TR 1218

Bulk Loading into an OODB: A Performance Study *

Janet L. Wiener
Jeffrey F. Naughton
Department of Computer Sciences
University of Wisconsin-Madison
1210 W. Dayton St., Madison, WI 53706
{wiener,naughton}@cs.wisc.edu

Abstract

As object-oriented databases (OODB) attract an increasingly large community of users, these users bring with them large quantities of legacy data (hundreds and thousands of megabytes, and sometimes hundreds of gigabytes). In addition, scientific OODB users continually generate new data, often tens and hundreds of megabytes at a time. All this data must be loaded into the OODB. Every relational database system has a load utility, but nearly all OODBs do not. The process of loading data into an OODB is complicated by inter-object references, or relationships, in the data. These relationships are expressed in the OODB as object identifiers, which are not known at the time the load data is generated; they may contain cycles; and there may be implicit system-maintained inverse relationships that must also be stored.

We introduce six algorithms for loading data into an OODB and present a performance study of them. The algorithms examine different techniques for dealing with circular and inverse relationships. Our performance study is based on both an analytic model and an implementation of all six algorithms on top of the Shore persistent object repository. We use the analytic model to explore a wide range of possible data and system configurations, and show that the implementation results strongly validate our analytic model. Our study demonstrates that it is important to choose a load algorithm carefully; in some cases the best algorithm achieved an improvement of one to two orders of magnitude over the naive algorithm.

1 Introduction

As object-oriented databases (OODB) attract more and more users, the problem of loading the users' data into the OODB becomes more and more important. The current methods of loading, i.e., **insert** statements in a data manipulation language, or **new** statements in a database programming language, are more appropriate for loading tens and hundreds of objects than tens and hundreds of megabytes of objects. Yet users want to load megabytes and even gigabytes of data:

- Users bring legacy data from relational and hierarchical databases (that is better suited to an OODB).
- Users with data already in an OODB sometimes need to dump and reload that data, into either the same or another OODB. The most common need for dumping and loading arises when a particular

*This work supported in part by NSF grant IRI-9157357 and by the Advanced Research Project Agency, ARPA order number 018 (formerly 8230), monitored by the U.S. Army Research Laboratory under contract DAAB07-91-C-Q518.

database must be reclustered for performance reasons. If the database uses physical OIDs, there may be no good way to recluster the objects, but if the objects are dumped to a data file in the order in which they should be clustered, it is simple to recluster them properly while reloading. Data must also be dumped and reloaded if the user is switching OODB products, or transferring a large quantity of data across a great distance, e.g., on tape.

- Scientists are starting to use OODB to store their experimental data. Scientific applications generate a large volume of data with many complex associations in the information structure [Sho93]. It is not uncommon for a single experiment to have input and output parameters that number in the hundreds and thousands, and must be loaded into the OODB for each experiment. As an example, the climate modeling project at Lawrence Livermore National Laboratory has a very complex schema and generates single data points in the range of 20 to 150 Megabytes; a single data set typically contains 1 to 20 *Gigabytes* of data [DLP+93].

Relational database systems provide a load utility to bypass the individual language statements; OODB need a similar facility. Users are currently spending too much time and effort just loading the data they want to examine. For example, Cushing reports that loading the experimental data was the most time-consuming part of analyzing a set of computational chemistry experiments [CMR92]. In addition, we know of another commercial OODB customer who currently spends 36 hours loading a single set of new data every month.

A load utility takes a description of all the data to be created, usually in text format, and loads the corresponding objects into the database. Additionally, a load utility can group certain operations, such as integrity checks, to dramatically reduce their cost for the load [Moh93a]. Although a load utility is common in relational databases, we are aware of only one OODB load utility, in Objectivity/DB [Obj92], and it is limited in that it cannot load external data that does not already contain system-specific OIDs.

Loading object-oriented data is complicated by the presence of relationships among the objects; these relationships prevent simply the use of a relational load utility for an OODB.

- In a relational database, all data stored in a tuple is either a string or a number. Tuples use foreign keys, which are part of the user data, to reference other tuples. Objects use relationships to reference each other by their object identifiers (OIDs). These OIDs are like pointers, in that they are created and maintained by the database, and are usually not visible to the user. Furthermore, these OIDs are not available at all when the load file is written, because the corresponding objects have not yet been created. Relationships must therefore be represented by some other means in the load file. We call this representation a *surrogate* identifier.

- Relationships may be circular. That is, an object A may refer to an object B which refers back to object A, either directly or via a chain of relationships. Therefore, the load utility must be able to resolve surrogate identifiers to objects that have not yet been created when the surrogate is first seen. We describe several alternative resolution mechanisms in section 2.

- Many OODB support system-maintained inverse relationships, sometimes called bidirectional relationships [Obj92, Ont92, Ver93, OHMS92, WI93]. In addition, inverse relationships are part of the ODMG standard [Cat93]. Inverse relationships are relationships that are maintained in both directions, so that an update to one direction of the relationship causes an update to the other. For example, suppose that object A has an inverse relationship with object B. Then when B's OID is stored in A, A's OID should be stored in B. The most obvious way to maintain inverse relationships — and the only way if each object is created separately — is to update each inverse object immediately after realizing that the update is needed, e.g, updating B immediately after creating A. There are two reasons why this method is not always appropriate: first, object B may not have been created yet; second, this approach leads to performance several orders of magnitude worse than is possible using a different approach.

We examine several techniques for dealing with circular and inverse relationships in our loading algorithms: pre-assigning OIDs, creating "todo" lists and sorting them to avoid random I/Os, and clearing todo lists on the fly to take advantage of objects found in the buffer pool after a todo list entry was created but before it was written to disk. We designed six algorithms that use various combinations of these techniques.

We used a two-prong approach to evaluate the performance of these algorithms. First, we built a simple analytic model to predict the performance of each algorithm. We used the analytic model to explore a wide range of load file and system configurations. We then implemented the algorithms on top of the Shore persistent object repository [CDF+94]. The implementation not only validates our analytic model, the performance of the algorithms also highlights several key advantages and disadvantages of using logical object identifiers. Furthermore, our performance results show that one algorithm almost always outperforms all the others.

We know of no other work involving loading data into an OODB. There are several published methods for mapping complex data structures to an ascii or binary file, and then reading it back in again. We based the data file format on that used by one such method called the Interface Description Language [Sno89]. Other methods include Pkl [Nel91] for Modula3 data, and Vegdahl's method for Smalltalk images [Veg86]. However, these methods do not address the problem of loading more data than can fit into virtual memory, and also ignore the performance issues that arise when the data to be loaded fits in virtual but not physical memory.

```
class Experiment
{
        attribute char scientist[16];
        relationship Input input inverse Input::expts;
        relationship Output output inverse Output::expt;
};
class Input
{
        attribute double temperature;
        attribute integer humidity;
        relationship Experiment expts inverse Experiment::input;
};
class Output
{
        attribute double plant_growth;
        relationship Experiment expt inverse Experiment::output;
};
```

Figure 1: Experiment schema definition in ODL.

Relational integrity checking is similar to the problem of updating inverse relationships. In both cases, when one tuple or object is loaded, another must be checked or updated. We discuss the similarity further in Section 2.5 when we describe techniqes for handling inverse relationships. Here we note that there are published techniques for smart handling of integrity checking [Moh93b].

The remainder of the paper is organized as follows. We present the loading algorithms in section 2. In Section 3 we describe the analytic cost model. Section 4 describes the parameters of the loading algorithms that we varied in our studies and in Section 5 we discuss the performance results obtained from the analytic model. Section 6 describes our implementation and experimental results on top of Shore. We conclude and outline our future work plan in Section 7.

2 Loading Algorithms

We present six algorithms for loading the database from data stored in a text file. In all the algorithms, we read the file and create the objects described in it. The algorithms differ in the way they handle object references (relationships) and in when they create system-maintained inverse relationships.

