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

Object-oriented database (OODB) users bring with them large quantities of legacy data (megabytes and even gigabytes). In addition, scientific OODB users continually generate new data. All this data must be loaded into the OODB. Every relational database system has a load utility, but most 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 seven algorithms for loading data into an OODB that examine different techniques for dealing with circular and inverse relationships. We present a performance study based on both an analytic model and an implementation of all seven algorithms on top of the Shore object repository. 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 database must be reclustered for performance reasons. If the database uses physical object identifiers (OIDs), there may be no good way to recluster the objects online, 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

*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.

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 can only load data that already contains 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 OIDs. These OIDs 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.

- Inverse relationships, sometimes called bidirectional 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. Many OODB support system-maintained inverse relationships [Obj92, Ont92, Ver93, OHMS92, WI93], and they are part of the ODMG standard [Cat93]. As an 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, as by **insert** or **new** — 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. We evaluate the performance of these algorithms with an analytic model and an implementation on top of the Shore object repository [CDF+94]. We use the analytic model to explore a wide range of load file and system configurations. 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, including Snodgrass's Interface Description Language [Sno89], 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.

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 seven 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 relationships between objects and in when they create system-maintained inverse relationships.

2.1 Example database schema

```
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 Set<Experiment> expts
        inverse Experiment::input;
};
class Output {
    attribute double plant_growth;
    relationship Experiment expt
        inverse Experiment::output;
};
```

Figure 1: Experiment schema definition in ODL.

We use an example schema, which describes the data for a simplified soil science experiment, to illustrate our algorithms. In this schema, each Experiment object has a many-to-one relationship with an Input object and a one-to-one relationship with an Output object. Figure 1 defines the schema in the Object Definition Language proposed by ODMG [Cat93].

2.2 Data file description

```
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.

The data file is an ASCII text file describing the objects to be loaded[1]. We illustrate the data file format

[1]Loading from binary data files would be similar. We chose to use ASCII files because they are transferrable across different

in Figure 2. Although we developed it for the Moose data model [WI93], it fits a generic OO data model. Furthermore, any existing data file can be converted easily by a simple script to this format. Such conversions will be important for loading pre-existing data, such as the data many scientists have previously kept in flat files.

Within the data file, objects are grouped together by class, although the classes may appear in any order and a given class may appear more than once. Each class is described by its name and relationships. If a relationship of the class is not specified, then objects get a null value for that relationship. Next, each object in the class is described by a surrogate identifier and a list of its values. In this example, all the surrogates are integers, and they are unique in the data file. In general, however, the surrogates may be strings or numbers; if the class has a key they may even be part of the object's data [PG88]. The values for a collection relationship are listed inside curly brackets.

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 includes translating all the surrogates into the 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

Surrogate	OID
101	OID1
102	OID2
103	OID3
1	OID4
2	OID5
3	OID6
201	OID7
202	OID8
203	OID9

Figure 3: Id table built by the load algorithms.

All the algorithms use an *id table* to map surrogates to the database's OIDs. 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. Table 3 shows the id table built for the Experiment data file.

hardware platforms and are easy for the user to examine.

2.4 Creating relationships from surrogates

For each relationship from an 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.

OID for object to update	Surrogate for OID to store	Update offset
OID4	201	24
OID5	202	24
OID6	203	24

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

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 OID of the object to be updated, the surrogate for the OID to store in the object, and the offset at which to store the relationship. Figure 4 contains the todo list created for the Experiment data file in by the resolve-early algorithms. The todo list is read and the updates performed after the entire data file has been read.

The third technique we call *assign-early*. 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 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 [Mai94] 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 present 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.

Surrogate for object to update	OID to store	Update offset
101	OID4	12
201	OID4	8
103	OID5	12
202	OID5	8
101	OID6	12
203	OID6	8

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

In the *inverse sort* algorithms, we make an entry on an *inverse todo list*. Inverse todo entries contain the surrogate for the object to update, the OID to fill in, and an offset. The inverse todo list created for the Experiment data file is shown in Figure 5.

