ALGORITHMS FOR LOADING OBJECT DATABASES

By

Janet Lynn Wiener

A dissertation submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN – MADISON

1995

Abstract

There has been a recent trend to add object functionality to database systems. The resulting technology has resulted in object-oriented databases, object-relational databases, and even relational databases augmented with objects. These object 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. These relationships are expressed in the OODB as object identifiers, which are either not known or not visible 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. This thesis explores different algorithms for dealing with the challenges posed by relationships.

In the initial chapters, we introduce techniques for identifying other objects in the load data file and for dealing with circular and inverse relationships. We evaluate their performance with an analytic model and an implementation within the Shore persistent object repository. The performance results show that eliminating repetitive disk I/O, whether caused by updating objects with inverse relationships or by performing associative accesses to a data structure that is larger than memory, is the critical factor in designing fast load algorithms.

We then turn to the problem of creating relationships between new objects and objects already in the database. We propose using queries within the load data file to identify the existing objects and develop a novel evaluation strategy for the queries that defers evaluating queries so that similar queries can be evaluated together. Our implementation and performance study show that the new strategy scales well with the number of queries and size of the database and provides better performance. Finally, we describe how to make the load algorithm resumable. We conclude that it is important to choose a load algorithm carefully; by incorporating good techniques, our best algorithm achieved an improvement of multiple orders of magnitude over the naive algorithm.

Acknowledgements

This thesis is dedicated to Mark McAuliffe, my partner. Mark not only put up with me at home as I worked furiously in crunch mode on each chapter, he also discussed many of the ideas in this thesis with me, made numerous suggestions, and edited much of my writing. In between crunch times, Mark helped me to be silly, play games, and generally enjoy life. I cannot imagine how different my life would be without him.

The other person most responsible for my completing this thesis is my advisor, Jeff Naughton. Jeff's constant encouragement, probing questions, and focus on the big picture were invaluable. Jeff truly knows how to inspire people to work hard, and how to make them laugh. I have learned a lot about research from him, and I only hope that I can pass some of that knowledge to others. Moreoever, Jeff's willingness to read drafts of my work at the last minute, over and over again, cannot be underappreciated.

Dave Maier deserves many thanks for suggesting that loading into an OODB is a hard problem, and for encouraging me to explore it as a potential research topic. He listened to my early, half-formed, ideas for loading, and has welcomed updates on my progress ever since.

I also want to thank Yannis Ioannidis for supporting me as a graduate student, both financially and intellectually. He made many contributions to my progress, and the idea to expand my initial work on loading into a thesis was his.

David DeWitt has also been a wonderful resource. I would like to thank him and Yannis for reading my thesis, as well as the other members of my thesis committee, Marv Solomon and Jeff Inman. In addition, I am indebted to the Shore staff, Mike Zwilling, C.K. Tan, Nancy Hall, and Dan Schuh for help with my Shore implementation. Odysseas Tsatalos, C. Mohan and S. Sudarshan deserve credit for helping me develop some of the key ideas in this thesis. I credit Laura Haas and Mary Vernon with keeping me in graduate school; they provide much-needed mentoring support at a time when I was very frustrated with my work. Additionally, I thank John Wilkes for my first "real world" research job and for continual encouragement since then.

My friends in Madison are the people to whom I owe my sanity: my first aptmate, Jim Elliott, my officemate Eben Haber (and his CD player!), and my exercise companions, Kristin Bennett, Renée Miller, and Becky Pearlman, who dragged me to the gym when I most needed it. Kurt Brown, Jay Cagle, Shaul Dar, Joey Hellerstein, Manish Mehta, Dan Ross, Jim Skrentny, Charlie Squires, Odysseas Tsatalos and Christina Margeli, Kristin Tufte, and the Madison Israeli Folk Dancers all had a hand in making Madison a wonderful place to spend 6 years, as well.

Finally, I want to thank my family. On the eve of every impending deadline, my mom has always been there with a lot of perspective and the ability to make me laugh. Both my parents, my grandparents, and my sister and brother have been constant sources of love and encouragement. Mark's parents hosted me multiple times as I interviewed for jobs this spring, listening eagerly to my tales of excitement and woe. I am happy to call them my family, too.

Contents

Chapter 1

Introduction

There has been a recent trend to add object functionality to database systems. The resulting technology has resulted in object-oriented databases, object-relational databases, and even relational databases augmented with objects. There are many commercial object-oriented database products today, including Ontos [Ont94], O2 [Deu90], Objectivity [Obj92], ObjectStore [LLOW91], Versant [Ver93], and Gemstone [MS90]. Yet in these systems, loading new data can only be accomplished by creating one object at a time. There are also several object-relational database products on the market, such as Illustra [Ube94] and UniSQL [Kim94]. These systems try to provide better facilities for loading data, but address only some of the challenges posed by object-oriented data. In this thesis, we explore alternative solutions to the problems presented for loading and develop novel algorithms for loading object-oriented and object-relational databases (OODBs) based on them. Portions of this thesis also appear elsewhere [WN94, WN95].

1.1 Motivation

As OODBs 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. 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 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.[1]

- 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. For example, the climate modeling project at Lawrence Livermore National Laboratory uses 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].

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; 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].

