OODB Bulk Loading Revisited: The Partitioned-List Approach*†

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 and object-relational databases (OODB) need to be able to load the vast quantities of data that OODB users bring to them. Loading OODB data is significantly more complicated than loading relational data due to the presence of relationships, or references, in the data; the presence of these relationships means that naive loading algorithms are slow to the point of being unusable. In our previous work, we presented the *late-invsort* algorithm, which performed significantly better than naive algorithms on all the data sets we tested. Unfortunately, further experimentation with the late-invsort algorithm revealed that for large data sets (ones in which a critical data structure of the load algorithm does not fit in memory), the performance of late-invsort rapidly degrades to where it, too, is unusable. In this paper we propose a new algorithm, the *partitioned-list* algorithm, whose performance almost matches that of late-invsort for smaller data sets but does not degrade for large data sets. We present a performance study of an implementation within the Shore persistent object repository showing that the partitioned-list algorithm is at least an order of magnitude better than previous algorithms on large data sets. In addition, because loading gigabytes and terabytes of data can take hours, we describe how to checkpoint the partitioned-list algorithm and resume a long-running load after a system crash or other interruption.

1 Introduction

As object-oriented and object-relational 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 create only one object at a time. These methods, 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 for loading millions of objects. Loading large amounts of data is currently a bottleneck in many OODB applications [CMR92, CMR+94].

Relational database systems provide a load utility to bypass the individual language statements. The load utility takes an ASCII description of all of the data to be loaded and returns when it has loaded it. For these relational systems, a load utility significantly improves performance even when loading only a

*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.

†This paper will appear in the Proceedings of the 21st International Conference on Very Large Data Bases, Zurich, Switzerland, September, 1995.

‡Author's current address: wiener@db.stanford.edu or Dept. of Computer Science, Stanford University, Stanford, CA 94305.

small number of objects, because it is based in the database server [Moh93b]. The load utility can therefore dramatically reduce the amount of client-server interaction, and hence both the layers of software traversed and the communication overhead to create each new object.

There are many commercial object-oriented database products today, including Ontos [Ont94], O2 [Deu90], Objectivity [Obj92], ObjectStore [LLOW91], Versant [Ver93], and Gemstone [MS90]. Yet to our knowledge *no commercial OODB has a load utility.*[1] Why not? Relative to loading relational data, loading object-oriented (and object-relational) data is complicated by the presence of relationships among the objects; these relationships prevent using a relational load utility for an OODB.

- Relationships between objects are represented by object identifiers (OIDs) in the database. 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, which we call a *surrogate* identifier. We use a data structure called an *id map* to map each surrogate to its corresponding OID as the objects are loaded.

- Relationships may be forward references in the data file. The surrogate used to represent a relationship may belong to an object described later in the data file. It may not be possible to resolve the surrogate into an OID when it is first seen in the data file.

- Relationships may have system-maintained *inverse relationships*, so that the description of one object in the data file may cause another object (its inverse for a given relationship) to be updated as well. Inverse relationships are sometimes called bidirectional relationships, and are part of the ODMG standard [Cat93]. Ontos, Objectivity, ObjectStore, and Versant all support inverse relationships [Ont94, Obj92, LLOW91, Ver93].

In addition to dealing with relationships, a good load utility must be able to load data sets of widely varying sizes. As the size of the data set increases, an increasing degree of care and cleverness is required to load the data quickly. We categorize data sets into three classes of sizes, relative to the amount of physical memory available for the load, as follows:

1. *All of the data to be loaded fits in physical memory.*

Naive algorithms suffice to load this class. However, a good load algorithm can achieve a 20-30% performance gain [WN94]. Also, as we mentioned above, a load utility can decrease client-server interaction to speed up

[1] Objectivity has something it calls a load utility, however, it can only load data that already contains system-specific object identifiers (OIDs) [Obj92]. Similarly, Ontos's bulk load facility is really just an option to turn off logging while running user code that creates large amounts of data [Ont94].

loading, which individual **new** and **insert** statements can not.

2. *The data itself is too large for memory, but the id map does fit.* (The size of the id map is a function of the number of objects regardless of their size.)

For this class of load sizes, our previously proposed algorithms [WN94] work well, improving performance by one to two orders of magnitude over naive algorithms.

3. *Neither the data nor the id map fits in memory.*

As a followup to [WN94], we ran experiments with loads in this range and found that our previously proposed algorithms exhibited terrible performance due to thrashing (excessive paging of the id map) on id map lookups. Unfortunately, it is for these large loads that a fast load algorithm is most needed. In this paper we propose the partitioned-list algorithm, which, unlike our previous algorithms, provides good performance even for this range of problem sizes.

The third class is not only the most challenging, it is also the class in which many data sets fall. We give two examples of such data sets from the scientific community. In both cases, the scientists involved are using or planning to use an OODB to store their data.

- The Human Genome Database [Cam95, CPea93] is currently just over 1 gigabyte of 50-200 byte objects, containing 3-15 bidirectional relationships each. Loading this database when it moves to an OODB (which is planned for sometime in 1995) will require an id map of at least 160 megabytes.
- The climate modeling project at Lawrence Livermore National Laboratory generates up to 20,000 time points, each a complex object (i.e., many interconnected objects), in a simulation history. In the range of 40 gigabytes to 3 terabytes of data is produced in a single simulation history [DLP+93]! An id map for a single data set requires upwards of 200 megabytes.

We note that the physical memory available for a load is not necessarily the total physical memory of the machine, but rather the memory not being used by other concurrent queries or utilities.

Additionally, because loading hundreds of megabytes takes hours, it is desirable not to lose all of the data loaded when a system crash occurs. Resuming a load (or any other transaction) after a crash is non-trivial, however. We show both what to save in a *restart checkpoint* of the partitioned-list algorithm and how to resume from one of these checkpoints. We show that only minimal information need be saved in the restart checkpoints for partitioned-list, the checkpoints can be taken relatively frequently, and only the work done after the last restart checkpoint is lost if the system crashes.

1.1 Related work

Several published techniques are available for loading complex data structures with relationships from an ASCII file, but they all assume the smallest class of data sets. That is, they assume that all of the data can fit in memory and are unsuitable when it can not. Both Snodgrass's Interface Description Language [Sno89] and Pkl for Modula 3 [Nel91] fall in this category.

Our previous work focused on loading data sets in the second class of sizes [WN94]. We presented alternative strategies for handling forward references in the data file and for handling updates due to inverse relationships. We recommended one clearly superior algorithm to use for a load utility, and we will revisit its performance for loading data sets of all sizes in this paper. There has been no work, however, on loading very large data sets, i.e., data sets so large that they fall in the third class of sizes and the id map does not fit in physical memory.