After reading the data file, we process the inverse todo list. The order of the entries 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' creation order in the database, which roughly corresponds to their physical order.[2] For the two-pass and resolve-early algorithms, OIDs are assigned sequentially as objects are created; therefore, the OID is the sorting key. For the assign-early algorithms, we use a creation order counter, store each object's order in its id table entry, and use the creation order as the sorting key. The creation order is chosen, instead of the actual physical order, because it matches the order of the objects seen when reading the data file a second time in two-pass algorithms and it matches the order of the todo entries in resolve-early algorithms.

Sorting is done in two phases. First, for each inverse todo list entry, we look up the OID (and creation order) of the object to be updated in the id table and add it to the entry. In this phase we read the inverse todo list in chunks and create sorted runs of 64 Kb. 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. Figure 6 shows the inverse todo list from Figure 5 after 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

[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.

OID for object to update	OID to store	Update offset
OID1	OID4	12
OID1	OID6	12
OID3	OID5	12
OID7	OID4	8
OID8	OID5	8
OID9	OID6	8

Figure 6: Inverse todo list after converting to OIDs and sorting.

load corresponds to building an inverse todo list and then processing it in a separate phase. For relational integrity checking, it is known to be faster to load relations when integrity checking is deferred, because the integrity checks can be reordered to get better sequential I/O [Moh93a].

We 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.

Minimally, the todo lists are cleared only when they become full and must be written out to disk. However, 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.

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.

Surrogate for object to update	OID to store	Update offset
201	OID4	8
202	OID5	8
101	OID6	12
203	OID6	8

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

2.7 The algorithms

We now present the seven 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 the inverse-sort technique 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.

Late-invsort Late-invsort is an optimization of smart-invsort that requires logical OIDs. In the first pass of smart-invsort, the objects are created simply to obtain their OIDs; they are not filled in until the second pass. In the first pass of late-invsort, OIDs are pre-assigned to the objects and the database is not touched. On the second pass over the data file, the inverse todo updates are merged with the object creations.

Res-early-invsort Res-early-invsort employs the resolve-early technique for surrogates and inserse-sort 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 an object are performed at once. Note that the todo list does not need to be sorted, since the order of the entries already corresponds to the creation order of the objects.

Assign-early-invsort Assign-early-invsort combines the assign-early technique for surrogates with inverse-sort for inverse relationships. It makes one pass over the data file, then sorts the inverse todo list, and 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 *nearest* locality of reference among the objects, which indicates that an object is most likely to have relationships with objects *near* it in the data file, and hence in the database. More specifically, x% of the relationships from a given object will be to objects within y% of the data file from it. The remaining (1-x)% will be to random objects. When x and y are 0, there is no locality of reference.

Nearest locality models different kinds of complex objects for a data file clustered by complex object. Y says how much of the data file each complex object spans. X 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 randomly, there would be no locality.

We now describe the cost formulas used in the analytic model. We present the (much simpler) formulas for when the id table fits in memory. We used 8 byte OIDs (this is the size used by Shore), so each two-pass and resolve-early id table entry is 12 bytes; each assign-early id table entry is 16 bytes (including the creation order); and the clearing algorithms' id table entries have an additional 4 bytes for the page numbers needed to check if an object is in the buffer pool. The parameters used in the cost of each algorithm are listed in Table 1.