2.1 Example database schema

For the rest of this section, we use the following example schema, which describes the data for a simplified soil science experiment, to illustrate our algorithms. In this schema, each Experiment object has a relationship to an Input and an Output object. Figure 1 defines the schema in the Object Definition Language proposed by ODMG [Cat93].

```
Input(temperature, humidity)
{
    101: 27.2, 14;
    102: 14.8, 87;
    103: 21.5, 66;
}
Experiment(scientist, input, output)
{
    1: "Lisa", 101, 201;
    2: "Alex", 103, 202;
    3: "Alex", 101, 203;
}
Output(plant_growth)
{
    201: 2.1;
    202: 1.75;
    203: 2.0;
}
```

Figure 2: Sample data file for the experiment schema.

2.2 Data file description

The data file is an ascii text file describing the objects to be loaded[1]. We originally developed this data file format for the Moose object-oriented data model [WI93], but it is generic enough to use for any object-oriented data model, and can easily be adapted to fit the output of a particular data generating program. Furthermore, any existing data file can easily be converted by a simple script to the data file format. Such conversions will be important for loading pre-existing data, such as the data many scientists have previously kept in flat files. We illustrate the data file format in Figure 2, with a small data file for the experiment schema.

Within the data file, objects are grouped together by class, although a given class may appear more than once in the file and the order of the classes is not important. Each class is described by its name and the relationships for which values will be given. If a relationship of the class is not specified, then new objects get a null value for that relationship. Next, the objects in the class are listed. Each object begins with a surrogate identifier and a colon, continues with a comma-separated list of its values, and ends with a semicolon. In this example, all the surrogates are integers, and they are unique in the data file. However, in general, the surrogates may be strings or real numbers; if the class has a key they may even be part of the object's data [PG88]. The values for a set or list relationship are listed inside curly brackets. Strings are surrounded by

[1]Loading from binary data files would be similar. We chose to use ascii files because they are transferrable across different hardware platforms and are easy for the user to examine.

quotes, have no maximum length, and may contain any characters. Whenever one object references another object, the data file entry for the referencing object contains the surrogate for the referenced object. The process of loading the data file includes translating all the surrogates into the object identifiers (OIDs) that the database assigns to the corresponding objects. To reference objects already in the database, surrogates may be assigned to them by using queries (either before the load or inside the load data file) to individually select the objects; in this study, we do not consider references to existing objects.

2.3 Mapping surrogates to OIDs

All of the algorithms use an *id table* to map surrogate object identifiers to the database's object identifiers. Initially, the id table is empty. As each object is created, its surrogate and OID are entered into the id table. The OID can subsequently be retrieved from the id table by using its surrogate as a key. Each id table entry contains a field for the physical page on which the object resides, for use by certain algorithms as described below, as well as the surrogate and OID. Additionally, if the load utility allows surrogates that are only unique within a class, and not in the entire data file (e.g., a keyu for the class), the id table contains a field identifying the class. We used unique surrogates for this study. Table 3 shows the id table built for the Experiment data file.

Surrogate	OID	Page
101	OID1	40
102	OID2	40
103	OID3	41
1	OID4	60
2	OID5	61
3	OID6	62
201	OID7	80
202	OID8	80
203	OID9	80

Figure 3: Id table built by the load algorithms.

2.4 Creating relationships from surrogates

For each relationship from one object A to another object B, the data file contains the surrogate of B in the description of A. At some point during the load, the load utility must store the OID of B inside object A. We present three techniques for converting that surrogate to an OID and storing it in A.

The first technique we call *two-pass*, because the data file is read twice. On the first pass, the objects are created without data inside them and their surrogates and OIDs are entered into the id table. On the

second pass, we reread the data file and store the data in the objects. Since all the objects have already been created, we are guaranteed to find all surrogates in the id table.

The second technique called *resolve-early*, employs a *todo list*. The data file is read only once, and we try to resolve all the surrogates to OIDs at that time. Surrogates that refer to objects described further down in the file, however, cannot be resolved immediately. These surrogates are placed on a todo list of updates to do later. Each todo list entry contains the surrogate for the OID to store in the object, the surrogate of the object to be updated, and the offset at which to store the relationship in the object. Variants of this technique read the todo list at different times; we discuss when in the next subsection as we describe the variants. Figure 4 contains the todo list created for the example data file in Figure 2 by the resolve-early algorithms.

Surrogate for object to update	Surrogate for OID to store	Update offset
1	201	24
2	202	24
3	203	24

Figure 4: Todo list built by the resolve-early algorithms.

The third technique we call *assign-early*, and it only applies to databases that use logical OIDs. Like in *resolve-early*, in *assign-early* we try to resolve all surrogates on the first and only pass through the data file. Unlike in *resolve-early*, when we encounter a surrogate for an as-yet-uncreated object, we *pre-assign* the OID. Pre-assigning the OID involves requesting an unused logical OID from the database without creating the corresponding object on disk. This is only possible with logical OIDs. We believe that any OODB that provides logical OIDs can also provide pre-assignment of OIDs; we know it is possible at the buffer manager level in GemStone [Mai] and in Ontos, as well as in Shore.

2.5 Creating inverse relationships

Whenever we find a relationship from object A to object B that has an inverse, we know we need to store the inverse relationship, i.e., store the OID for A in object B. We now describe two methods of performing inverse updates.

In the *immediate inverse update* algorithms, we update the inverse object as soon as we discover the relationship. We note that since surrogates may refer to objects not yet created, this technique only applies to the second pass of *two-pass* algorithms.

In the *inverse todo list* algorithms, we make an entry on an *inverse todo list*. Like todo list entries in the

resolve-early algorithms, inverse todo entries contain the surrogate for the object to update, the surrogate for the OID to fill in, and an offset. The inverse todo list created for the example data file is shown in Figure 5.

Surrogate for object to update	Surrogate for OID to store	Update offset
101	1	12
201	1	8
103	2	12
202	2	8
101	3	12
203	3	8

Figure 5: Inverse todo list built by the inverse-sort algorithms.

After reading the entire data file, we process the inverse todo list. The order of the entries in the inverse todo list is unrelated to the physical order of the objects to update. To avoid a large number of random disk reads, we first sort the inverse todo list so that the order of the entries corresponds to their objects' physical order in the database[2]. Sorting is done in two phases. First, for each inverse todo list entry, we look up the page number of the object to be updated in the id table, add it to the entry, and use this page number as the sorting key. In this phase we read the inverse todo list in chunks the size of the available memory and create sorted runs of this size. In the second phase, we merge the sorted runs. On the final merge pass, we perform all the updates, touching each page of the database at most once. Usually, only one merge pass is required. Figure 6 shows the inverse todo list from Figure 5 after sorting.

Surrogate for object to update	Surrogate for OID to store	Update offset	Page of update
101	1	12	40
101	3	12	40
103	2	12	41
201	1	8	80
202	2	8	80
203	3	8	80

Figure 6: Inverse todo list after adding page numbers and sorting.

Integrity checking is very similar to processing inverse updates. Doing integrity checks during the course of the load corresponds to immediate inverse updates, and deferring integrity checking until the end of the load corresponds to building an inverse todo list and then processing it in a separate phase. For relational

[2]We predicted that without sorting the inverse todo list, the performance would be similar to that of the immediate inverse update algorithms. Since immediate inverse updates had unacceptable performance, we did not implement an unsorted inverse todo list.

integrity checking, it is known to be faster to load relations when integrity checking is deferred [Moh93a]. Duplicate integrity checks can be eliminated when processing the deferred checks, and the other integrity checks can be reordered to get better sequential I/O. DB2 has an option to defer integrity checking until the end of a load, and employs such optimizations [Moh93a].

We also note that both of our inverse update techniques ensure the integrity of the inverse relationship, and could be used for other integrity checks that are not part of an inverse relationship.

2.6 An optimization: clearing the todo lists

Both the todo and the inverse todo list are initially constructed in memory. As each list exceeds the size of memory allotted to it, that portion of the list is written out to disk. An optimization for processing both the todo list and the inverse todo list involves checking the entries on each list before writing them to disk, and clearing (removing) those entries from the list that update objects currently in the buffer pool, as these updates can be performed with no I/O cost. Note that an entry can be cleared from the todo list only if the surrogate to store in the object can now be resolved to an OID, that is, if the corresponding object has been created since the todo entry was written.

