{-1}, the links Empl.dept^{-1} and dept.org^{-1} could be combined to form the collapsed link Empl.org^{-1}. Although this decreases the cost of propagating updates, it makes maintenance of the inverted path slightly more complicated (again, see [Shek89] for details).

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. This is because each update to a replicated field has to be propagated to every shared copy of that field. Clearly, the cost of propagating updates can become quite expensive if updates are frequent and sharing is heavy. Separate replication eliminates this problem by storing replicated values in separate objects that are shared within a set.

Figure 5 illustrates the way separate replication would work for the replication paths Empl.dept.budget and Empl.dept.name. As shown, replicated values are stored in separate objects, which are shared, and 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.

In Figure 5 it is important to notice that the objects in which replicated data is stored are kept in the same order as the corresponding objects in Dept. This is done to ensure that propagating updates generates as little I/O as possible. Also notice that the replicated values for D1 are stored together in one object, and similarly for D2.

One thing that needs to be emphasized about separate replication is that replicated values are *not* shared between sets. For example, if replication paths were defined on Empl.dept.name and Emp2.dept.name, then two sets would be generated for replicated values, one set for Empl's replicated values, and the other set for Emp2's replicated values; 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

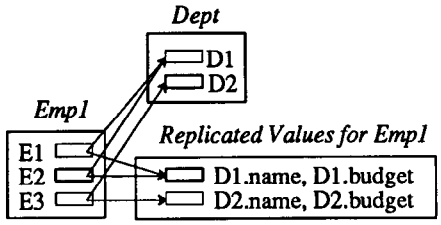

Figure 5: Separate Replication for Empl

of one set's replicated values does not interfere with the clustering of another set's replicated values.

Like in-place replication, separate replication can also be used in replication paths with two or more levels. Replicated values are stored in n-level paths the same way they are stored in 1-level paths, as illustrated in Figure 6, which will be discussed in more detail shortly.

5.1. Why Will Separate Replication Work?

At first glance, it may appear that separate replication provides no cost benefit for retrievals (at least in 1-level paths), since 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 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 paths. This follows because separate replication effectively reduces an n-level path to a 1-level path.

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 bottom of Figure 6. As shown, the objects in Org point directly to the objects containing replicated values (R1 and R2). So, for 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 and a tag of some sort to indicate which fields have been replicated in R1 so we know which updates to propagate; similarly for O2.

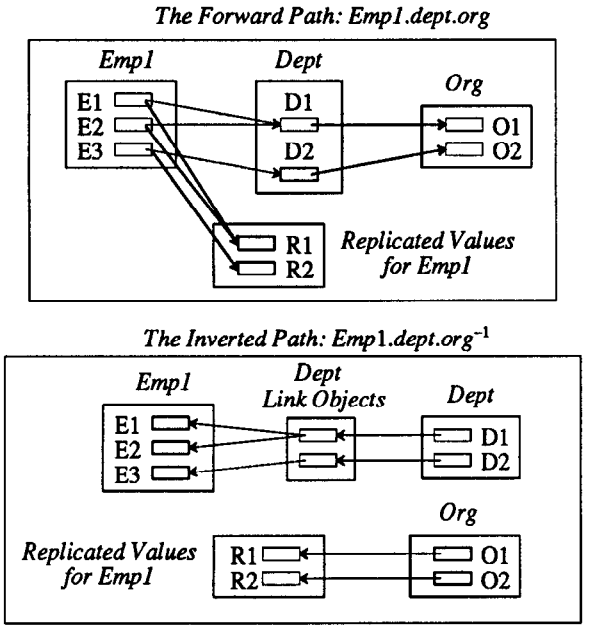

Figure 6: Inverted Paths in Separate Replication

When reference attributes such as D2.org are updated, things get slightly more complicated. As Figure 6 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. The reason why an inverted path still needs to be kept is to propagate updates to reference attributes. For example, 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} allows the necessary updates to be made.

In general, everything that was said earlier about maintaining inverted paths in the section on in-place replication, including link 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.