The cost for each algorithm is now as follows:

$$
\begin{aligned}
naive &= 2 * P_{file} + 3 * P_{db} + 2 * P_{immedupdates} \\
P_{immedupdates} &= N_{objs} * N_{invrel} * Pb_{notinmem} \\
Pb_{notinmem} &= 1 - [(x * \frac{S_{mem} - S_{idtable}}{S_{db} * y}) + ((1-x) * \frac{S_{mem} - S_{idtable} - (S_{db} * y)}{S_{db} * (1-y)})] \\
S_{idtable} &= S_{identry} * N_{obj}
\end{aligned}
$$

Variable	Meaning
P_{file}	pages in data file
P_{db}	pages in database
S_{db}	size of database (bytes)
S_{mem}	size of memory (bytes)
$S_{identry}$	size of an id table entry (bytes)
$S_{idtable}$	size of id table (bytes)
P_{todo}	pages in todo list
$P_{invtodo}$	pages in inverse todo list
$P_{clrtodo}$	pages in cleared todo list
$P_{clrinvtodo}$	pages in cleared inverse todo list
N_{objs}	number of objects to load
N_{rel}	number of relationships per object
N_{invrel}	average number of inverse relationships per object
x	% relnships to nearby objects
y	% database considered nearby
z	% database in buffer pool
$P_{immedupdates}$	pages read into memory by immediate inverse updates
$Pb_{notinmem}$	probability that a page is not in memory
Pb_{notclr}	probability that a todo entry is not cleared
$Pb_{invnotclr}$	probability than an inverse todo entry is not cleared

Table 1: Parameters of the cost model.

Naive's 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 $N_{obj} * N_{invrel}$. 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 x and y, and use that to determine the number of I/Os incurred.

$$\begin{aligned} smart-invsort &= 2 * P_{file} + 3 * P_{db} + 4 * P_{invtodo} \\ P_{invtodo} &= N_{obj} * N_{invrel} \end{aligned}$$

Smart-invsort's inverse todo list 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 $6 * P_{invtodo}$.

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, the size of the inverse todo list is thus the same as its upper bound.

$$late - invsort = 2 * P_{file} + P_{db} + 4 * P_{invtodo}$$

The cost for late-invsort is the same as for smart-invsort, except that it does not need to update the database after creating it.

$$\begin{aligned} res - early - invsort &= P_{file} + 3 * P_{db} + 2 * P_{todo} + 4 * P_{invtodo} \\ P_{todo} &= N_{obj} * N_{rel} * 0.5 \end{aligned}$$

Res-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 $N_{obj} * N_{rel}$. However, on average, only half of the references from each object will be to objects described later on in the data file. We therefore model the size of the todo list as one-half the potential number of entries.

$$assign - early - invsort = P_{file} + 3 * P_{db} + 4 * P_{invtodo}$$

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.

$$\begin{aligned} res - clear - invclear &= P_{file} + 3 * P_{db} + 2 * P_{clrtodo} + 4 * P_{clrinvtodo} \\ P_{clrtodo} &= N_{obj} * N_{rel} * Pb_{notcleared} \\ Pb_{notcleared} &= [(x * \frac{y - z}{y}) + ((1 - x) * (\frac{1 - z}{1 - y}))] * 0.5 \\ z &= \frac{S_{mem} - S_{idtable}}{S_{db}} \\ P_{clrinvtodo} &= N_{obj} * N_{invrel} * Pb_{invnotcleared} \\ Pb_{invnotcleared} &= (x * \frac{y - z}{y}) + ((1 - x) * (\frac{1 - z}{1 - y})) \\ assign - early - invclear &= P_{file} + 3 * P_{db} + 4 * P_{clrinvtodo} \end{aligned}$$

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, late-invsort, res-early-invsort, and assign-early-invsort are not.

We also note that the I/O cost of naive is a multiple of 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 also a multiple of the number of objects and the number of relationships.)

4 Data file 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 Mb database has 25,000 objects; the 20 Mb database has 100,000 objects. The data file for the 5 Mb database was actually 2.3 Mb. We varied the locality of reference from no locality to having 90% of references stay within the nearest 10% of the database (hence called 90-10 locality). 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 the first set of experiments with the analytic model, we varied the amount of memory available for the load. In Figures 8 and 9, we show the predicted number of I/Os to load a 5 Mb database with 90-10 locality. We varied the memory available from 0.5 Mb to 10 Mb. At 10 Mb, the entire database plus all auxiliary data structures, such as the inverse todo list, fit in memory.