At minimum, the todo lists are cleared only when they become full and must be written out to disk. However, we try to clear each todo list periodically and attempt to clear each entry several times before writing it out to disk. In our implementation, we clear the todo lists at intervals corresponding to a one-quarter turnover of the contents of the buffer pool and we keep an old and a new todo list. At the end of each interval, we clear both the old and the new todo list and write the old list out to disk. Therefore, we attempt to clear each todo entry twice before writing it to disk.

Surrogate for object to update	Surrogate for OID to store	Update offset
201	1	8
202	2	8
101	3	12
203	3	8

Figure 7: Inverse todo list after clearing, with a 3 page buffer pool.

Figure 7 shows the inverse todo list from Figure 5 as it would look after clearing, if the buffer pool contained three pages (which is half the database). In this example, we were able to clear two entries, or one-third of the total entries, from the inverse todo list.

2.7 The algorithms

We are now ready to present the six algorithms we studied, which span all the viable combinations of resolving surrogates and handling inverse relationships.

- *Naive*: Naive is the simplest algorithm. It is a two-pass algorithm in which inverse relationships are processed with immediate inverse updates. On the first pass, it reads the data file, creates all the objects (with empty contents), and builds the id table. On the second pass, the objects are filled in with the correct data. Updates for inverse relationships are performed as they are encountered.

- *Smart-invsort*: Smart-invsort is also a two-pass algorithm. However, it uses a inverse todo list to process inverse relationships. The inverse todo list is constructed during the first pass over the data file, and then sorted before the second pass. During the second pass, the inverse todo updates are read concurrently with the data file, and each object is updated only once.

- *Res-early-invsort*: Res-early-invsort employs the resolve-early technique for surrogates and the inverse todo list technique for inverse relationships. It therefore manages both a todo list and an inverse todo list, and merges the entries from the two lists (after sorting the inverse todo list) during the update phase so that all updates to any given object are performed at once.

- *Assign-early-invsort*: Assign-early-invsort combines the assign-early technique for surrogates with the inverse todo list technique for inverse relationships. It makes one pass over the data file, sorts the inverse todo list, and then makes one pass over the database to perform the updates dictated by the inverse todo entries.

- *Res-clear-invclear*: Res-clear-invclear is similar to res-early-invsort, except that it employs the clearing optimization for both the todo and the inverse todo lists.

- *Assign-early-invclear*: Assign-early-invclear is similar to assign-early-invsort, except that it uses the clearing optimization for the inverse todo list.

3 Analytic Cost Model

The analytic model measures projected disk I/O costs. We estimated the disk I/O costs to gauge the overall performance of the algorithms because we felt that loading data is inherently I/O bound: loading primarily involves reading a data file and creating (and updating) objects in the database.

Reading the data file once and creating the database objects accounts for the minimum number of I/O's possible in a load. Except for the assign-early algorithms, each algorithm had an additional cost for resolving surrogates to OIDs, and all the algorithms had additional costs for implementing inverse relationships.

We modeled different amounts of locality of reference among the objects, which affects the cost formulas for several of the algorithms. The locality of reference expresses how likely it is that a given object is referenced by another object. We parameterize locality with x and y to say that x percent of all references are to y percent of the objects. The remaining (1-y) percent of references are to objects chosen at random. When x and y are 0, there is no locality of reference.

When an object is more likely to reference objects *near* it in the data file, we call that *nearest locality.* Nearest locality models different kinds of complex objects for a data file clustered by complex object. The y parameter says how much of the data file each complex object spans. The x parameter says how many relationships are within a given complex object, versus between complex objects. If the data file were clustered by some other criterion, or totally at random, there would be no locality. We tried modeling other kinds of locality, such as making the y% of the database be specific "hot" objects randomly scattered throughout the data file, but found that they had comparable performance to no locality and were therefore not interesting.

We now describe the cost formulas used in the analytic model. The following variables parameterize the cost of each algorithm.

Variable	Meaning
filesize	size of data file in pages
dbsize	number of database pages containing loaded objects
todolist	number of pages in todo list
invtodolist	number of pages in inverse todo list
clrtodolist	number of pages in cleared todo list
clrinvtodolist	number of pages in cleared inverse todo list
numobjs	number of objects to load
numrels	number of relationships per object
numinvrels	number of inverse relationships per object
immedupdatecost	cost of immediate inverse updates
mem	size of the buffer pool (amount of db in memory)
id table	size of the id table
x	percent of relationships to objects "nearby" in the database
y	percent of database considered nearby
z	percent of database in buffer pool at any time
probnotinmem	probability that any given page is not in buffer pool
probcleared	probability that a todo list entry can be cleared
probinvcleared	probability that an inverse todo entry can be cleared

Table 1: Parameters of the cost model.

In the following, we only consider buffer pool sizes large enough to store the id table. We used 8 byte OIDs (this is the size used by Shore), so each id table entry is 16 bytes, and the total size of the id table is $numobjs * 16$ bytes. The analytic model formulas in our C program will predict I/O cost when the id table does not fit in memory; however, we present the much simpler cost formulas for when it does. We also ignore the fact that all probabilities are bounded by 0 and 1 to simplify their presentation. To model no locality in the database, both x and y are set to 0. The cost for each algorithm is now as follows:

- $naive = 2 * filesize + 3 * dbsize + immedupdatecost.$
- $immedupdatecost = numobjs * numinvrels * probnotinmem$
- $probnotinmem = 1 - [(x * \frac{mem-idtable}{dbsize*y}) + ((1-x) * \frac{mem-idtable-(dbsize*y)}{dbsize*(1-y)})]$

  The file cost is for reading the data file twice; the database cost is for creating the database and then updating (reading and writing) all the objects, one page at a time.

  The cost for the immediate inverse updates is more complicated. The number of updates is simply the number of objects times the (average) number of inverse relationships per object. However, an I/O is only incurred when the updated object is not in the buffer pool. We calculate a probability that the object is not in the buffer pool based on the locality parameters, and use that to determine the number of I/O's incurred.

- $smart\text{-}invsort = 2 * filesize + 3 * dbsize + 4 * invtodolist$
- $invtodolist = numobjs * numinvrels$

  The invtodolist cost involves writing the inverse todo list out to disk, reading it back in and writing out sorted runs, and then reading and merging the runs to produce the sorted list. If the sort required an extra merge pass, the cost would be $6 * invtodolist$. Although this variable cost is built into the analytic model, we found that in our tests, all the sorted runs could be merged concurrently with the database updates.

  The size of the inverse todo list is bounded by the number of inverse relationships per object. Since all inverse relationships are entered onto the inverse todo list when they are discovered, the size of the inverse todo list is thus the same as its upper bound.

- $resolve\text{-}early\text{-}invsort = filesize + 3 * dbsize + 2 * todolist + 4 * invtodolist$
- $todolist = numobjs * numrels * 0.5$

Resolve-early-invsort reads the data file only once. However, it incurs the cost of writing and reading both a todo list and a inverse todo list. The inverse todo list cost is the same as for smart-invsort. The size of the todo list is bounded by the number of objects times the number of relationships per object. However, on average, half of the references from each object will be to objects described previously in the data file, and half to objects described further on. We therefore model the size of the todo list as one-half the possible number of entries.

- $Assign\text{-}early\text{-}invsort = filesize + 3 * dbsize + 4 * invtodolist$

  Assign-early-invsort does not use a todo list, since it pre-assigns OIDs whenever an unresolved surrogate appears. The inverse todo list cost is the same as for smart-invsort.