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. The cost model is based on the following schema:

```
define type RTYPE
    (sref: ref STYPE, ...)
define type STYPE
    (repfield: SCALR-TYPE, ...)

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 cost 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. Thus, 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, and likewise 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:

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

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 Structures in the Model

Figure 7 shows the file structures used in the cost model. Each object set is assumed to be 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, while with in-place replication, there is R, S, and the file L, which is used to store links objects for the inverted path R.sref^{-1}. With separate replication, there is R, S, and the file S', which is used to store replicated values for the path R.sref.repfield. The arcs in the figure denote the reference structure that exists between files, while the triangular shapes alongside R and S denote the B$^+$trees on field$_r$ and field$_s$, respectively.

In Figure 7, it is important to notice that the objects in L and S' are 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 that introduced in-place and separate replication.

6.2. Assumptions in the Model

To make the analysis tractable, several assumptions are required in the cost 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, so R and S' are also assumed to be relatively unclustered.) We make this assumption because we feel it represents the most typical case. 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.

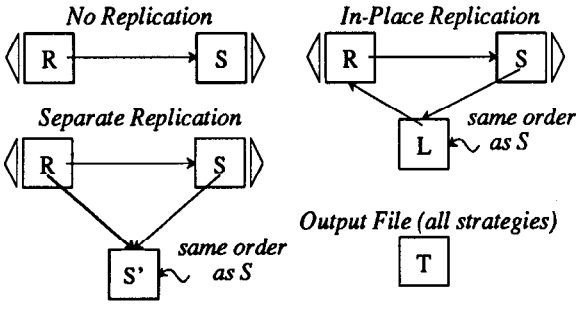

Figure 7: The File Structure with each Replication Strategy

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 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. This assumption is made because it allows us to ignore buffering and the exact details of how an efficient join algorithm might operate. Of course, for large joins, this is an overly optimistic assumption. If anything, though, 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.

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, as most queries will make use of some index.

6.3. The Parameters of the Cost Model

The parameters of the cost model are listed in Figure 8. Although there are a large number of parameters, 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, which consist of the parameters in the top half of the first table. Defaults for the core parameters are listed in the second table. To insure meaningful results, the values for B, h, $sz(type\text{-}tag)$, and $sz(OID)$ were taken from the EXODUS Storage Manager [Care86].

The meaning of most parameters should be clear from Figure 8. The parameters 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 to note about the parameter values 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 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.

6.4. Cost Analysis for Clustered Indexes

In this section, cost equations are derived under the assumption that the B$^+$ trees on field$_r$ and field$_s$ are both clustered indexes. This is where replication performs its best, because when both indexes are clustered, less overall I/O is generated, and consequently, the savings in I/O provided by replication (in joining R and S) is the largest as a percentage of the total I/O.

Parameter	Definition
B	bytes available for user data per disk page
h	overhead per object (i.e., object header)
m	B$^+$ tree fanout
$\|R\|$	number of objects in R
f	sharing level of objects in S
f_r	selectivity of read queries
f_s	selectivity of update queries
$sz(OID)$	size of OIDs
$sz(link\text{–}ID)$	size of link IDs
$sz(type\text{–}tag)$	size of type-tags
k	size of replicated field, repfield
r	size of objects in R (varies with strategy)
s	size of objects in S (varies with strategy)
t	size of objects in T
$\|S\|$	number of objects in S ($\|S\| = \|R\| / f$)
s'	size of objects in S' $s' = k + sz(type\text{–}tag)$