A load utility takes an ASCII description of all of the data to be loaded and returns when it has loaded it. For relational systems, a load utility significantly improves performance even when loading only a 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. Additionally, a load utility can group certain operations, such as integrity checks, to dramatically reduce their cost for the load [Moh93a]. A load utility will provide similar advantages for OODBs.

[1] We also know of at least two commercial OODBs — ObjectStore and Illustra — that required a complete dump and reload of all data between two consecutive releases of their systems.

Yet to our knowledge *no commercial OODB has a load utility.*[2] 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. (In a relational database, relationships between tuples are represented by foreign keys, which are created by the user.) Furthermore, OIDs for new objects 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. We use a data structure called an *id map* to map each surrogate to its corresponding OID as the objects are loaded.

- Objects already in the database do have OIDs, but the objects must somehow be identified and their OIDs retrieved.

- 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].

Loading object-relational data involves surmounting the same problems with relationships as loading object-oriented data. While the object-relational systems Illustra [Ube94] and Unisql [Kim94] provide a load utility, they only provide (not necessarily optimal) solutions to some of the problems posed by relationships. More specifically, they can create relationships among new objects and they allow

[2] Objectivity has something it calls a load utility, however, it can only load data that already contains system-specific 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].

forward references in the data file. However, neither system provides bidirectional relationships, and neither can connect new objects to objects already in the database during a load.

1.2 Contributions of the thesis

This thesis makes several contributions to the development of fast load algorithms. First, we provide solutions, usually several, to all of the problems posed by relationships, above. We know of no other work that addresses loading for OODBs. 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.

While load utilities exist for hierarchical databases, the hierarchical data model does not support the kind of arbitrary and complex relationships that make loading into an OODB a challenge. Load utilities also exist for network databases, which do face the same kinds of problems posed by relationships for OODBs, but we were unable to find any description of the techniques or algorithms used by such utilities.

Second, we explore the space of possible load algorithms with extensive performance studies of their behavior. We use both an analytic model and an implementation within the Shore persistent object repository [CDF+94] to examine a wide variety of conditions that affect the loading speed, and are able to recommend a single algorithm for OODB load utility writers to implement.

Third, we also present a simple but efficient method of checkpointing our best load algorithm and resuming from any checkpoint, so that a given load may be halted and resumed any number of times without losing a significant amount of work; this can be critical when loading data takes hours or days and a system crash or important query might interrupt the load at any time.

1.3 Organization of the thesis

The remainder of this thesis is organized as follows. Chapter 2 presents the experimental configuration we used to evaluate all of our techniques and algorithms. In Chapter 3, we explore different techiques for handling forward references in the data file and different ways of creating inverse relationships, while modeling the id map as an in-memory table. In Chapter 4, we examine the implications of keeping the id map in memory, and recommend alternative representations and subsequent ways to access them. In Chapter 5 we propose using queries in the data file to identify objects in the database and create relationships to them. We then investigate query evaluation strategies for retrieving and updating the existing objects. All three of these chapters include a comprehensive performance study of the proposed techniques, and preliminary conclusions about which techniques a load algorithm should use. We show how to make the load algorithm resumable in Chapter 6. We show both what to save in a restart checkpoint of the algorithm and how to resume from one of these checkpoints. We also survey the system features that Shore, or a similar OODB, must provide to support resumable transactions. Finally, we summarize the conclusions of this thesis, recommend a complete load algorithm, and outline future work in Chapter 7.

Chapter 2

Experimental configuration

In each of Chapters 3, 4, and 5, we present a performance study of the techniques and algorithms we introduce in that chapter. In this chapter, we describe the characteristics of the data we loaded, and the specifics of our implementation.

2.1 Data characteristics

For each performance experiment, we created a database from a description of the data in a data file. All of the created objects were 200 bytes. The schema for each object contained ten bidirectional relationships. We listed five relationships explicitly with each object in the data file. The other five relationships were their inverses. We varied the number of objects to control the size of the database; there were 5,000 objects per megabyte (Mb) of database. The ASCII data files were approximately half as large as the data set they described, e.g, the data file for the 10 Mb data set was 4.1 Mb.

In Chapter 5, when we create new objects with relationships to objects already in the database, four of the five explicit relationships were always shared with another new object. The fifth relationship was either to an existing object or to a new object. Whenever the relationship was shared with an existing object, the existing object was updated, causing it to grow larger than 200 bytes.

We varied the locality of reference among the objects from no locality to high locality. 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 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. In the data sets with no locality of reference, all relationships are to objects chosen at random.

2.2 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. 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 is needed by some of the techniques we explore.

Second, Shore provides both physical and logical OIDs. A physical OID is the physical (usually disk) address of the object. A logical OID uses a level of indirection to retrieve the physical address, thereby allowing the physical address to change without affecting the OID. Because Shore provides both, we were able to compare Shore's performance using physical versus logical OIDs, and to test techniques that require logical OIDs. Shore uses a logical OID index to map from logical OIDs to physical OIDs; this index is stored in the database.

We ran all of the algorithms on a Hewlett-Packard 9000/720 with 32 Mb 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 volume was a raw 2 gigabyte Seagate ST-12400N disk controlled exclusively by Shore. The data file resided on a separate disk in the 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.