Teradata provides a resumable load for their relational database [WCK93] as does DB2 [RZ89]. Mohan and Narang provide an algorithm for what to checkpoint for a resumable sort [MN92], which was the original inspiration for what to checkpoint during a load to make it resumable.

1.2 Structure of the paper

The remainder of the paper is structured as follows: In Section 2 we present an example database schema and data file. In Section 3 we review the best of our previous algorithms, describe adaptations of it for large data sets, and present our new algorithm. Section 4 describes our implementation of the algorithms and in Section 5 we present our performance tests and results. We specify how to make our new load algorithm resumable in Section 6. In Section 7 we conclude and outline our future work.

2 Loading example

We use an example database schema and data file to illustrate the loading algorithms, which we will describe in the next section.

2.1 Example database schema

The example schema describes the data for a simplified soil science experiment. 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].

```
interface Experiment {
    attribute char scientist[16];
    relationship Ref<Input> input
        inverse Input::expts;
    relationship Ref<Output> output
        inverse Output::expt;
};
interface Input {
    attribute double temperature;
    attribute long humidity;
    relationship Set<Experiment> expts
        inverse Experiment::input;
};
interface Output {
    attribute double plant_growth;
    relationship Ref<Experiment> expt
        inverse Experiment::output;
};
```

Figure 1: Experiment schema definition in ODL.

2.2 Example data file

Figure 2 shows a sample ASCII data file for the schema in Figure 1. 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, attributes, 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, 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 be part of the object's data [PG88].

Wherever 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 each surrogates into the OID assigned to the corresponding object. To facilitate this translation, we use an id map. When an OID is assigned to an object, an entry is made in the id map containing the surrogate and OID for that object. Whenever a surrogate is seen as part of the description of an object, a lookup in the id map yields the corresponding OID to store in the object.

3 Load algorithms

We first present the naive algorithm for comparison purposes. Then we present our best previous algorithm (formerly called *late-invsort* for loading the database when there is enough memory to store the id map. We also describe two simple modifications to the id map data structure which allow it to exceed memory size

```
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;
    4:  "Jill", 102, 202;
}
Output(plant_growth) {
    201:  2.1;
    202:  1.75;
    203:  2.0;
}
```

Figure 2: Sample data file for the Experiment schema.

but do not alter the basic load algorithm. Finally, we introduce our new algorithm, *partitioned-list*, which is optimized for the case when the id map is significantly larger than the amount of memory available.

3.1 Naive algorithm

The naive algorithm reads the data file twice, so that forward references in the data file can be handled correctly. The id map is stored as an in-memory open addressing hash table hashed on surrogate. It is kept separate from the database buffer pool. The algorithm has 2 steps.

1. *Read the data file.*

– For each object, read its surrogate, create an empty object, and store the surrogate and returned OID in the id map. Ignore all attribute and relationship descriptions.

At the end of this step, the id map is complete, and all of the objects have been created and initialized, although their contents are blank. Figure 3 shows the id map for the example data file.

2. *Read the data file again and create the database.*

– For each object, look up its surrogate in the id map and find out its OID.

– Read the object from the database.

– For each relationship described with the object, look up its surrogate in the id map. Store the corresponding OID in the object. Note that all valid surrogates will be found in the id map, since it was completed in step 1.

  - If the relationship has an inverse, read the inverse object from the database and store this object's OID in it.

- For each other attribute of the object, store it in the object.

- Write the updated object back to the database.

In this algorithm, each inverse relationship will cause another object to be updated, probably changing the size of that object. Sometimes the inverse object will be described later in the data file, and sometimes earlier. It is necessary to create empty objects in step 1 so that the inverse object always exists in step 2, even if it is described later in the data file. Similarly, it is necessary to read the object from the database in step 2 because it may already contain OIDs for its inverse relationships, stored there by inverse updates to this object.

One additional step is necessary before pronouncing the load complete. If an archive copy of the database is maintained in case of media failure, a full archive copy of the newly loaded data must be made before the data is read or written [MN93]. Otherwise, the data might be lost due to media failure, since there is no log record of the loaded data. This step will be necessary for all of the algorithms although we do not mention it again.

3.2 Basic algorithm: id map is an in-memory hash table

In this algorithm, the data file is also read twice to handle forward references, and the id map is also an in-memory hash table. Unlike in the naive algorithm, OIDs are *pre-assigned* to objects. Pre-assigning an OID involves requesting an unused OID from the database without creating the corresponding object on disk. In general, pre-assignment is only possible with logical OIDs, since the future location and size of the object are not known when its OID is pre-assigned. If the database only provides physical OIDs, the algorithm may be modified to create the object in order to get its OID.

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 [CDF$^+$94]. Using logical OIDs does require the database to store an extra index that maps logical to physical OIDs. However, we found previously that the advantages of using logical OIDs (such as having smaller OIDs) outweigh the costs of maintaining and using this index [WN94].

The algorithm has 3 steps.

1. *Read the data file.*

- For each object, pre-assign an OID, and make an entry in the id map.

Surrogate	OID
101	OID1
102	OID2
103	OID3
1	OID4
2	OID5
3	OID6
4	OID7
201	OID8
202	OID9
203	OID10

Figure 3: Id map built by the load algorithm.

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
102	OID7	12
202	OID7	8

Figure 4: Inverse todo list built for the load algorithms.

- For each relationship described with that object, if the relationship has an inverse, make an entry on an *inverse todo list*, indicating that the inverse object should be updated to contain the OID of this object. Note that the inverse object is described by a surrogate in the data file. Therefore each inverse todo entry contains a surrogate for the object to update, the OID to store in that object, and an offset indicating where to store it.

At the end of this step, the id map is complete and the inverse todo list contains entries for all of the updates dictated by inverse relationships. The inverse todo list is stored in the database, because it is too large to reside in memory. Figures 3 and 4 show the id map and inverse todo list for the example data file.

2. *Sort the inverse todo list.*

- First, translate to an OID the surrogate in each entry indicating which object to update. Each translation involves a lookup in the id map. We call the new entries *update* entries, and the translated list an *update list*. Each update entry consists of the OID of the object to update, the OID to store in it, and an offset.
- Sort the update entries, using a standard external sort, by the OID of the object to update. In our sort, we sort chunks of the update list in memory as the update list is generated, and write each chunk out to

OID for object to update	OID to store	Update offset
OID1	OID4	12
OID1	OID6	12
OID2	OID7	12
OID3	OID5	12
OID8	OID4	8
OID9	OID7	8
OID9	OID5	8
OID10	OID6	8

Figure 5: Update list after sorting.

disk as a sorted run. Then we merge the sorted runs, as many as can fit in memory at once, until the entire list is sorted. However, we defer the final merge pass until step 3. Figure 5 shows the sorted update list.