l	size of objects in L $l = 1 + sz(type\text{–}tag) + f \cdot sz(OID)$
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 ($O_l = \lfloor B / (h + l) \rfloor$)
O_t	objects per page in T ($O_t = \lfloor B / (h + t) \rfloor$)
P_r	pages in R ($P_r = \lceil \|R\| / O_r \rceil$)
P_s	pages in S ($P_s = \lceil \|S\| / O_s \rceil$)
$P_{s'}$	pages in S' ($P_{s'} = \lceil \|S\| / O_{s'} \rceil$)
P_l	pages in L ($P_l = \lceil \|S\| / O_l \rceil$)
P_t	pages in T ($P_t = \lceil f \|R\| / O_t \rceil$)

Defaults for Core Parameters	
B	4056 bytes
h	20 bytes
m	350
$\|R\|$	100,000
f	1 (varied)
f_r	0.001 (varied)
f_s	10 / $\|S\|$
$size(OID)$	8 bytes
$size(link\text{–}ID)$	1 byte
$size(type\text{–}tag)$	2 bytes
k	20 bytes
r	100 bytes
s	200 bytes
t	100 bytes

Figure 8: The Parameters of the Cost Model

Throughout the section, C_{read} is used to denote the net cost of a read query, while C_{update} is used to denote the net cost of an update query. In the cost equations that follow, readers will note that no distinction is made between sequential I/O and random I/O. We initially distinguished between the two, but found that it had little affect on the resulting graphs, which are in terms of percent improvement.

6.4.1. 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 of reading the index on field$_r$ is therefore:

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

The cost to read R consists of reading the $f_r P_r$ pages in R that satisfy the clause of the query:

$$C_{read\ R} = f_r P_r$$

To calculate the cost to read S, we first consider the probability that a particular page P_i in S is *not* read by a read query. Since R and S are relatively unclustered, we can assume that any particular page in S is just as likely to be read as any other page. Therefore, the probability that a 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:

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

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

$$expected\ pages\ read\ in\ S = P_s \left[1 - \frac{\binom{|R| - fO_s}{f_r |R|}}{\binom{|R|}{f_r |R|}} \right]$$

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 S is:

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

The function $y()$ has been introduced because it will used again later. 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:

$$C_{read} = C_{read\ index\ on\ R} + C_{read\ R} + C_{read\ S} + C_{generate\ T}$$
$$= \lceil \log_m |R| \rceil + \left\lceil \frac{f_r |R|}{m} - 1 \right\rceil + f_r P_r$$
$$+ P_s y(|R|, fO_s, f_r |R|) + P_t$$

6.4.2. 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 one for the index on field$_r$. Updating S consists of reading $f_s P_s$ pages in S, updating them, and then writing them back to disk. The net cost to process an update query with no replication is therefore:

$$C_{update} = C_{read\ index\ on\ S} + C_{update\ S}$$
$$= \lceil \log_m |S| \rceil + \left\lceil \frac{f_s |S|}{m} - 1 \right\rceil + 2 f_s P_s$$

6.4.3. 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:

$$C_{read} = C_{read\ index\ on\ R} + C_{read\ R} + C_{generate\ T}$$
$$= \lceil \log_m |R| \rceil + \left\lceil \frac{f_r |R|}{m} - 1 \right\rceil + f_r P_r + P_t$$

6.4.4. 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 $C_{read\ S}$ 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 read and written by an update query. Since R and S are relatively unclustered, we can assume that any particular page in R is just as likely to be updated as any other page. Based on this observation, and following the reasoning that was used to derive $C_{read\ S}$ with no replication, the cost of updating R is:

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

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

$$C_{update} = C_{read\ index\ on\ S} + C_{update\ S} + C_{read\ L} + C_{update\ R}$$
$$= \lceil \log_m |S| \rceil + \left\lceil \frac{f_s |S|}{m} - 1 \right\rceil + 2 f_s P_s + f_s P_t$$
$$+ 2 P_r y(|R|, O_r, f_s f |S|)$$

6.4.5. 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. Therefore, the cost equations for separate replication are obtained by simply substituting S' for S in the cost equations for no replication. Doing this, the net cost to process a read query with separate replication is:

$$C_{read} = C_{read\ index\ on\ R} + C_{read\ R} + C_{read\ S'} + C_{generate\ T}$$
$$= \lceil \log_m |R| \rceil + \left\lceil \frac{f_r |R|}{m} - 1 \right\rceil + f_r P_r$$
$$+ P_{s'} y(|R|, fO_{s'}, f_r |R|) + P_t$$

6.4.6. 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 no replication. 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:

$$C_{update} = C_{read\ index\ on\ S} + C_{update\ S} + C_{update\ S'}$$
$$= \lceil \log_m |S| \rceil + \left\lceil \frac{f_s |S|}{m} - 1 \right\rceil + 2 f_s P_s + 2 f_s P_{s'}$$

6.5. Performance Results for Clustered Indexes

The results for clustered indexes are presented in Figure 9. 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 each graph therefore represents no replication.

Four graphs are shown in Figure 9, each one for a different sharing level f. In all the graphs, |R| was fixed at 100,000 objects, and the value of f was set at 1, 10, 20, and 50. As shown, three lines have been drawn for both in-place and separate replication in each graph. Each line corresponds to a

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

different setting of the read query selectivity f_r. In each graph, f_r was set at 0.001, 0.002, and 0.005, and the update query selectivity f_s was fixed at $10 / |S|$. Consequently, read queries read 100, 200, or 500 objects, while update queries always updated 10 objects. The size of objects in R was fixed at 100 bytes, those in S at 200 bytes, and the replicated field at 20 bytes.

In examining the graphs, it is important to bear in mind that vertical axis is on a percentage scale, not an absolute scale. This allows all the lines to be graphed together, which greatly increases readability, but it can also be misleading if one is not careful. For example, in terms of percentages, the difference between a reduction in I/O cost of 90% and 95% may not seem that significant. In absolute terms, however, a 90% reduction corresponds to reducing the I/O cost by a factor of 10, whereas a 95% reduction corresponds to reducing the I/O cost by a factor of 20. Thus, as far as system performance goes, the difference is indeed significant.

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 in-place 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'. The size of S' is inversely proportional to the value of f, so this effect becomes more pronounced as f increases, especially for large joins (i.e., for large f_r).

Comparing the two replication strategies, the graphs show that in-place replication almost always outperforms separate replication when the probability of an update query is less than 0.10. The only exception is when $f = 50$, where in-place replication outperforms separate replication only when the probability of an update query is less than 0.05. 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 0.35.

In terms of numbers, the graphs show that the performance of in-place replication is quite sensitive to both the value of f and the probability of an update. For $f = 1$, in-place replication reduces I/O costs by 70% to 90% when the probability of an update is less than 0.50. But for $f = 10$, the probability of an update has to be less than 0.10 to achieve the same results, and for $f = 20$, it has to be less than 0.05. By the time $f = 50$, the probability of an update has to be extremely small to achieve the same results.

Compared to in-place replication, the performance of separate replication is far less sensitive to the probability of an update. This is because update queries only update 10 objects in S; and 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. Separate replication reduces I/O costs by 5% to 15% for f =1, by 25% to 60% for $f = 10$, by 40% to 65% for $f = 20$, and by 60% to 65% for $f = 50$. For all values of f, the performance of separate replication does not decrease significantly until the probability of an update exceeds 0.80.

6.6. Performance Results for Unclustered Indexes