Chapter 3

Alternative load algorithms

3.1 Introduction

As described in Chapter 1, our goal is develop efficient algorithms for loading object databases. More specifically, we need to address the four challenges posed by relationships between objects. In this chapter, we address the challenges of resolving surrogate identifiers to OIDs and of creating inverse relationships efficiently.

We examine several techniques for dealing with circular and inverse relationships and introduce seven algorithms based on these techniques. We evaluate the performance of these algorithms with an analytic model and an implementation on top of the Shore persistent object repository [CDF$^+$94]. We use the analytic model to explore a wide range of load file and system configurations. The implementation results both validate the analytic model and highlight several key advantages and disadvantages of using logical OIDs. Furthermore, our performance results show that one algorithm almost always outperforms all of the others. We first describe an example database that we will use both in this chapter and the next chapter, and then present our techniques, algorithms, and performance study.

3.2 Loading example

3.2.1 Example database schema

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].

```
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 integer 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.

3.2.2 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;
    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.

The data file is an ASCII text file describing the objects to be loaded[1]. We illustrate the data file format in Figure 2. Although we developed it for the Moose data model [WI93], it fits a generic OO data

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

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 type with a type header. However, the types may appear in any order, and objects of the same type may be listed in different parts of the data file by repeating the type header. Each type header contains the type name, attributes, and relationships. Each new object is described by a surrogate identifier and a list of values for the attributes and relationships described in the type header. If an attribute or relationship of the type is not specified in the type header, then the object gets a null value for that relationship. 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 surrogate into the OID assigned (by the system) to the corresponding object.

3.3 Techniques for handling relationships

3.3.1 Mapping surrogates to OIDs

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 algorithms.

All of the algorithms use an *id map* to map surrogates to the database's OIDs. As each object is created and assigned an OID, 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. Figure 3 shows the id map built for the Experiment data file.

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

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

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 of the surrogates to OIDs at that time. Surrogates that refer to objects described further down in the data 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 other information indicating where and how to store the relationship (not shown). Figure 4 contains the todo list created for the Experiment data file by the resolve-early algorithms. The todo list is read and the updates are 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 to OIDs 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, because the OID does not depend on a physical representation of the object. 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.

3.3.3 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
101	OID4