Sorting by OID groups all of the updates to a single object together. Additionally, because the OIDs are assigned in increasing order, it organizes the updates into the same order as the objects appear in the data file. (If OIDs are not assigned in a monotonically increasing sequence, an integer counter can be used to assign a creation order to each object. Then the creation order for each object is stored as an additional field in the id map and in each update entry, and used as the sort key.)

3. *Read the data file again and create the database.*

– For each object, look up its surrogate in the id map and retrieve the OID that has been assigned to it.

– For each relationship described with the object, look up its surrogate in the id map. Store the retrieved OID in the object.

– Store each other attribute of the object in the object.

– For each update entry that updates this object, read the entry from the sorted update list and store the appropriate OID in the object.

The last merge pass of sorting the update list happens concurrently with reading the data file and creating the objects. Note that except for the id map, which is in memory, no other data structures or objects are being accessed repeatedly during any step of the algorithm. Therefore, the buffer pool is fully available for merging the last set of sorted runs in step 3 and the last merge pass is never written to disk; the merged entries can be discarded as soon as they have been retrieved and read.

An important advantage of this algorithm over similar variants is that by pre-assigning OIDs to objects in step 1, creating the objects can be postponed until *all* the data (including inverse relationships) to be stored in an object is available. Therefore, when the object is initially created, it is of the correct size. (This advantage is only possible with logical OIDs.) Updates that add inverse relationships, on the other hand, usually change the size of the object. Since changing the sizes of objects often severely impacts the clustering of the objects, this advantage is quite significant.

Note that, except for the id map, no data structure — inverse todo list, update list sorted run, database object — is read or written more than once. We have eliminated all of the random and multiple accesses to objects that were caused by inverse relationship updates in the naive algorithm. Lookups in the id map are still by random associate access. However, if the id map is resident in physical memory, then no I/Os are necessary to read from it.

3.3 Modification 1: id map is a persistent B$^+$-tree

The basic algorithm requires enough virtual memory to store the id map. However, it accesses the id map so frequently that it is more correct to say that it expects and relies on both the buffer pool and the id map being in physical memory. To overcome the limitation of needing physical memory for the id map, we redesigned it as a persistent B$^+$-tree. The amount of the id map resident in memory is constrained by the size of the buffer pool. Paging the id map is delegated to the storage manager, and only the buffer pool needs to remain in physical memory. The load algorithm remains the same; id map lookups are still associative accesses, now to the B$^+$-tree.

Note that the choice of a B$^+$-tree instead of a persistent hash table will not be relevant: the important feature of the B$^+$-tree is that it supports random accesses to the id map by paging the id map in the buffer pool. The same would be true of a persistent hash table, and in both cases, random accesses to the id map mean random accesses to the pages of the persistent B$^+$-tree or hash table.

3.4 Modification 2: id map is a persistent B$^+$-tree with an in-memory cache

Changing the id map from a virtual memory hash table to a persistent B$^+$-tree had the side effect of placing the id map under the control of the storage manager. In addition to managing the id map's buffer pool residency, the storage manager also introduces concurrency control overhead to the id map. To avoid some of this overhead, we decided to take advantage of a limited amount of virtual memory for the id map. For this algorithm, we keep the B$^+$-tree implementation of the id map, but introduce an in-memory cache of id map entries.

The cache is implemented as a hash table. All inserts store the id map entry in both the B$^+$-tree and the cache. Whenever an insert causes a collision in a hash bucket of the cache, the previous hash bucket entry is discarded and the new entry inserted. All lookups check the cache first. If the entry is not found in the cache, the B$^+$-tree is checked, and the retrieved entry is inserted into the cache.

The load algorithm remains the same as the basic algorithm; id map lookups are still associative accesses, now first to the cache and then, if necessary, to the B$^+$-tree.

3.5 New algorithm: id map is a persistent partitioned list

The associative accesses to the id map in the above algorithms are random accesses, and when the id map does not fit in physical memory, they cause random disk I/O. The goal of this algorithm is to eliminate the random I/O.

In the above algorithms, half of the id map lookups were to convert the inverse todo entries into update entries. A fundamental observation led to the new algorithm: these lookups, grouped together, constitute a join between the inverse todo list and the id map. The other half of the id map lookups were performed while reading the data file a second time, in step 3, to retrieve the OIDs for each relationship. These lookups were interspersed with reading the data file and creating objects. However, by separating the surrogates from the data file, into a different todo list, the surrogates can also be joined, as a group, with the id map.

In this algorithm the id map is written to disk in two different forms at the same time. First, it is written sequentially, so that the OIDs can be retrieved in the same order as they are generated, which is necessary for creating the objects with the OIDs that have been pre-assigned to them. Second, the id map is written so that it can be joined with the surrogates to be looked up. We use a hash join with the id map as the inner relation, so the second time, the id map is written into hash partitions. As many partitions as can fit in the buffer pool are initially allocated. The algorithm has 5 steps.

Partition 0	
Surrogate	OID
102	OID2
2	OID5
4	OID7
202	OID9

Partition 1	
Surrogate	OID
101	OID1
103	OID3
1	OID4
3	OID6
201	OID8
203	OID10

Figure 6: Id map with 2 partitions.

1. *Read the data file.*

- For each object, read and hash the surrogate. Pre-assign an OID to the object, and make an entry in both the sequential id map and in the id map hash-partitioned on surrogate.

- For each relationship described with the object, hash the surrogate for the relationship and make an entry

Partition 0		
OID for object to update	Surrogate for object to store	Update offset
OID5	202	24
OID7	102	16
OID7	202	24

Partition 1		
OID for object to update	Surrogate for object to store	Update offset
OID4	101	16
OID4	201	24
OID5	103	16
OID6	101	16
OID6	203	24

Figure 7: Todo list with 2 partitions.

Partition 0		
Surrogate for object to update	OID to store	Update offset
202	OID5	8
102	OID7	12
202	OID7	8

Partition 1		
Surrogate for object to update	OID to store	Update offset
101	OID4	12
201	OID4	8
103	OID5	12
101	OID6	12
203	OID6	8

Figure 8: Inverse todo list with 2 partitions.

in the appropriate partition of a *todo list*. Each todo entry contains the OID just pre-assigned to the object, the surrogate to look up, and an offset indicating where to store it. The todo list is hash-partitioned by surrogate, using the same hash function as the id map.