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 10% of the database, naive performs a full order

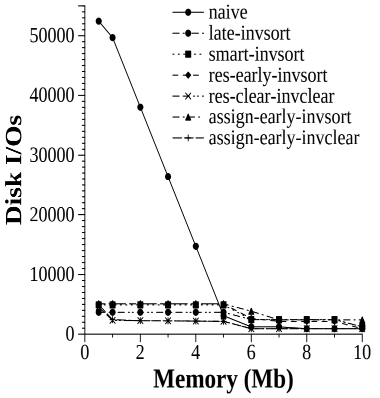

Figure 8: 5 Mb database with 90-10 locality.

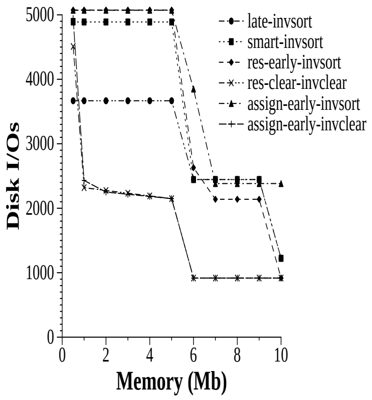

Figure 9: 5 Mb database with 90-10 locality, without naive.

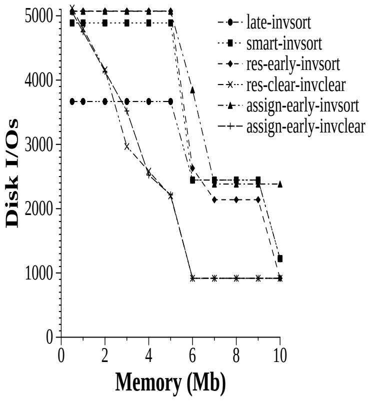

Figure 10: 5 Mb database with no locality.

of magnitude worse. Figure 9 shows the differences in performance among the remaining algorithms. At 10% of the database, or 0.5 Mb of memory, late-invsort is the best algorithm. Once 20% of the database, or 1 Mb, fits in memory, the clearing algorithms 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. Late-invsort continues to dominate the non-clearing algorithms. 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, late-invsort outperforms over the clearing algorithms until approximately half the database fits in memory, as shown in Figure 10. The relative performance of the other algorithms remains the same. However, while the non-clearing algorithms are unaffected by the locality, the clearing algorithms perform significantly worse, because fewer of the todo list and inverse todo list entries update objects that are in the buffer pool when 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 magnitude worse! With 1 Mb of available memory, naive requires 427,000 I/O's, while late-invsort performs merely 3,700 and res-clear-invclear only 4,800.

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

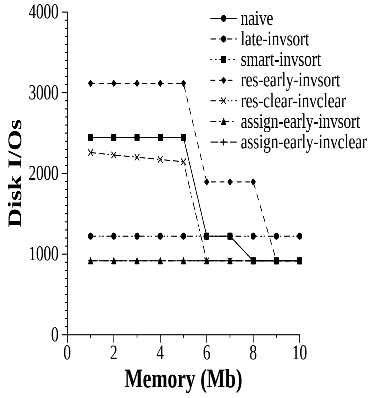

Figure 11: 5 Mb database with 20 relationships and 0 inverses.

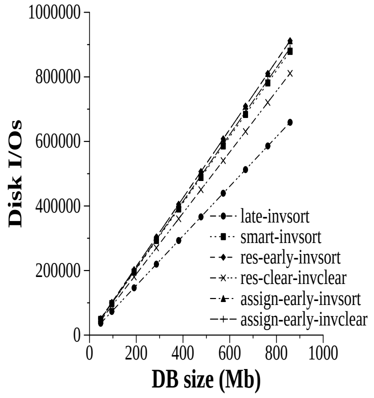

Figure 12: Scaling database size to 1 Gb, with 10% in memory.