201	OID4
103	OID5
202	OID5
101	OID6
203	OID6
102	OID7
202	OID7

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 other information about the relationship (not shown). 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 order of the objects to update. To avoid a large number of repetitive and random disk reads, we first sort the inverse todo list so that updates to the same object are grouped together, and the updates are performed in an optimal order.[2] The sort key (and the optimal order) depends on the technique

[2] We predicted that without sorting the inverse todo list, the performance would be similar to that of the immediate

used for handling forward references, as follows.

For the two-pass algorithms, we already update the objects as we read the data file the second time. By sorting the inverse updates into the same order as the objects appear in the data file, we can perform the inverse updates concurrently with reading the data file. We use a sequence counter for the objects as the sort key; the sequence numbers are generated and stored in the id map as the data file is read the first time. In Shore, logical OIDs are assigned in sequential order, so the OIDs serve as a sequence counter. However, a separate counter works equally well, although it does increase the size of the id map entries.

For the resolve-early algorithms, we already have a todo list containing updates to objects. Updates for entries in the todo list appear in data file order, so sorting by sequence number (as for the two-pass algorithms) allows the todo and inverse todo updates to be read and applied concurrently. Alternately, both the todo and inverse todo updates could be sorted by the same criterion. If the sequence order of the objects did not resemble the clustering order of the created objects, for example, sorting both lists by physical location might lead to more efficient updates. We assume that at the time each object is created and pinned in the buffer pool it is possible to retrieve its physical address. The physical address can then be stored in the id map, along with the object's surrogate and OID, to allow sorting by physical location.

For the assign-early algorithms, processing the inverse todo updates is the only step left after reading the data file. Therefore, sorting the updates by physical location yields the best performance. Each object's physical address is retrieved when the object is created and stored in the id map. When the surrogate for the object to update is converted to an OID, the physical address is also retrieved, and used as the sort key.

Sorting is done in two phases. First, for each inverse todo list entry, we look up the OID and sort key (sequence number or physical address) of the object to be updated in the id map and add it to the entry. In this phase, we read the inverse todo list in chunks and create sorted runs. In the second phase, we merge the sorted runs. On the final merge pass, we perform all of the updates. Figure 6 shows the inverse todo list from Figure 5 after sorting.

We note that both of our inverse update techniques ensure the integrity of the inverse relationship,

---

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
OID1	OID4
OID1	OID6
OID2	OID7
OID3	OID5
OID8	OID4
OID9	OID7
OID9	OID5
OID10	OID6

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

and could be used for other integrity checks that are not part of an inverse relationship.

3.3.4 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 more frequent intervals[3] 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 of the database). In this example, we were able to clear two entries, or one-third of the total entries, from the inverse todo list.