– If the relationship has an inverse, hash the surrogate for the relationship and make an entry on an *inverse todo list*, indicating that the inverse object should be updated to contain the OID of this object. Each inverse todo entry contains a surrogate for the object to update, the OID to store in that object, and an offset indicating where to store it. The inverse todo list is also hash-partitioned by surrogate, using the same hash function as the id map.

The buffer pool size determines the number of partitions of the id map. Since there must be a corresponding todo list partition and inverse todo list partition for each id map partition, the number of partitions is no more than the number of pages in the buffer pool divided by 3.

Figure 7 shows the todo list constructed for the example data file, with 2 partitions. The id map and the inverse todo list are the same as for the basic algorithm, shown in Figures 3 and 4, except that they are now partitioned as shown in Figures 6 and 8.

2. *Repartition the id map as necessary.*

If any of the id map partitions is too large to fit in memory, split that partition and the corresponding todo list and inverse todo list partitions by further hashing on the surrogates. Repeat until all id map partitions can (individually) fit in memory, which is necessary for building the hash table in the following step.

3. *Join the todo and inverse todo lists with the id map to create the update list. Join one partition at a time.*

- Build a hash table on the entries in the id map partition in virtual memory.
- For each entry in the todo list partition, probe the hash table for its surrogate. Make an entry on the update list containing the OID of the object to update (taken from the todo entry), the OID to store in the object (just retrieved from the hash table), and the offset (from the todo entry).
- For each entry in the inverse todo list partition, probe the hash table for its surrogate. Make an entry on the update list containing the OID of the object to update (just retrieved from the hash table), the OID to store in the object (taken from the inverse todo entry), and the offset (from the inverse todo entry).

Note that all entries on the update list look alike, regardless of whether they were originally on the todo or inverse todo list.

4. *Sort the update list.*

- As the update entries are generated in step 3, sort them by OID of the object to update and write them out in sorted runs. In this step, use an external merge sort to merge the sorted runs. As in the basic algorithm, postpone the final merge pass until the last step.

5. *Read the data file again and create the database.*

- For each object, look up its surrogate in the sequential id map and retrieve the OID that has been assigned to it.
- For each non-relationship attribute of the object, store it in the object.
- For each update entry in the sorted update list that updates this object, read the entry and store the appropriate OID in the object. The final merge pass of sorting the update list happens as the update entries are needed.
- Create the object with the storage manager now that the in-memory representation is complete.

Note that since the sequential id map entries are read in the same order in which they are generated, it is only necessary to store the OID in each entry, and not the corresponding surrogate.

Each data structure used by the load is now being written and read exactly once, in sequential order. There is very little random I/O being performed on behalf of the algorithm, because there is no longer random access to any load data structure.

4 Implementation

We implemented all of the algorithms in C++. The database was stored under the Shore storage manager [CDF+94]. We used the Shore persistent object manager, even though it 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. Second, Shore provides logical OIDs, which allowed us to pre-assign OIDs. Each list or list partition (e.g., each partition of the id map in the new algorithm) was stored as a single large object in Shore.

We used a Hewlett-Packard 9000/720 with 32 megabytes of physical memory to run all of the algorithms. However, due to operating system and daemon memory requirements, we were only able to use about 16 Mb for any test run. The database volume was a 2 gigabyte Seagate ST-12400N disk controlled exclusively by Shore. The data file resided on a separate disk and thus did not interfere with the database I/O. For these tests, we turned logging off, as would be expected in a load utility [Moh93a].

5 Performance tests

We ran a series of performance tests to show how quickly (or slowly!) each algorithm could load different size data sets. In this section, we will use the names in Table 1 to refer to each algorithm.

Algorithm	Major data structures	Described in
naive	in-memory hash table id map immediate inverse updates	Section 3.1
in-mem	in-memory hash table id map inverse todo list	Section 3.2
btree	B^+-tree id map inverse todo list	Section 3.3
cache	B^+-tree plus cache id map inverse todo list	Section 3.4
partitioned -list	partitioned list id map todo and inverse todo lists	Section 3.5

Table 1: Algorithm names used in performance graphs.

5.1 Data sets loaded

For the performance experiments we created 200 byte objects. The schema for each object contained ten bidirectional relationships. In the data file, we listed five relationships with each object. Each object also contained an average of five relationships that were listed with the inverse object, but stored in both objects.

(For the final set of experiments, we explicitly listed all ten relationships with each object and had no inverses in the schema.) We varied the number of objects to control the size of the database; each data set had 5000 objects per megabyte (Mb) of data. The data files were approximately one-third as large as the data sets they described, e.g., the data file for the 100 Mb data set was 36 Mb.

We modeled a high degree of locality of reference among the objects for most of the experiments. In the data sets with a high degree of locality, 90% of relationships from an object are to other objects within 10% of it in the data file (and hence the database). The remaining 10% of the relationships are to other objects chosen at random from the entire database. We believe that a high degree of locality models a database clustered by complex object. For a few experiments, we also modeled no locality of reference. In those data sets, all relationships are to objects chosen at random.

5.2 Performance results

5.2.1 Comparing algorithms with different classes of data set sizes

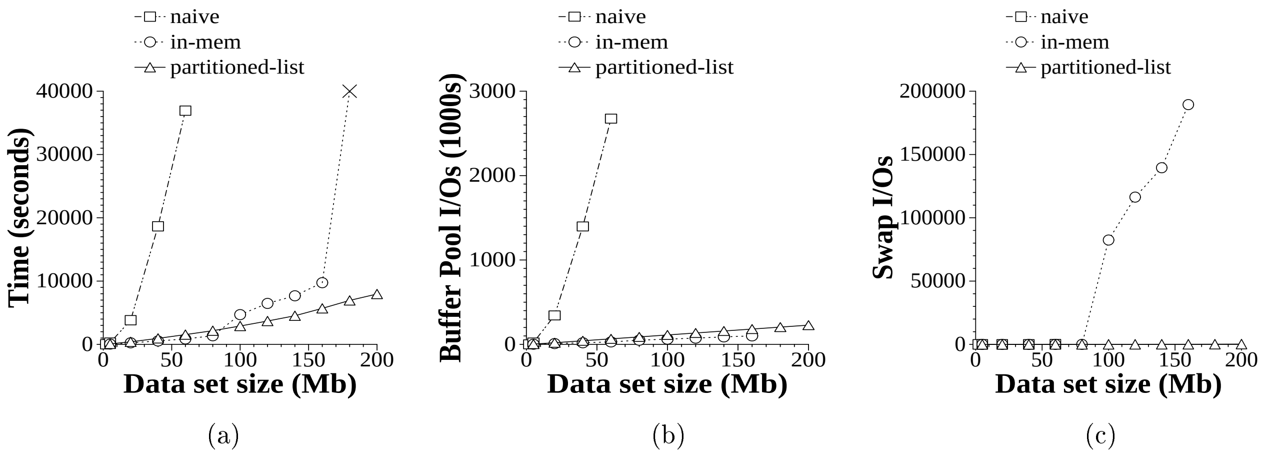


Figure 9: Comparing algorithms across different classes of data set sizes.