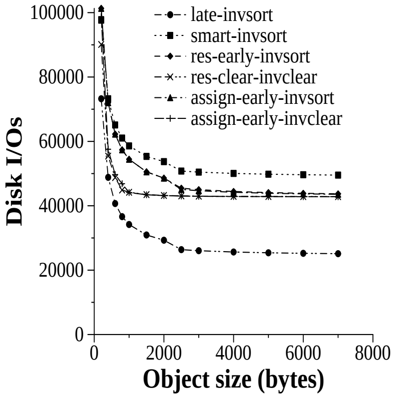

Figure 13: Scaling object size for 5 Mb database.

schemas, there are no inverse relationships. We therefore 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, 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, shown in Figure 12, we scale the database size from 5 Mb to 1 Gb, while keeping the buffer pool size equal to 10% of the database. We chose 10% since we do not expect more than that to be available for loading massive amounts of data. All the other parameters are the same as before. We verify with this experiment that the relative performance 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, shown in Figure 13, we held 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.

5.1 Discussion

According to the analytic model, the relative ranking of the algorithms is late-invsort, followed closely by assign-early-invclear and res-clear-invclear, when there is a relatively small buffer pool available, and the opposite when most of the database fits in the buffer pool. Also, the clearing algorithms perform better when there is a higher locality of reference. They are then followed 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 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 the same order of magnitude 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 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. For example, the predicted cost for late-invsort for a 5 Mb database is only 4,900 I/Os with 0.5 Mb of memory, which just barely holds the id table, but 524,000 I/Os with 0.1 Mb of memory. All of the algorithms exhibit similar one-hundred-fold increases in cost. 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 seven loading algorithms on a Hewlett-Packard 9000/720 with 32 Mbytes of physical memory. However, we were only able to use about 16 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 Seagate ST-12400N 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 users have enough disk space to accomodate such a log.

We used Shore as the underlying persistent object manager, even though Shore is still under development, for two reasons. First, Shore provides the notion of a "value-added server" (VAS), which allowed us to place the load utility directly in the server. We feel that this is the best place for a load utility; the client-server communication overhead is greatly reduced. The implementors of DB2 experienced significantly better performance when the load utility interacted directly with the buffer manager, instead of as a client [Moh93b]. Additionally, the load algorithms have direct access to the server buffer pools and can determine what is in the buffer pool at any given time, which was needed by the algorithms that try to clear the todo list and inverse todo list. The non-clearing algorithms, however, could be implemented at the client level.

Second, Shore provides logical OIDs, which we needed to test the late-invsort and assign-early algorithms, as well as physical OIDs. Shore uses a logical OID index to maps from logical OIDs to physical OIDs. This index is stored in the database.

We stored the todo list as a single large object, and the inverse todo list as several large objects, since they are too large to keep in main memory. 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 one person only one month to write.

6.1 Experimental Results

We ran experiments to load a database with 5, 20, and 50 Mb of data. 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 7, 27, and 66 Mb. The memory used by each test reflects the sum of the id table (in transient memory) and the buffer pool, since in the analytic model we did not distinguish between the two.

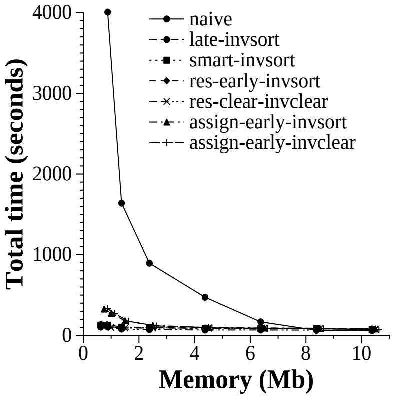

Figure 14: 5 Mb database with 90-10 locality.