[3]We cleared the todo lists whenever one-quarter of the buffer pool pages had been replaced. The number of entries that can be cleared depends on how many new objects have been created since each entry was generated, and the number of pages in the buffer pool containing new objects is a reasonable measure.

Surrogate for object to update	OID to store
201	OID4
202	OID5
101	OID6
203	OID6

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

3.4 Loading algorithms

We now propose seven algorithms for loading the database from a data file. In all of the algorithms, we read the data file and create the objects described in it. Each algorithm uses an id map, and a different combination of the techniques presented in Section 3.3. The algorithms span all of the viable combinations of resolving surrogates and handling inverse relationships.

Forward references	Inverse Relationships	
	Immediate updates	Inverse-sort
Two-pass	Naive	2pass-invsort 2pass-invsort^{++}
Resolve-early	×	Resolve-early-invsort Resolve-clear-invclear
Assign-early	×	Assign-early-invsort Assign-early-invclear

Table 1: Combining the techniques into algorithms.

We illustrate how the techniques are combined into algorithms in Table 1, and explain the algorithms in more detail below. Each row of the table corresponds to one technique for handling forward references in the data file, and each column corresponds to a technique for creating inverse relationships. There are no algorithms that combine either resolve-early or assign-early with immediate updates; at the time when resolve-early and assign-early process the surrogate for a relationship, the inverse object may not yet exist and so updates cannot be applied to it.

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 of the objects (ignoring relationships), and builds the id map. On the second pass, the objects are filled in with the relationship data. Updates for inverse relationships are performed as they are encountered on the second pass.

2pass-invsort: 2pass-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.

2pass-invsort^{++}: 2pass-invsort^{++} is an optimization of 2pass-invsort that requires logical OIDs. During the first pass of 2pass-invsort, the objects are created simply to obtain their OIDs; they are not filled in until the second pass. During the first pass of 2pass-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. An important advantage of this algorithm is that the objects are never updated; they are completely filled in when they are created. Therefore, the objects never change size, which they may in the other algorithms. Since changing the sizes of objects often severely impacts the clustering of the objects, this advantage is quite significant.

Resolve-early-invsort: Resolve-early-invsort employs the resolve-early technique for surrogates and inverse-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 the inverse-sort technique for inverse relationships. It makes one pass over the data file to create the objects, then sorts the inverse todo list and makes one pass over the database to perform the updates dictated by the inverse todo entries.

Resolve-clear-invclear: Resolve-clear-invclear is similar to resolve-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.

3.5 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 during the load. Except for the assign-early algorithms, each algorithm had an additional cost for resolving surrogates to OIDs, and all of the algorithms had additional costs for implementing inverse relationships.

We modeled locality of reference among the objects using the parameters x and y. More specifically, x% of the relationships from a given object are to objects that are within y% of the database from it. The remaining (100-x)% are to randomly chosen objects. When x and y are 0, there is no locality of reference.