- $Resolve\text{-}clear\text{-}invclear = filesize + 3 * dbsize + 2 * clrtodolist + 4 * clrinvtodolist$

- $clrtodolist = numobjs * numrels * probcleared$

- $probcleared = [(x * \frac{y-z}{y}) + ((1-x) * (\frac{1-z}{1-y}))] * 0.5$

- $z = \frac{mem-idtable}{dbsize}$

- $clrinvtodolist = numobjs * numinvrels * probinvcleared$

- $probinvcleared = (x * \frac{y-z}{y}) + ((1-x) * (\frac{1-z}{1-y}))$

- $Assign\text{-}early\text{-}invclear = filesize + 3 * dbsize + 4 * clrinvtodolist$

  The costs for the inverse-clear algorithms are superficially the same as for their inverse-sort counterparts. The difference lies in the size of the todo and inverse todo lists. Since some of the todo list entries are removed when the todo list is cleared, the cleared todo list and cleared inverse todo list are significantly smaller than their non-cleared counterparts.

  When the entire database fits in the buffer pool, the sizes of the todo list and the inverse todo list drop to zero, since all entries will be cleared. At the other extreme, when the buffer pool holds only the id table, no entries are cleared. In between, the percentage of the database in the buffer pool is used in conjunction with the locality to determine how many entries can be cleared. Since each entry will be checked for clearing shortly after it is created, the probability of clearing the entry is much greater if the object being referenced (in the case of the todo list) or the object to be updated (in the case of the inverse todo list) is physically nearby the object that generated the todo or inverse todo entry in the database, and therefore in the buffer pool at the same time. We model writing each todo list entry

out to disk at the same time as the object that generated that entry is flushed from the buffer pool. Hence, the formulas for clearing the todo and inverse todo lists are very similar.

We note that only the algorithms that try to update objects in a random order are affected by the locality of reference. For this purpose, random means any order that is not the same as the data file order. Thus, naive, res-clear-invclear and assign-early-invclear are affected by locality, and by the size of the buffer pool, while smart-invsort, res-early-invsort, and assign-early-invsort are not.

We also note that the I/O cost of naive is quadratic in the number of objects and the number of inverse relationships. For all the other algorithms, the cost is linear in the number of objects when the id table fits in memory. (When the id table does not fit, the cost is quadratic in the number of objects and the number of relationships.)

4 Data file configuration and system parameters

For most of the analytical and implementation experiments, we used 200 byte objects. Each 200 byte object had 10 slots for relationships, and 10 slots for inverse relationships to it. Additionally, each object had a 40 byte string field. We varied the number of objects to control the size of the database. The 5 Megabyte database has 25,000 objects; the 20 Megabyte database has 100,000 objects. We varied the size of the buffer pool available for each data file, to gauge the effect that more buffer pool had on the algorithms' performance.

We varied the number of relationships and inverse relationships from each object and found no significant difference in the relative performance of the algorithms. We therefore present the experiments for 10 relationships and 0 and 10 inverse relationships. Additionally, since in some cases the user will have a choice whether to use inverse relationships in the load file, or to put both directions of the relationship explicitly in the data file, we present results for 20 relationships and no inverse relationships.

We varied the locality of reference from no locality to having 90% of references stay within the nearest 10% of the database. In the implementation experiments, the locality was built into the actual references in the data file. In the analytic experiments, it was a parameter.

5 Analytic model results

For each of the first set of experiments with the analytic model, we choose a data file configuration and then varied the amount of memory available for the load. In Figures 8 and 9, we show the predicted number of I/O's to load a 5 Mb database with nearest 90-10 locality. (We use 90-10 as an abbreviation for x = 90% and y = 10% from here on.) We varied the memory available by 1 Mb at a time, from 1 Mb to 15 Mb. At

15 Mb, the entire database plus all auxiliary data structures, such as the inverse todo list, fit in memory. No further performance gain is possible by adding more memory.

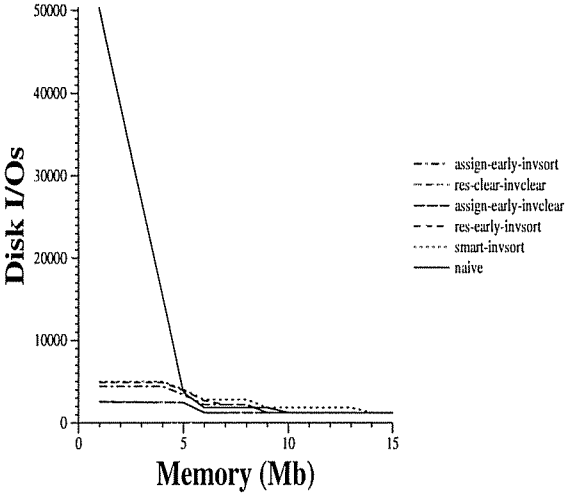

Figure 8: 5 Mb database with 90-10 locality.

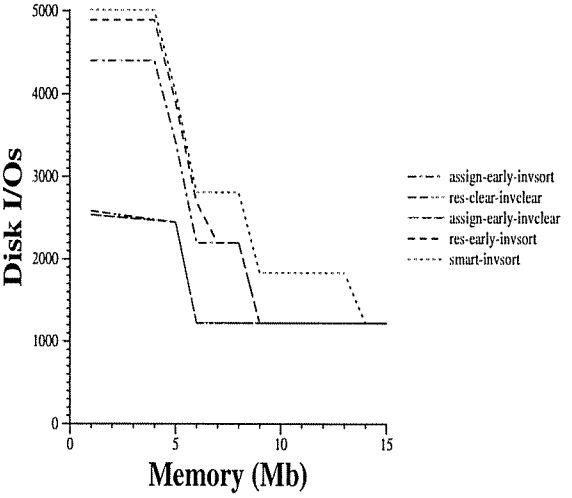

Figure 9: 5 Mb database with 90-10 locality, without naive

Figure 8 illustrates how much worse the naive algorithm performs relative to the others until the entire database fits in memory; when the buffer pool holds only 20% of the database, naive performs a full order of magnitude worse. Figure 9 shows the differences in performance among the remaining algorithms. Until the database and auxiliary data structures, around 8 Mb, fit in the buffer pool, the clearing algorithms decidedly outperform the non-clearing algorithms. This is due to their writing and reading much smaller versions of the todo list and inverse todo list. When both a todo list and an inverse todo list are needed, res-clear-invclear is able to perform as well as assign-early-invclear because the updates dictated by both lists are merged in the same pass over the database. Although res-clear-invclear has the additional cost of writing and reading a todo list, that cost is minimal compared to the cost of updating the entire database. However, that cost does show up in a slight performance difference between assign-early-invsort and res-early-invsort, whose todo lists are somewhat larger. Smart-invsort performs comparably to res-early-invsort. Although smart-invsort does not create a todo list, it incurs approximately the same number of I/O's because it reads the data file a second time.

When there is no locality of reference among the objects, the relative performance of the algorithms remains the same, as shown in Figure 10. However, while the non-clearing algorithms are unaffected by the locality, the clearing algorithms perform significantly worse, because fewer of the todo list entries and inverse todo list entries update objects that are in the buffer pool around the same time as the entry is generated. We do not show naive's performance in this graph because it is so much worse that the other algorithms appear as a single line on the graph. Relative to the other algorithms, naive now performs two orders of

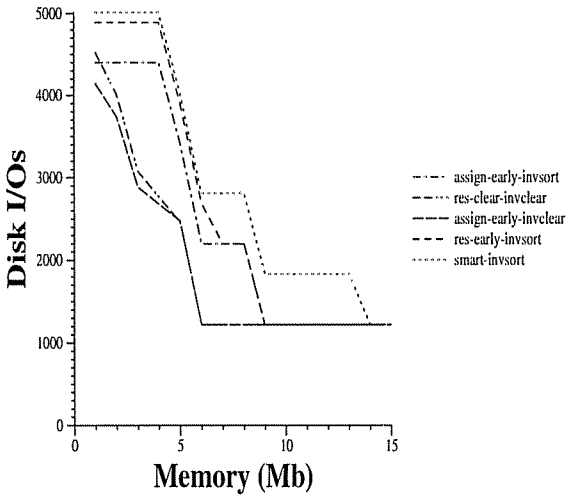

Figure 10: 5 Mb database with no locality.