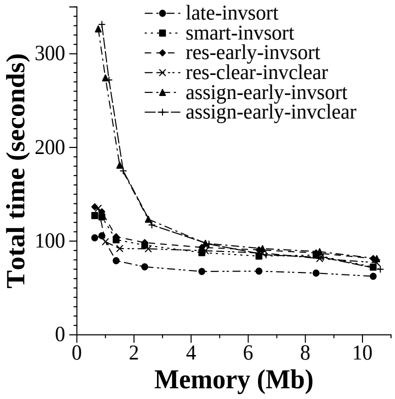

Figure 15: 5 Mb database with 90-10 locality, without naive.

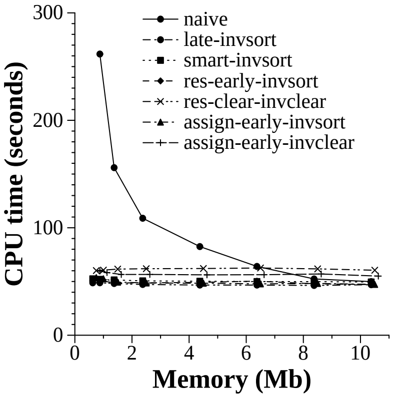

Figure 16: 5 Mb database: CPU time.

For the first set of experiments, we created a database with 5 Mb of data, which was actually 7 Mb when created and hence first fits in the buffer pool at 7 Mb. In the first experiment, shown in Figure 14, we loaded a 5 Mb of data 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 naive in Figure 15. The anomalous performance of the assign-early algorithms with a small buffer pool is caused by the logical OID index. The two-pass 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.

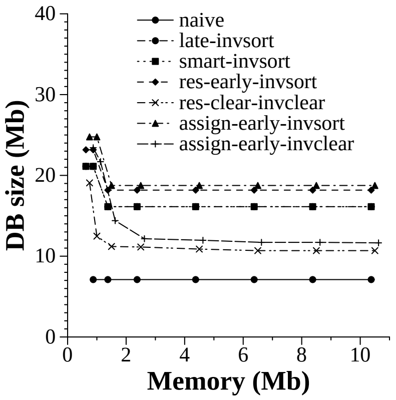

Figure 17: 5 Mb database: Disk space used by database and todo lists.

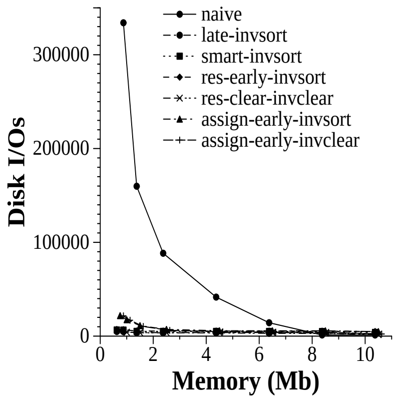

Figure 18: 5 Mb database: Disk I/Os.

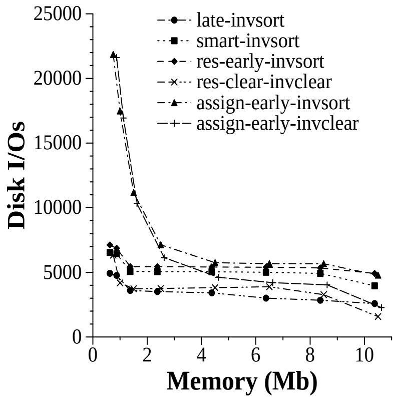

Figure 19: 5 Mb database: Disk I/O, without naive.

In all cases, late-invsort is the fastest algorithm. 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, and hence late-invsort is still better. 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 amount of disk space needed by each algorithm. The includes the size of the database, the logical OID index, and the auxiliary data structures (the todo list and inverse todo list) used. (The auxiliary data is deleted at the end of the load.) Naive uses the least amount of disk because it has no auxiliary structures. For the 5 Mb database, the logical OID index accounts for approximately 1.5 Mb of the 7 Mb stored. Like the size of the id table, the size of the logical OID index corresponds to the number of objects, rather than the absolute size of the database.