Due to space limitations, neither the analysis nor the graphs for unclustered indexes can be presented here (see [Shek89] for the analysis). In general, though, the graphs for unclustered indexes follow the same general pattern observed in Figure 9, except that the effectiveness of replication is reduced in all cases. This is because when unclustered indexes are used, more overall I/O generated, and consequently, the savings in I/O provided by replication (in joining R and S) is smaller as a percentage of the total I/O. With unclustered indexes, the effectiveness of in-place replication is reduced by about 40% for all values of f, while the effectiveness of separate replication is reduced by about 5% for $f = 1$, by 10% for $f = 10$, by 20% for $f = 20$, and by 40% for $f = 50$.

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 out-of-date results.

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: In this strategy, cached results are kept up-to-date rather than being invalidated. 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-tuple strategies proposed for 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 quite different. Second, because field replication is more primitive than caching, it should prove both easier to implement and 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 will be 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

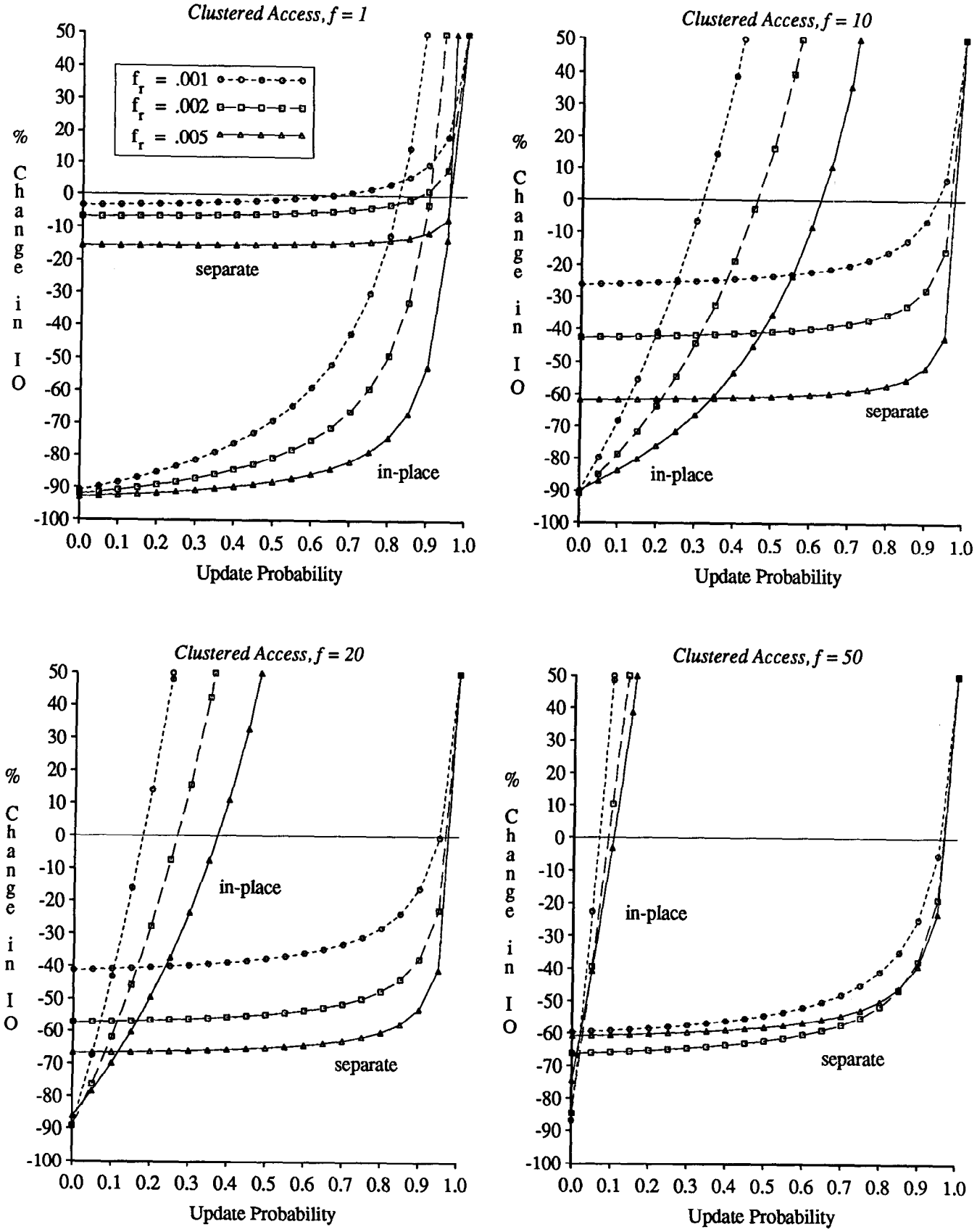

Figure 9: Results for Clustered Indexes

maintenance algorithms have to be used to keep replicated data up-to-date. Finally, query optimization should be simpler 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 based on 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 Emp1.dept.name would map department names to objects in Emp1. One of the key issues addressed by the Gemstone researchers was how to maintain an inverted path. In essence, maintaining an index on Emp1.dept.name boils down to maintaining the inverted path Emp1.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 disadvantage in using B$^+$ trees to implement inverted paths is that, assuming B +trees are typically two levels, traversing an inverted path in Gemstone requires roughly twice as much I/O. Another disadvantage is that clustering options are reduced. Since we were predominantly concerned about I/O costs and not interested in associative OID-based 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 replication strategies were discussed: in-place and separate replication. For each 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 presented for clustered indexes. In the analysis we found that in-place replication reduced I/O costs by 70% to 90% percent when the update probability and level of sharing were small, while for separate replication, I/O costs were reduced by 40% to 65% over a wide range of update probabilities and sharing levels. 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 propagated 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

[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., "Object and File Management in the EXODUS Extensible Database System," *Proc. of 1986 VLDB Conf.*, Kyoto, Japan, Aug. 1986.

[Care88] M. Carey et al., "A Data Model and Query Language for EXODUS," *Proc. of the 1988 ACM-SIGMOD Conf.*, Chicago, ILL, 1988.

[Care89] M. Carey et al, "The EXODUS Extensible DBMS Project: An Overview," in *Readings in Object-Oriented Databases*, S. Zdonik and D. Maier, eds., Morgan-Kaufman Publ. Co., 1989.

[Date87] C. Date, "*A Guide to THE SQL STANDARD*," Ch. 11, pp. 113-120, Addison-Wesley Publ. Co., 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.

[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.

[Hull87] R. Hull and R. King, "Semantic Database Modeling: Survey , 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.

[Maie86] 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.

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

[Shek89] E. Shekita and M. Carey "*Performance Enhancement Through Replication in an Object-Oriented DBMS*" Univ. of Wisconsin Tech Report #817, Jan. 1989.

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

[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.