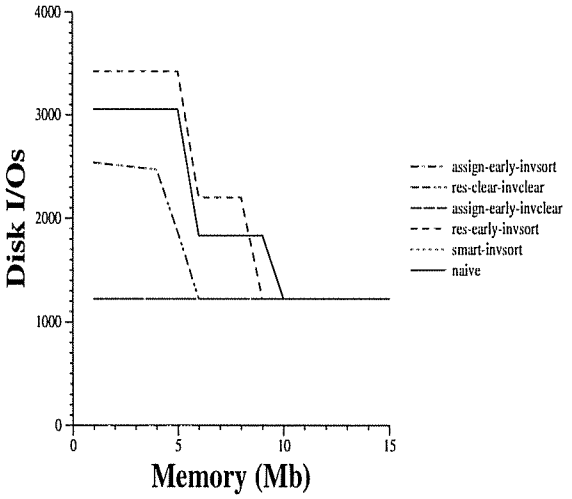

Figure 11: 5 Mb database with 20 relationships and 0 inverses.

magnitude worse! With 3 Mb of available memory, naive requires 218,000 I/O's, while smart-invsort needs only 10,000 and res-clear-invclear performs merely 3,100.

In some cases, such as when an OODB is dumped to a file and then reloaded, it is possible to dump both halves of an inverse relationship. That is, instead of storing only the fact that A has an inverse relationship with B in the data file, and letting the load algorithm take care of storing the relationship from B to A, it is possible to indicate both the relationship from A to B and the relationship from B to A explicitly in the data file. That way, the load algorithm does not need to perform any inverse updates. Also, in some schemas, there are no inverse relationships. We therefore decided to test the algorithms' performance for a data file containing twice as many relationships, to represent both halves of an inverse relationship but no implicit inverse relationships. We show this experiment in Figure 11. For all the algorithms, the performance was improved two-to-fourfold. The assign-early algorithms achieved the best performance possible: since they resolve all surrogates to OIDs on the first pass over the database, they did not need an second (update) pass over the database. Naive and smart-invsort appear as a single line, since they differ only in their handling of inverse updates. Res-clear-invclear performs slightly better than smart-invsort because the cost of writing and reading the cleared todo list is less than that of rereading the data file; res-early-invsort performs slightly worse for the opposite reason.

In the next experiment, we varied both the database size and the buffer pool size, while holding the percentage of the database that fit in the buffer pool constant. Figure 12 shows the results of this experiment, where we scale the database from 5 Mb up to 1 Gb, keeping the buffer pool size equal to 10% of the database. We chose 10% since it is the most practical for loading massive amounts of data. All the other parameters are the same as in the experiment in Figure 8. We verify with this experiment that the relative performance

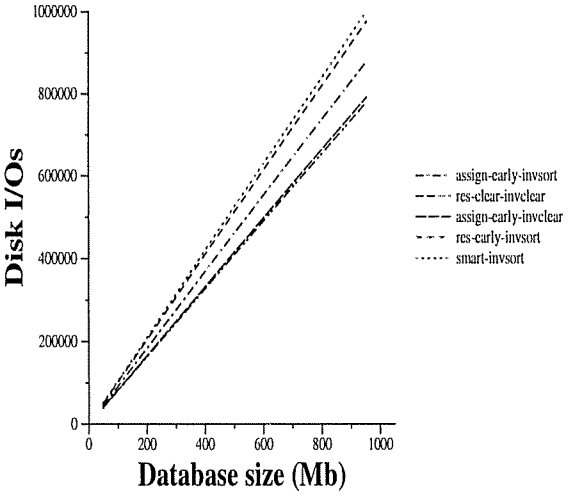

Figure 12: Scaling size of database to 1 Gb, with buffer pool holding 10% of database.

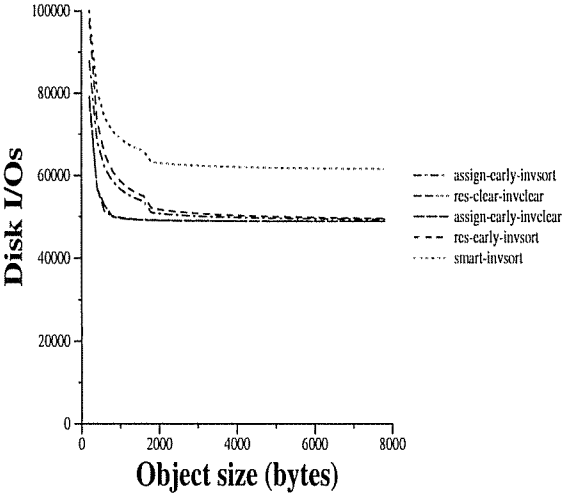

Figure 13: Scaling object size from 200 bytes to 8 Kb.

of the algorithms does not change as we scale the database, and that with a corresponding increase in the buffer pool, the increase in I/O cost for the algorithms (except naive) is linear.

For the final experiment, we held the the database size constant and varied the object size from 200 bytes to 8 Kb, the size of a Shore page. To keep the database size constant, we decreased the number of objects as we increased the objects' size. For this test, we used a 100 Mb database with a 10 Mb buffer pool. Although the relative performance of the algorithms does not change, as the objects get larger the individual performance of each algorithm improves. There are two reasons why the corresponding decline in object size causes the improved performance: first, the id table shrinks and so more of the database fits in the buffer pool. Second, the absolute number of relationships declines, and so the size of the todo and inverse todo lists also declines.

In the above graphs, we omit costs for the algorithms when the id table does not fit in memory, because they make all the algorithms look like a single line when the id table does fit. As we mentioned when we discussed the cost model, the number of I/Os for the algorithms goes from being linear in the number of objects to quadratic in the number of relationships. For example, the predicted cost for smart-invsort for a 5 Mb database is only 5,000 I/Os with 0.5 Mb of memory, which just barely holds the id table, but 363,000 I/Os with 0.1 Mb of memory. All of the algorithms exhibit similar one-hundred-fold increases in cost.

5.1 Discussion

According to the analytic model, the relative ranking of the algorithms is assign-early-invclear, followed closely by res-clear-invclear, followed less closely by assign-early-invsort and then res-early-invsort and smart-invsort, and this ranking is fairly consistent regardless of the locality in the data file or the number of objects

or relationships. Naive, on the other hand, performs very poorly in the presence of inverse relationships, unless the entire database fits in memory. At that point, it doesn't really matter which algorithm is used.

The more locality of reference is present in the data file, the better the clearing algorithms will perform. However, the difference in performance is not significant enough to warrant sorting the data file before the load unless (1) very little memory is available for the load; (2) the sort can be done with only one extra read and write of the data file, e.g., with a technique such as we use for sorting the inverse todo list and where the final merge pass is directly piped into the load program as the data file; (3) a clearing algorithm is used for the load; and (4) the locality after sorting is fairly high, e.g., at least 90-10. Naive is similarly influenced by the locality, but the other non-clearing algorithms are unaffected until a given data structure, such as the inverse todo list or the database, first fits in memory. Then they exhibit a sharp decrease in cost and remain at that cost until the next data structure also fits.

The resolve-early and assign-early algorithms have the added benefit that since they only read the data file once, they can read the data file from a pipe. Therefore, if the program generating the data produces it in the data file format, the data file need never be physically stored. This can be very important when disk space is tight, because the data file tends to be around the same size as the database it describes.

All the algorithms cost significantly less when there are no inverse relationships. However, we have already noted that most commercial OODB systems (Ontos, Objectivity, Versant, ObjectStore) today support inverse relationships and sometimes it is not feasible to generate both halves of the relationship for the data file. For example, a dumped relational database would have foreign keys in one relation for one-half of the relationship, but the other relation would most likely store nothing that references the first relation. In a relational system, this would still allow either relation to be the "inner" relation in a join, but in an OODB the join order of a functional join (a join that traverses a relationship) can only be switched if the relationship has an inverse. In addition, explicitly storing twice as many relationships in the data file can substantially increase the size of the data file and may not be a viable option when disk space is at a premium. Furthermore, when the load utility handles inverse relationships, it also handles all the referential integrity checks for the inverse relationships. The cost of doing first a load, and then referential integrity checks, would be much higher than doing the checks as part of the load. If the data to be stored contains no relationships at all, this study does not apply.