For the first set of experiments, we compared the performance of naive, in-mem and partitioned-list. We held the buffer pool size constant at 4 Mb, and varied the size of the data set being loaded from 5 Mb to 200 Mb.

Each algorithm used some transient heap memory in addition to the buffer pool. Naive and in-mem allocate the id map in virtual memory. With the 5 Mb data set, the id map was 0.4 Mb; with the 100 Mb data set it was 12.5 Mb. In general, the size of the id map was 7-13% of the data set size. As the data set size increases, so does the amount of memory used by naive and in-mem; the total amount of memory each used was the size of the buffer pool plus the size of the id map.

Partitioned-list creates a full page of data for each id map, todo list, and inverse todo list partition in memory before it sends that page to the storage manager. This minimizes the number of calls to the storage manager and reduces the rate of pinning and unpinning of pages and objects in the buffer pool, but it requires roughly as many pages of heap memory as there are pages in the buffer pool. The total amount of memory required by partitioned-list is therefore twice the size of the buffer pool.

Therefore, for the smallest class of data set sizes, the partitioned-list algorithm used more memory than the other algorithms. However, with a 4 Mb buffer pool, naive and in-mem were already using more memory to load the 60 Mb data set: they used 10.3 Mb while partitioned-list used only 8 Mb.

Figure 9(a) shows the wall clock time for the naive, in-mem, and partitioned-list algorithms as we loaded data sets of 2 to 200 Mb. Figures 9(b) and 9(c) show the number of I/Os that were performed by the same experiments: 9(b) depicts the number of I/Os in the buffer pool, and 9(c) depicts the number of I/Os performed as virtual memory is swapped in and out of physical memory. (We measured the virtual memory page swaps with the *getrusage* system call; Shore provided the buffer pool I/O statistics.) We show the partitioned-list algorithm above for comparison and discuss it in more detail in the next section.

The naive algorithm performs comparably to the in-mem algorithm on the 2 Mb data set when nearly the entire data set fits in the 4 Mb buffer pool: they complete the load in 19.6 and 19.0 seconds, respectively. However, as the data set size increases, naive starts thrashing as it tries to bring the inverse relationship objects into the buffer pool, as is clear from the correlation between the wall clock time and the buffer pool I/Os. At 5 Mb, naive is already taking 5 times as long to load: 276 vs. 51 seconds. Naive's performance is so poor because the buffer pool must randomly read and write an object for each inverse update. By the 60 Mb data set, naive is over an order of magnitude worse than in-mem, taking over 10 hours to load while in-mem finishes in 15 minutes. Naive is clearly unsuitable for loading once the data set exceeds the size of the buffer pool.

The in-mem algorithm performs quite well — the best — until the id map no longer fits in physical memory. For the 80 Mb data set, the id map still fits in physical memory. As the id map grows for the data sets between 100 and 160 Mb, it no longer fits in physical memory, and the load time for in-mem becomes proportional to the number of I/Os performed for virtual memory page swaps. By the 180 Mb data set, the id map is 25 Mb and virtual memory begins to thrash so badly that the load cannot complete at all. In fact, in over 4 hours, in-mem had completed less than 10% of step 1 of the algorithm. (In-mem loaded the entire 160 Mb data set in under 3 hours.) We therefore recommend that in-mem be used *only* when there is plenty of physical memory for the id map.

In-mem is better than partitioned-list when the id map does fit in memory because it writes neither the id map nor a todo list to disk. This performance gap could be narrowed by using a hybrid hash join [Sha86] in the partitioned-list algorithm instead of the standard Grace hash join [Kea83] to join the id map, todo list, and inverse todo lists. However, it would only save writing the id map to disk; the todo list (which is not needed by in-mem) would still be written to disk.

5.2.2 Comparing viable algorithms for loading large data sets with very little memory

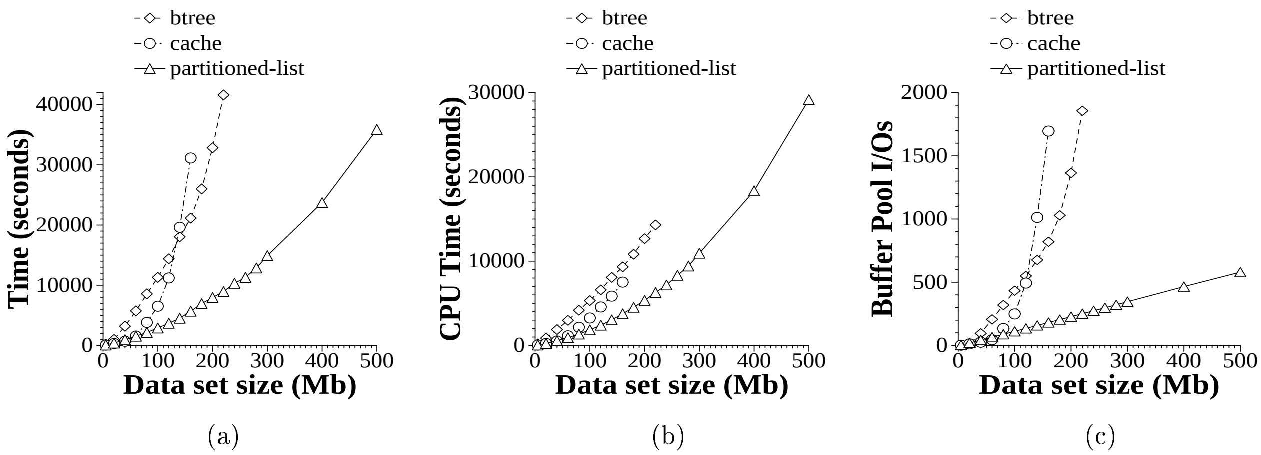


Figure 10: Comparing algorithms for loading data sets with very little memory available.

In-mem is simply not a viable algorithm when the size of the id map, which depends on the number of objects to be loaded, exceeds the size of memory. However, both modifications to the in-mem algorithm place the id map in the buffer pool, and allow the storage manager to handle paging it in and out of physical memory. Therefore, the amount of physical memory required remains constant. We now compare the modified algorithms, btree and cache, to partitioned-list.