We now describe the cost formulas used in the analytic model. We present the (much simpler) formulas for when the id map fits in memory, although the model also estimates the cost of paging the id map when it does not fit. We used 8 byte OIDs (this is the size used by Shore), so each id map entry is 12 bytes; the clearing algorithms' id map entries have an additional 4 bytes for the page numbers we use to check if an object is in the buffer pool. The parameters used in the cost of each algorithm are listed in Table 2.

The cost for each algorithm is now as follows:

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

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 map entry (bytes)
S_{idmap}	size of id map (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	% relationships 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 2: Parameters of the cost model.

$$2pass{-}invsort = 2 * P_{file} + 3 * P_{db} + 4 * P_{invtodo} \quad (2)$$

$$P_{invtodo} = N_{obj} * N_{invrel}$$

$$2pass{-}invsort{+}{+} = 2 * P_{file} + P_{db} + 4 * P_{invtodo} \quad (3)$$

$$Resolve{-}early{-}invsort = P_{file} + 3 * P_{db} + 2 * P_{todo} + 4 * P_{invtodo} \quad (4)$$

$$P_{todo} = N_{obj} * N_{rel} * 0.5$$

$$Assign{-}early{-}invsort = P_{file} + 3 * P_{db} + 4 * P_{invtodo} \quad (5)$$

$$Resolve{-}clear{-}invclear = P_{file} + 3 * P_{db} + 2 * P_{clrtodo} + 4 * P_{clrinvtodo} \quad (6)$$

$$P_{clrtodo} = N_{obj} * N_{rel} * Prob_{notcleared}$$

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

$$z = \frac{S_{mem} - S_{idmap}}{S_{db}}$$

$$P_{clrinvtodo} = N_{obj} * N_{invrel} * Prob_{invnotcleared}$$

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

$$Assign{-}early{-}invclear = P_{file} + 3 * P_{db} + 4 * P_{clrinvtodo} \quad (7)$$

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 of 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.

2pass-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 requires an extra merge pass, the cost is $6 * P_{invtodo}$. 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. The cost for 2pass-invsort^{++} is the same as for 2pass-invsort, except that it does not need to update the database after creating it.

Resolve-early-invsort reads the data file only once. However, it incurs the cost of writing and reading both a todo list and a inverse todo list. The inverse todo list cost is the same as for 2pass-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 of the potential number of entries.

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 2pass-invsort.

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 map, no entries are cleared. In between, the percentage of the database in the buffer pool is used in conjunction with the locality of reference 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, resolve-clear-invclear and assign-early-invclear are affected by locality, and by the size of the buffer pool, while 2pass-invsort, 2pass-invsort^{++}, resolve-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 of the other algorithms, the cost is linear in the number of objects when the id map fits in memory. When the id map does not fit, the cost is also a multiple of the number of objects and the number of relationships.

3.6 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 high 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 todo and inverse todo lists, 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 of magnitude worse. Figure 9 shows the differences in performance among the remaining algorithms. 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, resolve-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. 2pass-invsort^{++} dominates the non-clearing algorithms, and even beats the clearing algorithms at some points. 2pass-invsort performs comparably to resolve-early-invsort; although 2pass-invsort does not