Although we do not present the results for loads when the id table does not fit in the buffer pool, we note that the I/O cost greatly increases: we do an insert in the id table for each object, and a lookup for each relationship and inverse relationship. When each of these inserts and lookups causes a I/O for the correct id table page, the cost skyrockets to the same magnitude as the naive algorithm, for all algorithms. Therefore,

we recommend enough memory to store the id table as the minimum amount of memory that should be made available to the load. This limitation does not absolutely constrain the amount of data that can be loaded at one time, but rather the number of objects that may be loaded: a data file containing 1 Gb of 8 Kb objects builds an id table of only 2 Mb.

6 Implementation

We ran all six loading algorithms on a Hewlett-Packard 720 with 32 Mbytes of physical memory. However, we were only able to use about 12 Mb for any test run, due to operating system and daemon memory requirements. The database was stored under the Shore storage manager [CDF+94] on a raw Maxtor LXT-213SY disk controlled exclusively by Shore. The data file resided on a separate disk on the local file system, and thus did not interfere with the database I/O. For these tests, we turned logging off. It is important to be able to turn off logging when loading a lot of new data [Moh93a]; we found that when we used full logging, the log outgrew the database. It is unlikely that many users have enough extra disk space to accomodate such a log.

We used Shore as the underlying persistent object manager even though Shore is still under development. We decided to use Shore instead of another storage manager, such as Exodus, for two reasons. First, Shore provides the notion of a "value-added server" (VAS), which allowed us to add the load utility directly into the server. We feel that this is the best place for a load utility; the load algorithms have direct access to the server buffer pools and can determine what is in the buffer pool at any given time. This was especially important for the algorithms that try to clear the todo list and inverse todo list. The implementors of DB2 similarly experienced significantly better performance when the load utility interacted directly with the buffer manager, instead of as a client [Moh93b]. However, there is nothing in the algorithms that prevents them from being implemented at the client level.

Second, Shore provides logical OIDs, which we needed to test the assign-early algorithms. Exodus provides physical OIDs. To translate the logical OIDs into physical addresses on disk, Shore uses a logical OID index that maps from logical OIDs to physical addresses. The logical OID index is stored in the database.

Shore stores 1:N relationships entirely inside the objects. Higher levels of Shore use variable-size objects to store exactly the N OIDs in the containing object. To keep all objects the same size in our tests, we used fixed size objects and overwrote previously stored elements of a relationship if N became greater than the number of slots allotted. This allowed us to accurately measure the cost of the updates, since all updates are performed, without varying the object size.

We stored the todo list as a single large object, and the inverse todo list as several large objects. The id table is implemented as a open addressing hash table, hashed on the surrogate. Our code for all the load algorithms combined was about 5000 lines of C++ code, and took only one month to write.

6.1 Experimental Results

We ran experiments to load a database with 5, 20, and 50 Mb of data, using the same data file configurations as in the analytic model. In each case, all the objects were 200 bytes and we increased the number of objects to increase the database size. Due to metadata overhead and Shore's logical OID index, the databases created were actually 8, 32, and 80 Mb. Also, we implemented the id table as a transient data structure for simplicity. The memory size on each graph reflects the sum of the heap memory and the buffer pool size, since in the analytic model we did not distinguish between the two. In multiple runs of the same data point, we found less than 1% variance in the results; we therefore include one set of actual data points in the following graphs rather than averages.

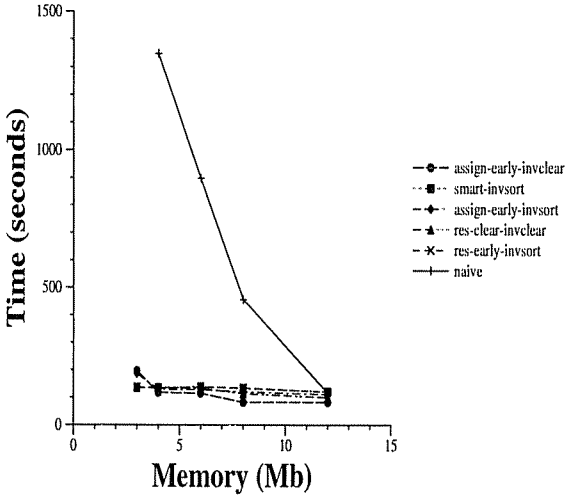

Figure 14: 5 Mb database with 90-10 locality.

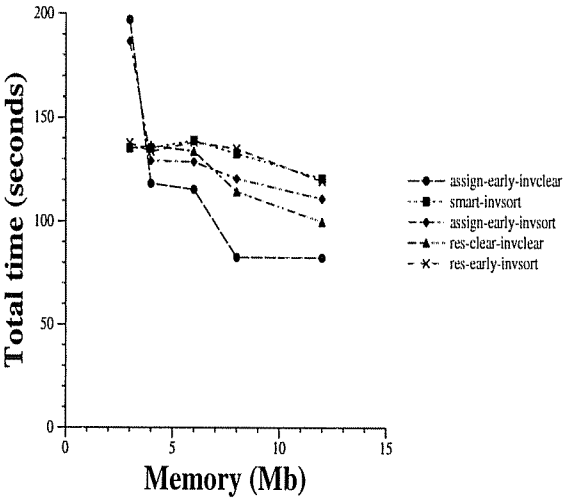

Figure 15: 5 Mb database with 90-10 locality, without naive

For the first set of experiments, we created a database with 5 Mb of data, and allocated 2 Mb to the heap for the 0.5 Mb id table and the todo and inverse todo lists. The buffer pool size is thus 2 Mb less than the total memory shown in the graphs. Moreover, since the created database was actually 8 Mb, the entire database first fits in the buffer pool when the total allocated memory is 10 Mb.

In the first experiment, shown in Figure 14, we loaded a data file for a 5 Mb database with 90-10 locality. As predicted by the analytic model, the times for the naive algorithm dominate by an order of magnitude. We therefore present the results again without the naive algorithm in Figure 15. The far left points demonstrate what we consider a small buffer pool. In this case, the logical OID index does not fit entirely in the buffer

pool. This had only a small effect on the performance of the smart-invsort and resolve-early algorithms, but a dramatic effect on the assign-early algorithms. The smart-invsort and res-early algorithms assign OIDs to objects as the objects are created, and hence the OIDs are inserted into the logical OID index in clustered order. The assign-early algorithms, in direct contrast, assign OIDs to objects as the objects' surrogates are encountered. As the objects are created, their OIDs are entered in the logical OID index in a random order (i.e., not clustered by OID). Since the logical OID index did not fit in the buffer pool, each object creation caused (on average) an extra disk I/O to insert the OID into the index.