As we noted in the previous section, partitioned-list used both the 4 Mb buffer pool and an equal amount of transient memory, for a total of 8 Mb. Cache used both the buffer pool and an in-memory cache of the id map. We therefore allocated a 4 Mb cache as well as the 4 Mb buffer pool, so that cache also used 8 Mb of physical memory. B^+-tree had no transient memory requirements; therefore, to be fair in terms of total memory allocated, we tested the B^+-tree algorithm with an 8 Mb buffer pool.

Figure 10 shows the wall clock time, CPU time, and number of buffer pool I/Os incurred by the btree, cache, and partitioned-list algorithms to load data sets of 5 to 500 Mb. (We were unable to load more than 500 Mb due to disk space limitations.)

Partitioned-list is clearly the best algorithm; it loaded all of the data sets in the least amount of wall clock time and CPU time, with the fewest number of I/Os. However, although the number of I/Os scales linearly with increasing data set sizes, the CPU time (and hence the wall clock time) does not. This is due to a bug in the page allocation routine of the Shore storage manager which has since been fixed.[2] Comparing

[2]Preliminary tests with the bug fix show linear CPU time, so that, e.g., partitioned-list finishes loading the 500 Mb data set

the wall clock time to the CPU time for partitioned-list reveals that approximately 75% of the wall clock time is CPU-time. This is due to two factors. First, a background thread writes the dirty pages of the buffer pool out to disk asynchronously, in groups of sequential pages, and so many of the write operations overlap with the CPU time. Second, the partitioned-list algorithm was carefully designed to minimize I/O. We succeeded in this regard; partitioned-list is not I/O-bound.

The cache algorithm is competitive with partitioned-list while the id map fits in memory. As the data set sizes exceeds 80 Mb, however, the close correlation between the wall clock time and the amount of buffer pool I/O for the cache algorithm show that it is spending most of its time bringing id map pages into the buffer pool. To load the 160 Mb data set, cache takes nearly 9 hours (while partitioned-list completes the load in 1.5 hours).

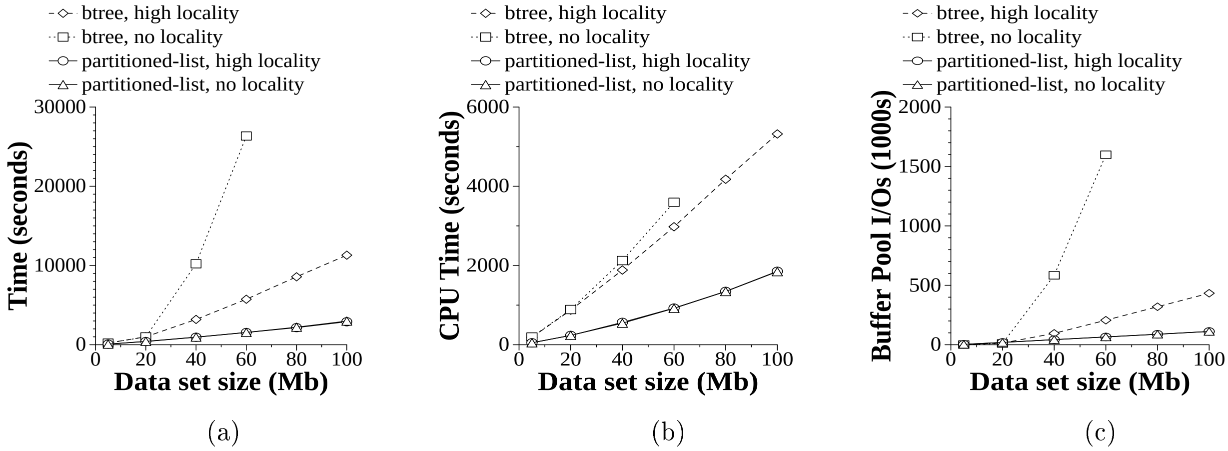


Figure 11: Comparing the sensitivity of the btree and partitioned-list algorithms to the locality of reference in the data set.

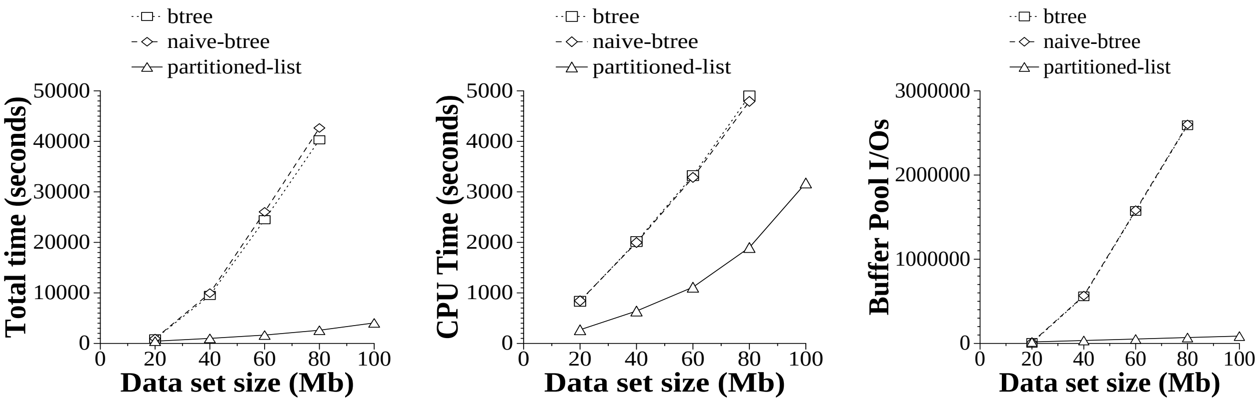
The cache algorithm is better than the btree algorithm while the id map fits in the total allocated memory, because cache accesses are much faster than B^+-tree accesses in the buffer pool. (Even when the desired page of the B^+-tree is resident in the buffer pool, accessing it still involves fixing the page, pinning and locking the B^+-tree entry, and other concurrency control operations.) However, once the id map greatly exceeds the cache size, most id map lookups go through the B^+-tree. Then the btree algorithm is better because it has twice as large a buffer pool in which to keep pages of the id map resident. (The cache algorithm has the same total amount of memory, but there is a high degree of duplication between the cache and B^+-tree entries.)