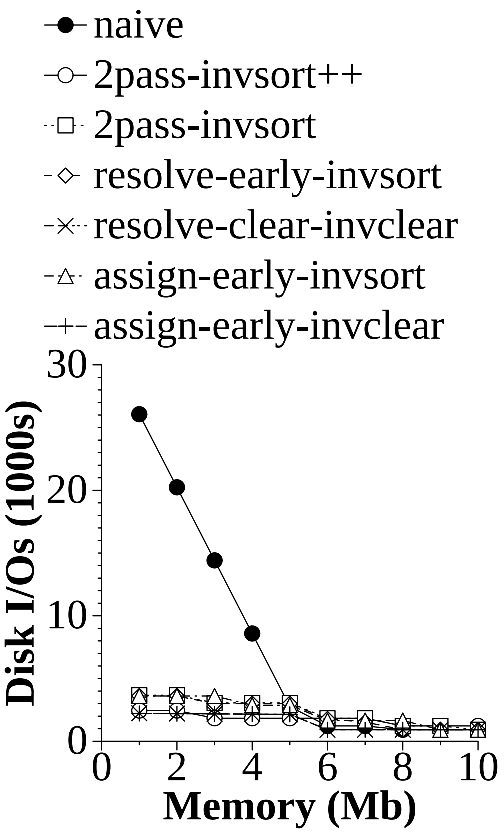


Figure 8: 5 Mb database with high locality.

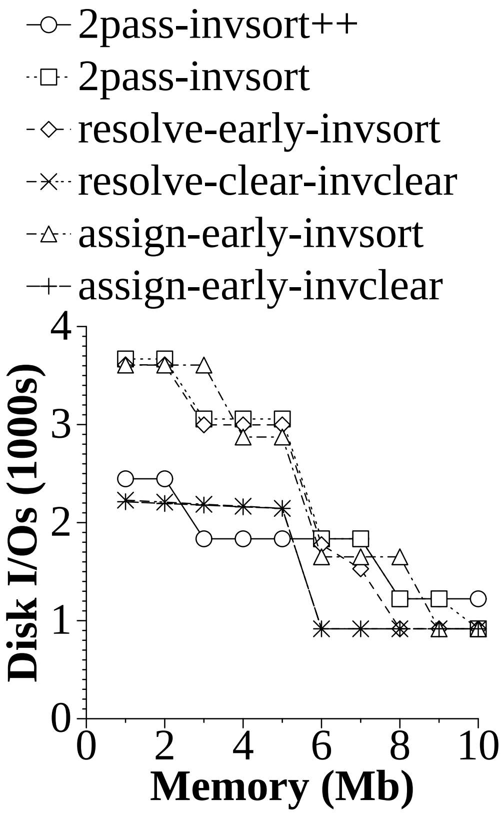


Figure 9: 5 Mb database with high locality, without naive.

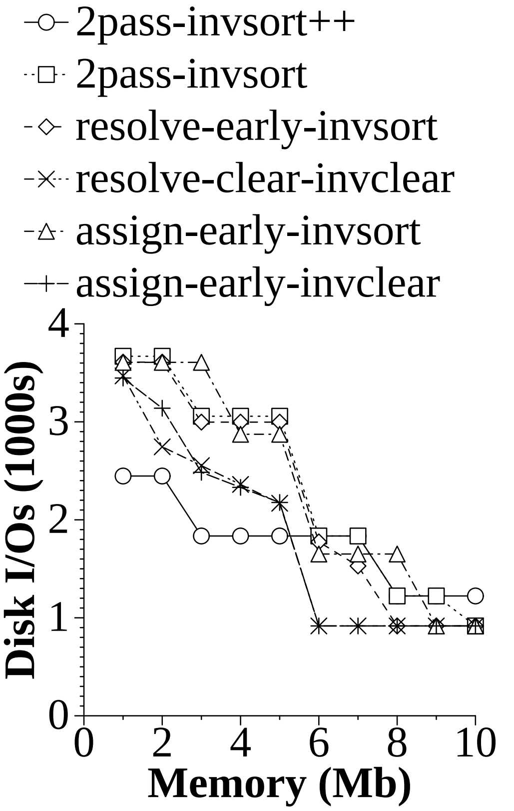


Figure 10: 5 Mb database with no locality.

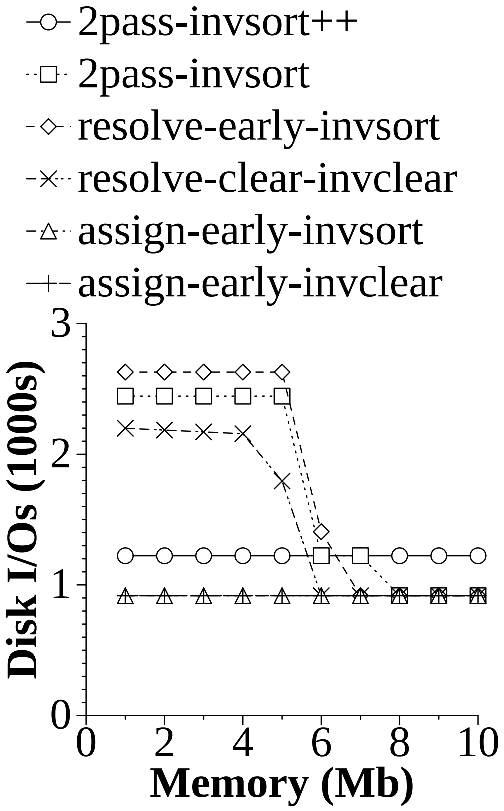


Figure 11: 5 Mb database with 10 relationships and no inverses.

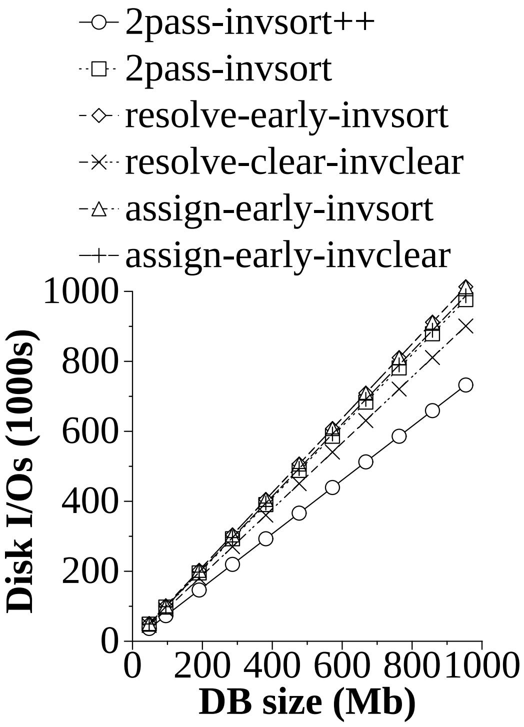


Figure 12: Scaling database size to 1 Gb, with 10% in memory.

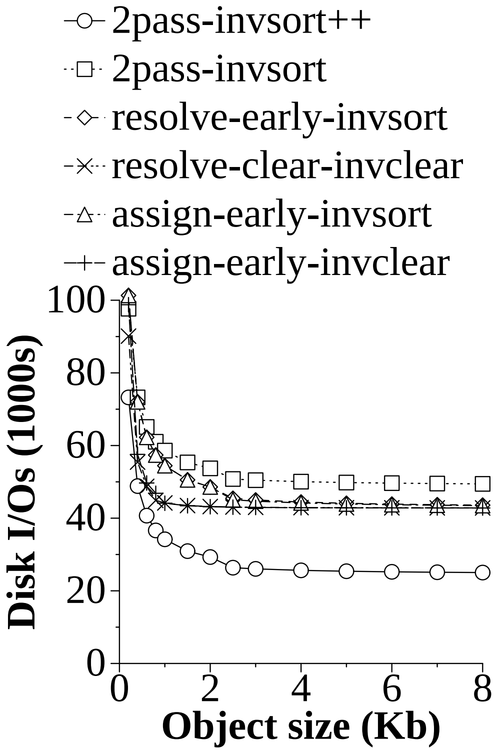


Figure 13: Scaling object size for 5 Mb database.

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, 2pass-invsort^{++} outperforms the clearing algorithms until approximately the entire 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 215,000 I/O's, while 2pass-invsort^{++} performs merely 2,450 and resolve-clear-invclear only 3,500 I/Os. For the remainder of the analytical model experiments, all of the data files had a high locality of reference.

In some cases, such as when an OODB is dumped to a file and then reloaded, it is possible to dump both halves of an inverse relationship. That is, instead of storing only the fact that A has an inverse relationship with B in the data file, and letting the load algorithm take care of storing the relationship from B to A, it is possible to indicate both the relationship from A to B and the relationship from B to A explicitly in the data file. That way, the load algorithm does not need to perform any inverse updates. Also, in some schemas, there are no inverse relationships. We therefore 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 of 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 a second (update) pass over the database. Naive and 2pass-invsort appear as a single line, since they differ only in their handling of inverse updates. Resolve-clear-invclear performs slightly better than 2pass-invsort because the cost of writing and reading the cleared todo list is less than that of rereading the data file; resolve-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 gigabyte (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 of the other parameters are the same as before. We verify with this experiment that the relative performance of the non-clearing 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. The clearing algorithms no longer perform as well, because a much smaller percentage of the todo and inverse todo lists stays in memory long enough to be cleared. Therefore, 2pass-invsort^{++} is always the best algorithm for loading more than 50 Mb.

For the final experiment, shown in Figure 13, we held the database size constant and varied the object size from 200 bytes to 8 kilobytes (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 modeled 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 map shrinks and so more of the database fits in the buffer pool. Second, the absolute number of relationships declines, and so the sizes of the todo and inverse todo lists also decline.

3.6.1 Discussion

According to the analytic model, the relative ranking of the algorithms is 2pass-invsort++, followed closely by assign-early-invclear and resolve-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. These three algorithms are then followed by assign-early-invsort and then resolve-early-invsort and 2pass-invsort; this ranking is fairly consistent regardless of the locality of reference in the data 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 does not 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 size of the data file tends to be the same order of magnitude as the database it describes.