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, in Figure 9 we predicted 3597 I/Os for late-revsort with 1 Mb memory. In our experiment, late-revsort took 3667 I/Os, which is less than a 5% deviation.

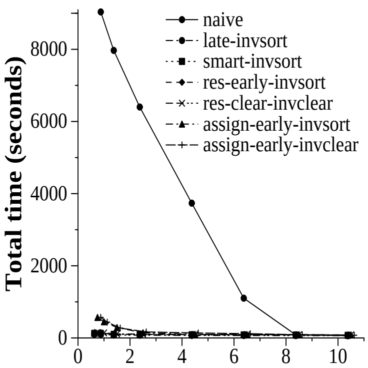

Figure 20: 5 Mb database with no locality.

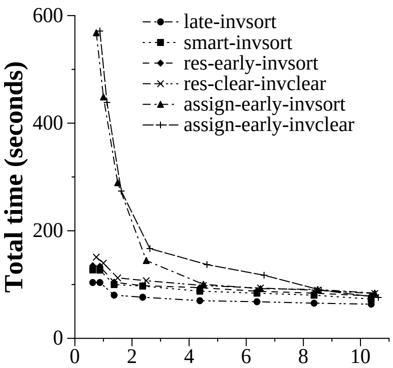

Figure 21: 5 Mb database with no locality, without naive.

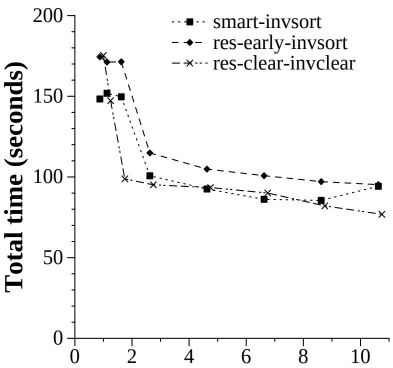

Figure 22: 5 Mb database with physical OIDs.

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 2 hours to complete the load with 1 Mb of memory, and 1 hour with 4 Mb. All the other algorithms, in contrast, take 1 to 2 minutes. The relative performance of the algorithms is similar to that with 90-10 locality, but the assign-early algorithms pay an even greater penalty for inserting into the logical OID index out of order.

We therefore decided to run some experiments to see how the algorithms perform with physical OIDs. Figure 22 show the results of these experiments. Late-invsort and assign-early depend on logical OIDs and could not be run; we also omitted naive. Contrary to our expectations, the tests with physical OIDs took *longer* to run than their logical OID counterparts. In Shore, logical OIDs are 8 bytes but physical OIDs are 12 bytes. The size of the objects thus grew from 200 bytes to 280 bytes to store the same information. The physical OID tests thus incurred many more I/Os to create the database, and since I/O costs dominate loading, the physical OID tests were slower.

In Figure 23 we show how the size of the database, todo list, and inverse todo list grew; where we needed 16 Mb of disk space for smart-invsort with logical OIDs, we needed 19 Mb with physical OIDs. The actual database grew from 7 Mb (including the logical OID index) to 7.9 Mb.

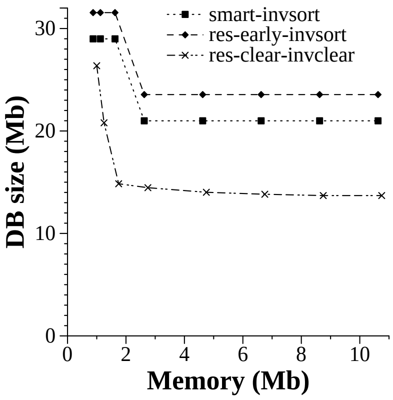

Figure 23: DB size, including auxiliary structures, with physical OIDs.