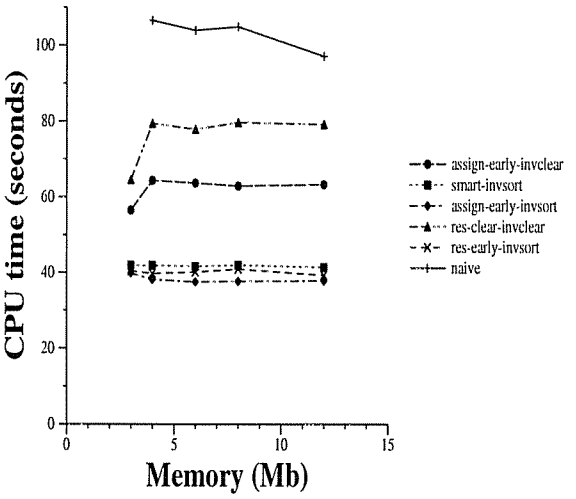

Figure 16: 5 Mb database: CPU time

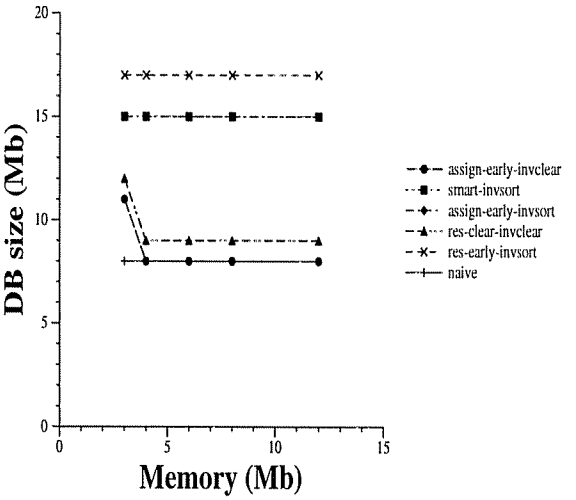

Figure 17: 5 Mb database: Actual DB size, including auxiliary data structures

As the buffer pool grows to hold nearly the entire database, we see the most improvement in performance by the algorithms that take advantage of the contents of the buffer pool, namely, the clearing algorithms, assign-early-invclear and res-clear-invclear. However, the improvement is not as dramatic as the analytic model predicts. This difference is explained by the relative CPU costs of the algorithms, shown in Figure 16. The clearing algorithms perform significantly more work to check the buffer pool for each entry on the todo and inverse todo list. In addition, while clearing an entry has no associated I/O cost, there is a fair amount of overhead involved in pinning the corresponding object in the buffer pool and updating it. The clearing algorithms pin the object for each "free" update. The updates done in the second phase, however, only pin each object once, no matter how many updates to a given object there are.

Figure 17 shows the size of the database, including the auxiliary data structures (the todo list and inverse todo list) created by each algorithm. Naive generates the smallest database because it has no auxiliary structures. For the 5 Mb database, the logical OID index accounts for approximately 2.5 Mb of the 8 Mb stored. Like the size of the id table, however, the size of the logical OID index corresponds to the number of objects, rather than the absolute size of the database.

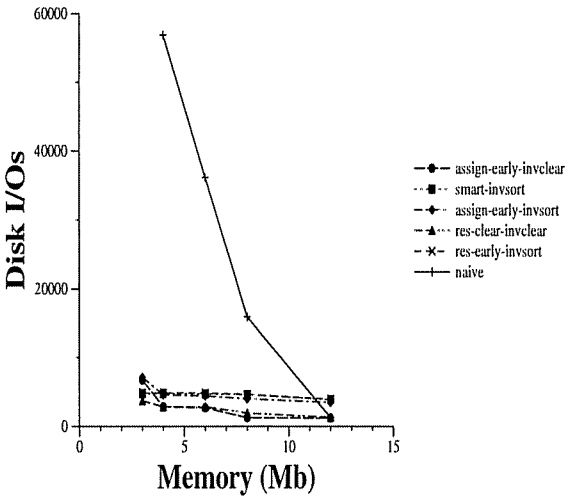

Figure 18: 5 Mb database: Disk I/O

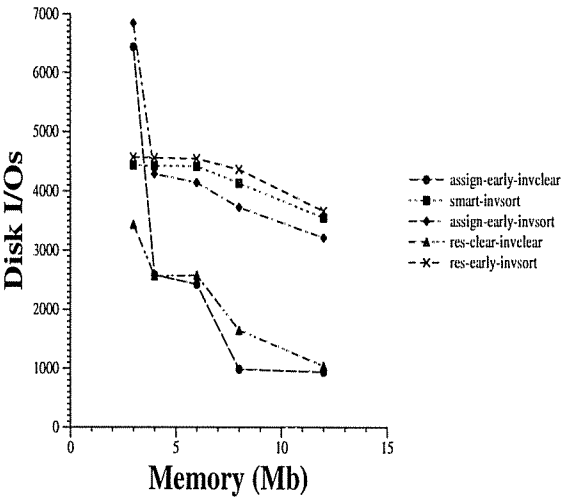

Figure 19: 5 Mb database: Disk I/O, without naive

In Figure 18 we show the I/O cost of each algorithm; in Figure 19 we repeat the results without the naive algorithm. Except for the anomalies in the assign-early algorithms with a small buffer pool, due to the logical OID index, we note that the actual I/O cost of each algorithm is extremely close to the I/O cost predicted by our analytic model. For example, we predicted 5011 I/Os for smart-revsort with 4 Mb memory. In our experiment, smart-revsort took 4802 I/Os, which is less than a 5% deviation.

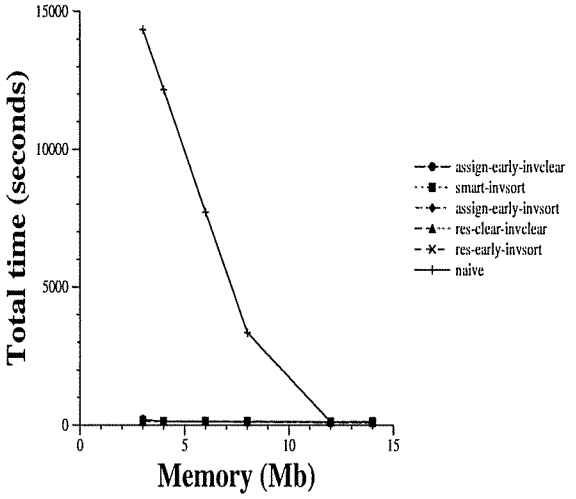

Figure 20: 5 Mb database with no locality

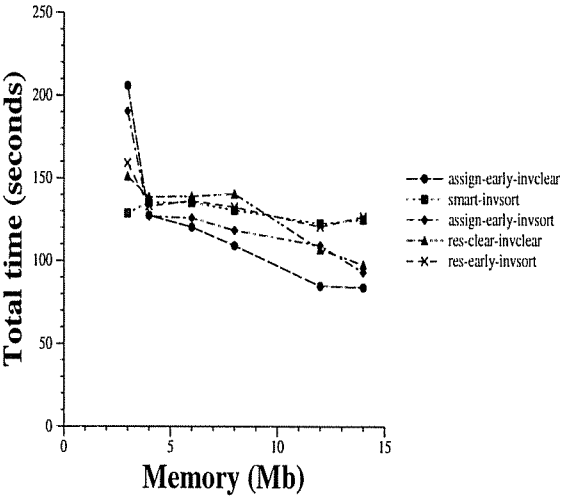

Figure 21: 5 Mb database with no locality, without naive

We next experimented with a 5 Mb data file with no locality of reference. As we predicted in the analytic model, naive becomes an even worse choice, taking 4 hours to complete the load with 3 Mb of memory, and 2 hours with 6 Mb. All the other algorithms, in contrast, take 2 to 3 minutes. The relative performance of the algorithms is similar to that with 90-10 locality, but the gap between the clearing and non-clearing

algorithms narrows substantially. In fact, res-clear-invclear, which was the second best algorithm with 90-10 locality, now lags behind the other algorithms until the entire database fits in memory. This is due to the high CPU cost of clearing, which has no benefit when there is a small buffer pool and most of the updates are to objects far from the object generating the todo or inverse todo entry.