We expected the btree algorithm to begin paging the id map once the size of the id map exceeded the buffer pool. Yet it is not until loading the 180 Mb data set that the btree algorithm begins to page in the buffer pool. A combination of three factors explains btree's ability to load 160 Mb (with a 36 Mb id map)

in 3.5 hours. The perforance of I/O-bound algorithms such as btree and naive is not significantly affected.

without excessive paging in the buffer pool. First, surrogates are assigned sequentially in our test data files, so two objects listed consecutively in a data file have consecutive surrogates. Second, a high locality of reference means that most of the lookups will be for surrogates sequentially close to that of the referencing object. Third, a clustered B^+-tree index means that sequentially similar keys (surrogates) will be in nearby entries. Therefore, for the above data sets, keeping the nearest 20% of the id map in the buffer pool suffices to satisfy 90% of the id map lookups. 20% of the id map for the 160 Mb data set is only 7 Mb, and fits easily in the 8 Mb buffer pool.

Loading some data sets with no locality of reference validated this theory, and we show the results in Figure 11. There was no difference in the partitioned-list algorithm when loading data sets with and without locality of reference; the times varied so slightly that the lines appear to overlay each other on the graph. The btree algorithm, by contrast, was very sensitive. As soon as the entire id map did not fit in the buffer pool, the btree algorithm began thrashing as we originally predicted. This happened at the 40 Mb data set, when the id map was approximately 10 Mb, and got worse for larger data sets. It is interesting to note that there is very little extra CPU overhead to fetch a non-resident page; most of the CPU time is spent on concurrency control inside the buffer pool, which happens whether or not the page must first be (expensively) fetched from disk.



(a) (b) (c)

Figure 12: Comparing algorithms when there are no inverse relationships.

5.2.3 Comparing large data set algorithms when there are no inverse relationships

Although all of the major commercial OODB vendors support inverse relationships, many object-relational DBs do not (e.g., Illustra [Ube94] and UniSQL [Kim94]) and/or users may choose not to use them. As a final comparison of the algorithms for handling large data sets, we altered the data file to explicitly list all ten relationships from each object and removed the inverse relationships from the schema. We ran both partitioned-list and btree, as well as a version of naive, identified as naive-btree, that was adapted to use a B^+-tree id map instead of keeping the id map in memory. Figure 12 shows the results. Naive-btree and btree are nearly indistinguishable on the graphs (btree is actually 5-6% faster). The major difference between them, their handling of inverse relationships, has been removed. However, partitioned-list is clearly still an order of magnitude faster than both of them, completing the load of 80 Mb in less than 1 hour while btree takes over 11 hours.

5.2.4 Discussion

When the id map fits in memory, partitioned-list, our new algorithm, is less than twice the cost of in-mem, which does not create a todo list nor write the id map to disk. Using hybrid hash join instead of Grace hash join to join the id map with the todo list and inverse todo list would eliminate the extra cost of writing the id map and narrow the gap, but the todo list would still be written. When the id map does not fit in memory, in-mem is simply inviable, first because it thrashes virtual memory and then because it runs out of swap space.

Partitioned-list is an order of magnitude better than either btree or cache, the other algorithms that can deal with an id map that does not fit in memory. Without locality of reference in the data set, partitioned-list completed 16 times faster than btree on a 60 Mb data set! Even when there are no inverse relationships in the schema, e.g., in object-relational databases such as Illustra [Ube94] and UniSQL [Kim94], partitioned-list is an order of magnitude faster. By eliminating all random accesses to data structures, and by writing and reading each data item exactly once, we achieve *linear I/O costs* for partitioned-list in the size of the data set. For a system that needs to handle very large loads, e.g., gigabytes of data, and does not have gigabytes of memory, we recommend partitioned-list as the best algorithm to implement.

6 Resumable load

From the above performance results, it is clear that loading gigabytes and terabytes takes hours. Losing all of the first ten hours of a load due to a system crash would be extremely undesirable. Rather, we would like to be able to resume the load after the system recovers, and to resume it close to where it was at the time of the crash. It is also desirable to resume a load that was stopped for other reasons, e.g., to use all of the CPU power for an urgent query, or because disk space was temporarily unavailable. Furthermore, we wish to resume a load without requiring logging of the newly loaded data, and without undoing most of the load before restarting.

Therefore, the best solution for resuming a load is to periodically take a *restart checkpoint*: to commit the current state of the load and save persistent information that indicates where and how to resume the load from this checkpoint. In this section we discuss how to adapt the partitioned-list algorithm presented in Section 3.5 to make it resumable. We identify the specific information to save during a restart checkpoint and discuss how to resume the partitioned-list algorithm from a given checkpoint.

6.1 Restart checkpoint records

Whenever a restart checkpoint is taken, it is necessary to flush all partitions and lists from memory to the buffer pool, and then flush the dirty pages of the buffer pool to disk. Then a restart checkpoint record containing the necessary information is written and also flushed to disk. Flushing the buffer pool ensures that the state of the load as of the checkpoint can be recovered from disk after a crash. Teradata also flushes all loaded data to disk when taking a resumable load checkpoint [WCK93].

For each step of our partitioned-list load algorithm, we now summarize the action of the step, describe when a checkpoint is permitted, what to write in the checkpoint record, and how to use the checkpoint record information to resume the load.

1. *Read the data file and create the sequential id map, id map partitions, todo list partitions, and inverse todo list partitions.*

A restart checkpoint is permitted between reading any two objects. After the Nth object, record the current position in the data file, the sequential id list, and each id map, todo list, and inverse todo list partition. When resuming a load at this checkpoint, discard all entries in the above lists and partitions after the recorded positions. Then continue by reading the $(N+1)$th object from the data file.

2. *Join the id map, one partition at a time, with the todo list and inverse todo list to create the update list.*

A restart checkpoint is permitted at any time. Record which partition is being joined, and the current position in either the todo or inverse todo list (whichever is being joined at the time). Record the current end of the update list. When resuming a load from this checkpoint, first rebuild the hash table on the id map partition. Discard all update entries after the recorded point in the update list. Then continue reading the todo or inverse todo list and joining it with the id map. Note that taking a checkpoint terminates a sorted run of the update list; very frequent checkpoints may generate more runs to merge than would otherwise be created.

3. *Merge sorted runs of the update list until only one merge pass remains.*

A restart checkpoint is permitted at any time. Record which runs are being merged, the location of the next entry to merge in each run, and the end of the new merged run. To resume, discard all entries in the merged run after the recorded point, and all entries in subsequent merged runs. Resume merging from the recorded points in each sorted run.

4. *Read the data file and sequential id map, perform the final merge pass of the update list and create the database objects.*

A restart checkpoint is permitted between creating any two objects. Record the current position in the data file, sequential id map, and each sorted run of the update list. Record the OID of the last created object.