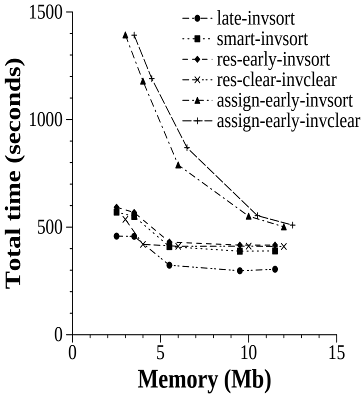

Figure 24: 20 Mb database with 90-10 locality.

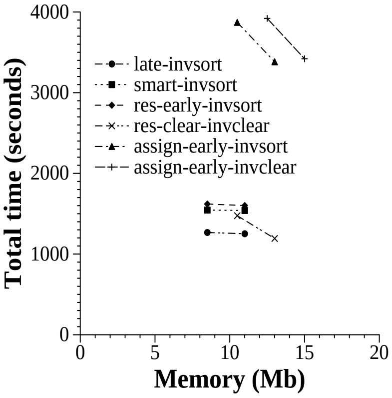

Figure 25: 50 Mb database with 90-10 locality.

Figures 24 and 25 show the results of loading 20 Mb and 50 Mb of data, respectively, with 90-10 locality 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. For example, late-invsort loaded the 5 Mb database with 0.9 Mb of Memory in 38 seconds; naive took 50 and res-early-invsort ran in 67 seconds. The fastest time to load the same database with inverse relationships was 105 seconds.

6.2 Discussion

The implementation results confirm that the analytic model predicts the actual disk I/Os for each algorithms accurately. However, because the analytic model does not account for CPU time, and did not take such factors as the logical id index under consideration, it is only a moderate predictor of actual algorithm performance.

For example, although the analytic model predicted that assign-early-invclear would sometimes beat late-invsort, the logical OID index imposed substantial I/O overhead for assign-early and thus it did not perform well. While the analytic model predicted that res-clear-invclear would beat late-invsort with high locality and a medium or large buffer pool, the CPU time involved in clearing made up for the savings in disk I/Os, and late-invsort still proved faster.

Late-invsort is the clear best choice according to the implementation results. Of the other algorithms, both smart-invsort and res-early-invsort are good choices. 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. We expect that res-early-invsort will be better when there are relatively few relationships in the data file, e.g., if much of the file describes images or other bulk data. There is not much advantage to implementing the more complicated res-clear-invclear. We therefore recommend that users implement late-invsort if pre-assigning of OIDs is possible, and either smart-invsort or res-early-invsort if not.

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 solve the problems due to relationships by (1) using a sorted inverse todo list to avoid random reads and updates and (2) using pre-allocation of OIDs to avoid updates in the first place. Of the algorithms we explored, we recommend that users implement late-invsort if logical OIDs are available, and either smart-invsort or res-early-invsort otherwise. We also note that naive's abominable performance is to be expected if objects are loaded one at a time, e.g., by **insert** or **new** statements, since inverse updates cannot be batched.

Our future work includes running experiments that load 1 Gb of data (when we get a larger disk to store the database); looking at techniques to decrease the cost of loading when the id table does not fit in memory; creating techniques to connect newly loaded objects to existing objects in 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. In addition, we plan to integrate the load implementation with the higher levels of Shore and turn it into a utility to be distributed with a future release of Shore.

8 Acknowledgements

We would like to thank David Maier for the original inspiration to study loading and for feedback on our early algorithm ideas; Mike Zwilling and C. K. Tan for advice and support for our implementation as a Shore value-added server; and Mark McAuliffe, Praveen Seshadri, Yannis Ioannidis, C. Mohan, and Dan Weinreb for their helpful comments on the content and presentation 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*, pages 383–394, 1994.

[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.

[Mai94] David Maier, January 1994. Personal communication.

[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*, pages 279–300, Chicago, Il., 1993. Springer-Verlag.

[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 of 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.