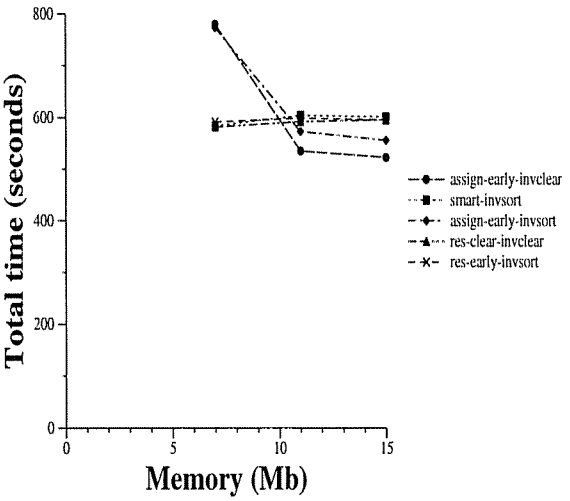

Figure 22: 20 Mb database with 90-10 locality

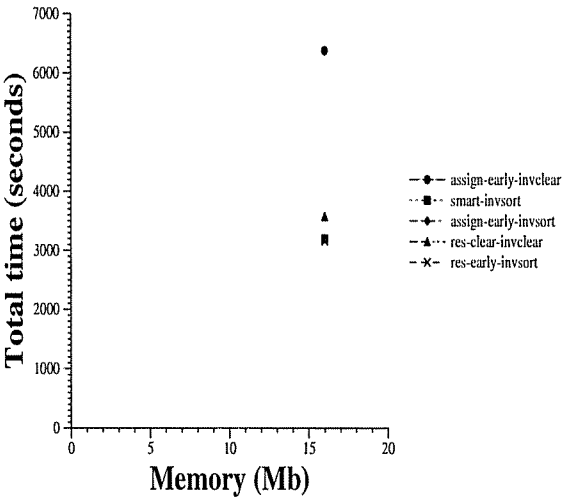

Figure 23: 50 Mb database with 90-10 locality

Figures 22 and 23 show the results of loading 20 Mb and 50 Mb of data, respectively, with nearest 90-10 in the data file. We present these graphs primarily to show that the performance of the algorithms scales as we increase the amount of data to load. Note, however, that because of the 16 Mb physical limitations on combined buffer pool and heap memory size for the load process, we could test only a small and medium buffer pool for 20 Mb, and only a small buffer pool for 50 Mb.

Although we do not present the graphs, when we ran experiments with 20 relationships but no inverse relationships, we found that the analytic model was correct: all the algorithms run much faster, and the assign-early algorithms are by far the fastest because they do not need an update pass over the created database. For example, assign-early-invsort loaded the 5 Mb database with 4 Mb of available memory in 43 seconds, while res-early-invsort took 100 seconds and naive and smart-invsort both took 93. The fastest time to load the same database with inverse relationships was 135 seconds.

6.2 Discussion

The implementation results confirm that the analytic model predicts both the relative and the actual performance of the algorithms fairly accurately. Although the analytic model predicted that assign-early-invclear would be the best algorithm, it did not account for the logical OID index. The index imposed substantial I/O overhead for assign-early when it did not fit in the buffer pool, and in those cases, neither assign-early

algorithm is a winner. When the index does fit in the buffer pool, assign-early-invclear is the algorithm of choice, as predicted.

Of the other algorithms, both smart-invsort and res-early-invsort are good choices for a small buffer pool. Neither is affected much by the logical OID index, and neither wastes CPU time trying to clear entries on the todo lists when there are very few objects in the buffer pool that could be updated.

If pre-assignment of OIDs were not provided by the OODB, resolve-clear-invclear was the best contender for medium size buffer pools when there is locality in the data file, and for large size buffer pools all the time. However, for small buffer pools, smart-invsort and res-early-invsort are better choices.

We recommend that two algorithms be implemented in any OODB: smart-invsort to be used for small buffer pool sizes, and either assign-early-invclear (if logical OIDs are supported) or res-clear-invclear (otherwise) for the rest of the time. The load procedure should be able to take hints about the contents of the data file, such as the number and average size of objects to create, and whether there are inverse relationships. (The actual buffer pool size will be fixed at run-time; whether it is small or large depends on the size of the database to create.) These hints can then be used to choose the appropriate algorithm.

7 Conclusions

A bulk loading utility is critical to users of OODBs with significant amounts of data. These users include those switching from a relational or hierarchical database; those switching OODB products; those who want to recluster their OODB data for better performance; and scientists running applications that continually generate vast amounts of new data. However, loading in an OODB may be very slow due to relationships among the objects; inverse relationships exacerbate the problem. In our performance study we showed that the best algorithms mitigate the problems due to relationships by (1) using sorted todo lists to avoid random reads and updates, (2) using pre-allocation of OIDs to avoid making entries on the todo lists in the first place, and (3) using a "clearing" technique to avoid writing todo list entries out to disk.

Our future work includes running experiments that load 1 Gb of data (when we get a larger disk to store the database); investigating algorithms for a loading in parallel on one or more servers with multiple database volumes; looking into reclustering algorithms that dump and then reload objects; and adding smart integrity checking to the load algorithms. Integrity checking is similar to doing inverse relationship updates, but only involves a read instead of an update. Also, arbitrary integrity constraints may involve more than a single relationship. In addition, we would like to integrate the load implementation with the higher levels of Shore and turn it into a utility to be distributed with Shore.

8 Acknowledgements

We would like to thank several people who made this paper possible. David Maier provided the original inspiration to study loading and feedback on our early ideas for loading algorithms. Mike Zwilling provided advice and support for our implementation as a Shore value-added server. Mark McAuliffe supplied copious comments which improved many aspects of this paper.

References

[Cat93] R. G. G. Cattell, editor. *The Object Database Standard: ODMG-93.* Morgan-Kaufman, Inc., San Mateo, CA, 1993.

[CDF+94] M. Carey, D. DeWitt, M. Franklin, N. Hall, M. McAuliffe, J. Naughton, D. Schuh, M. Solomon, C. Tan, O. Tsatalos, S. White, and M. Zwilling. Shoring Up Persistent Applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1994. To appear.

[CMR92] J. B. Cushing, D. Maier, and M. Rao. Computational Proxies: Modeling Scientific Applications in Object Databases. Technical Report 92-020, Oregon Graduate Institute, December 1992. Revised May, 1993.

[DLP+93] R. Drach, S. Louis, G. Potter, G. Richmond, D. Rotem, H. Samet, A. Segev, and A. Shoshani. Optimizing Mass Storage Organization and Access for Multi-Dimensional Scientific Data. In *Proceedings of the IEEE Symposium on Mass Storage Systems*, Monterey, CA, April 1993.

[Mai] David Maier. Private conversation, January 27, 1994.

[Moh93a] C. Mohan. A Survey of DBMS Research Issues in Supporting Very Large Tables. In *Proceedings of the International Conference on Foundations of Data Organization and Algorithms*, Evanston, Il., 1993.

[Moh93b] C. Mohan. IBM's Relational DBMS Products: Features and Technologies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 445–448, 1993.

[Nel91] G. Nelson, editor. *Systems Programming with Modula-3.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[Obj92] Objectivity, Inc. *Objectivity/DB Documentation*, 2.0 edition, September 1992.

[OHMS92] J. Orenstein, S. Haradhvala, B. Margulies, and D. Sakahara. Query Processing in the ObjectStore Database System. In M. Stonebreaker, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 403–412, 1992.

[Ont92] Ontos, Inc. *Ontos DB Reference Manual*, release 2.2 edition, February 1992.

[PG88] N. W. Paton and P. M. D. Gray. Identification of Database Objects by Key. In K. R. Dittrich, editor, *Advances in Object-Oriented Database Systems: 2nd International Workshop on Object-Oriented Database Systems*, pages 280–285, Berlin, Germany, September 1988. Springer-Verlag.

[Sho93] A. Shoshani. A Layered Apporach to Scientific Data Management at Lawrence Berkeley Laboratory. *IEEE Data Engineering Bulletin*, 16(1):4–8, March 1993.

[Sno89] R. Snodgrass. *The Interface Description Language: Definition and Use.* Computer Science Press, 1989.

[Veg86] S. R. Vegdahl. Moving Structures between Smalltalk Images. In *Proceedings the International Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 466–471, 1986.

[Ver93] Versant Object Technology. *Versant Object Database Management System C++ Versant Manual*, release 2 edition, July 1993.

[WI93] J. L. Wiener and Y. Ioannidis. A Moose and a Fox Can Aid Scientists with Data Management Problems. In *Proceedings of the International Workshop on Database Programming Languages*, pages 376–398, New York, NY, 1993. Springer-Verlag.