Resuming from a checkpoint here is trickier; objects that were created after this checkpoint, but before the crash, cannot simply be "recreated:" they already exist and trying to create an object with the same OID would cause an error. However, if only part of an object (spread across multiple pages) was written to disk, then the entire object is invalid. It is therefore necessary to remove the objects created after the checkpoint and recreate them. However, the objects cannot simply be deleted because that would invalidate their OIDs, which we are already using to reference those objects. We recommend truncating the objects to zero length instead.

Resume reading the data file, sequential id map, and update list from their respective recorded points. For each object, try to read the object with the corresponding OID from the database. If the object is found, first truncate it, then update it with the correct data. When an object is not found, resume normal loading with creating that object.

As we indicate in each step above, a checkpoint can be taken at virtually any time during the load. However, there is a tradeoff between taking frequent checkpoints (and losing little work) and occasional checkpoints (and avoiding the overhead of flushing the buffer pool). Some balance between the two should be struck.

7 Conclusions

A bulk loading utility is critical to users of OODBs with significant amounts of data. Loading new data is a bottleneck in object-oriented applications; however, it need not be. In this performance study we showed that even when less than 1% of the data fits in memory, good performance can still be achieved. The key lies in minimizing the number of random accesses to both the database and any other secondary storage data structures.

In this paper we developed algorithms to load a data set so large that its id map can not fit in physical memory. We believe that many scientific data and legacy data sets fit in this category. We presented a new algorithm, partitioned-list, in which we were able to eliminate random data accesses by writing the id map out to disk as a persistent list, and then using a hash join to perform lookups on the id map. This fundamental change allowed the algorithm to scale gracefully with increasing data sizes, instead of spending all its time bringing needed id map pages into memory once the id map (as either a virtual memory structure or as a persistent B^+-tree) no longer fit in physical memory.

Our new algorithm also incorporates techniques for efficiently handling inverse relationships. We note that a load utility must be able to turn off the automatic maintenance of inverse relationships for the duration of the load. Otherwise, the load utility can do no better than a naive algorithm, i.e. orders of magnitude worse than a clever algorithm for handling inverse relationships. However, the partitioned-list algorithm achieves another order of magnitude performance improvement on top of that for handling inverse relationships due to its handling of the id map lookups. This performance gain occurs even in data whose relationships have no inverses.

We also presented a resumable load algorithm. We described both what to write in a checkpoint record for the partitioned-list algorithm and how to resume the algorithm from the last checkpoint. Restart checkpoints allow a single load transaction to be paused and resumed many times, for any reason, with a minimal loss of work.

Our future work includes looking at techniques for loading new objects which share relationships with existing objects in the database. Specifically, we are investigating when to retrieve such objects from the database and when to update them to maximize both concurrency and loading speed. We also plan to investigate algorithms for loading objects in parallel on one or more servers with multiple database volumes. In addition, we are integrating the load implementation with the higher levels of Shore and turning it into a utility to be distributed with a future release of Shore.

8 Acknowledgements

We would like to thank S. Sudarshan for suggesting that the id map lookups could be seen as a join, and C. Mohan and Nancy Hall for discussion of our resumable load ideas. We also want to thank Mike Zwilling and C. K. Tan for many hours of discussion and support for our implementation as a Shore value-added server, and Mark McAuliffe for many helpful suggestions regarding both our implementation and this paper.

References

[Cam95] John Campbell, Jan. 1995. Personal communication.

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

[CDF+94] M. Carey, *et al.*. Shoring Up Persistent Applications. *Proc. SIGMOD*, p. 383–394, 1994.

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

[CMR+94] J. B. Cushing, D. Maier, M. Rao, D. Abel, D. Feller, and D. M. DeVaney. Computational Proxies: Modeling Scientific Applications in Object Databases. *Proc. Scientific and Statistical Database Management*, Sept. 1994.

[CPea93] M.A. Chipperfield, C.J. Porter, *et al.* Growth of Data in the Genome Data Base since CCM92 and Methods for Access. *Human Genome Mapping*, p. 3–5, 1993.

[Deu90] O. Deux. The Story of O2. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):91–108, Mar. 1990.

[DLP+93] R. Drach, *et al.* Optimizing Mass Storage Organization and Access for Multi-Dimensional Scientific Data. *Proc. IEEE Symposium on Mass Storage Systems*, Apr. 1993.

[Kea83] M. Kitsuregawa and et al. Application of Hash to Data Base Machine and its Architecture. *New Generation Computing*, 1:62–74, 1983.

[Kim94] W. Kim. UniSQL/X Unified Relational and Object-Oriented Database System. *Proc. SIGMOD*, p. 481, 1994.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore Database System. *CACM*, 34(10):50–63, Oct. 1991.

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

[MN92] C. Mohan and I. Narang. Algorithms for Creating Indexes for Very Large Tables Without Quiescing Updates. *Proc. SIGMOD*, p. 361–370, 1992.

[MN93] C. Mohan and I. Narang. An Efficient and Flexible Method for Archiving a Data Base. *Proc. SIGMOD*, p. 139–146, 1993.

[Moh93a] C. Mohan. A Survey of DBMS Research Issues in Supporting Very Large Tables. *Proc. Foundations of Data Organization and Algorithms*, p. 279–300, 1993.

[Moh93b] C. Mohan. IBM's Relational DBMS Products: Features and Technologies. *Proc. SIGMOD*, p. 445–448, 1993.

[MS90] D. Maier and J. Stein. Development and Implementation of an Object-Oriented DBMS. In S. B. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*, p. 167–185. Morgan-Kaufman, Inc., 1990.

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

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

[Ont94] Ontos, Inc. *Ontos DB Reference Manual*, release 3.0 beta edition, 1994.

[PG88] N. W. Paton and P. M. D. Gray. Identification of Database Objects by Key. *Proc. 2nd Workshop on Object-Oriented Database Systems*, p. 280–285, 1988.

[RZ89] R. Reinsch and M. Zimowski. Method for Restarting a Long-Running, Fault-Tolerant Operation in a Transaction-Oriented Data Base System Without Burdening the System Log. U.S. Patent 4,868,744, IBM, 1989.

[Sha86] L.D. Shapiro. Join Processing in Database Systems with Large Main Memories. *ACM Trans. on Database Systems*, 11(3), Sept. 1986.

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

[Ube94] M. Ubell. The Montage Extensible DataBlade Architecture. *Proc. SIGMOD*, p. 482, 1994.

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

[WCK93] A. Witkowski, F. Cariño, and P. Kostamaa. NCR 3700 — The Next-Generation Industrial Database Computer. *Proc. VLDB*, p. 230–243, 1993.

[WN94] J. L. Wiener and J. F. Naughton. Bulk Loading into an OODB: A Performance Study. *Proc. VLDB*, p. 120–131, 1994.