All of the algorithms cost significantly less when there are no inverse relationships. However, we have already noted that most of the commercial OODB systems (Ontos, Objectivity, Versant, and 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 direction 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 of 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 complete analytic model results for loads when the id map does not fit in the buffer pool, we note that the I/O cost greatly increases: we do an insert in the id map 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 map page, the costs skyrocket. For example, the predicted cost for 2pass-invsort^{++} for a 5 Mb database is only 2,450 I/Os with 0.5 Mb of memory, which just barely holds the id map, but 279,000 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 map as the minimum amount of memory that should be made available to these load algorithms. 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 map of only 2 Mb. We remove this restriction in Chapter 4 when we propose alternate representations for the id map.

3.7 Implementation experiments and results

We ran implementation experiments to load a database with 5 Mb of data. (In Chapter 4 we will load up to 500 Mb.) Due to metadata overhead and Shore's logical OID index, the databases created are actually 7 Mb and hence first fit in the buffer pool at 7 Mb. In the analytic model, if the id map did not fit in the buffer pool, we counted I/Os for accessing it. In the implementation for this chapter, the id map is an open addressing hash table that we allocated in transient memory, and assumed that it would always fit. We revisit this assumption in Chapter 4.

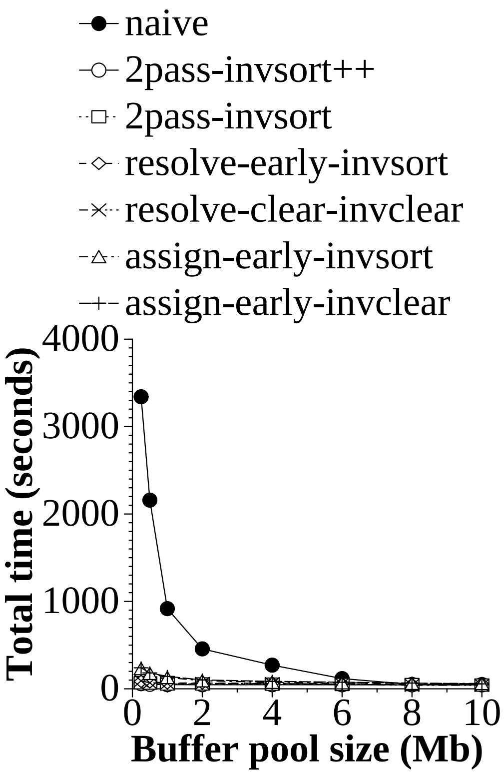


Figure 14: 5 Mb database with high locality.

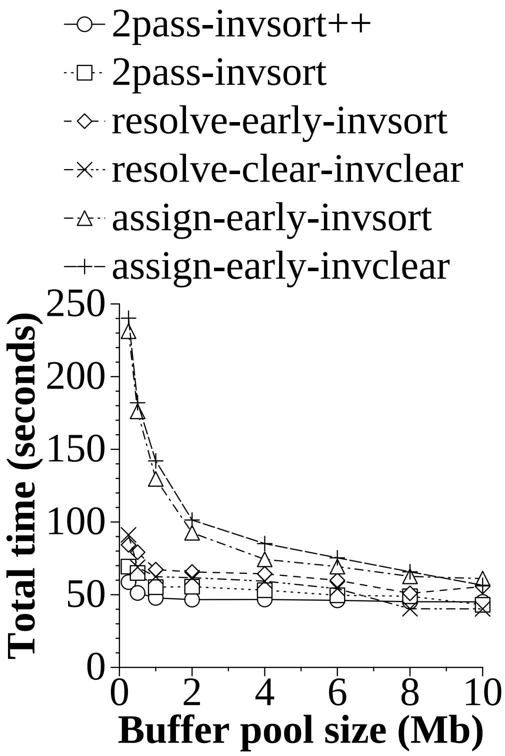


Figure 15: 5 Mb database with high locality, without naive.

In the first experiment, shown in Figure 14, we loaded a 5 Mb database with a high locality of reference. As predicted by the analytic model, the load times for the naive algorithm overwhelm the times for the other algorithms 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 resolve-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.

In all cases, 2pass-invsort^{++} is the fastest algorithm. As the buffer pool grows to hold nearly the

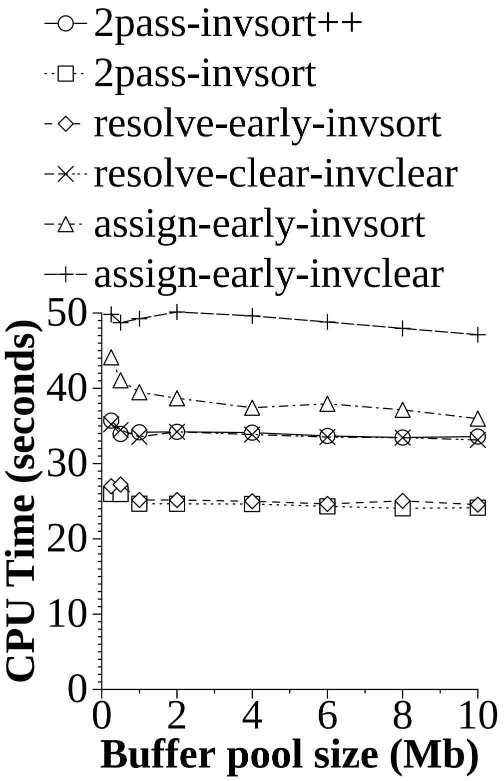


Figure 16: 5 Mb database: CPU time.

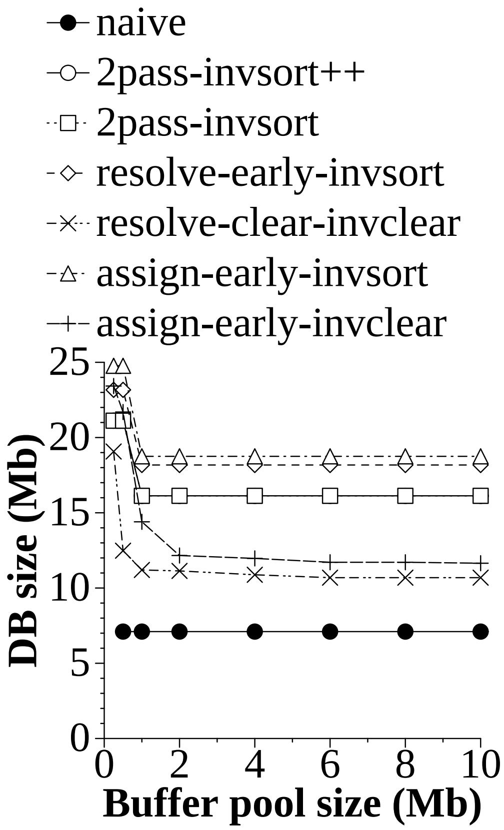


Figure 17: 5 Mb database: Disk space used by database and todo lists.

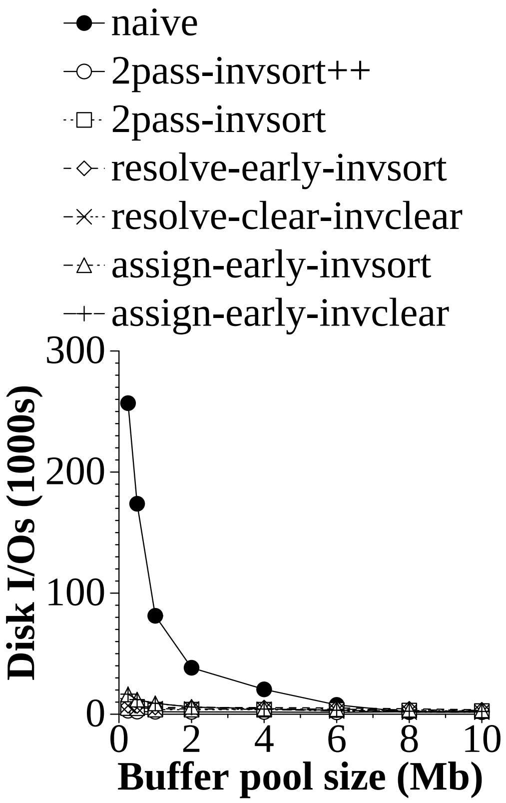


Figure 18: 5 Mb database: Disk I/Os.

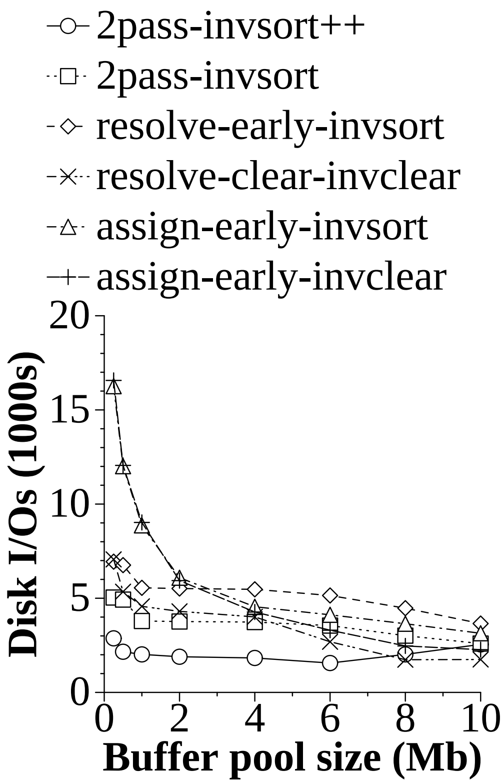


Figure 19: 5 Mb database: Disk I/O, without naive.

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 resolve-clear-invclear. However, the improvement is not as dramatic as the analytic model predicts, and hence 2pass-invsort^{++} is still better. The difference in predicted versus actual improvement 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 of each algorithm, 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. This 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 space 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 map, 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 exhibited by the assign-early algorithms, due to the logical OID index, we note that the actual I/O cost of each algorithm is quite close to the I/O cost predicted by our analytic model. For example, in Figure 9, we predicted 2447 I/Os for 2pass-invsort^{++} with 1 Mb of memory. In our experiment, 2pass-invsort^{++} took 2159 I/Os, which is less than a 15% deviation.

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 over an hour to complete the load with 1 Mb of memory, and 35 minutes with 4 Mb. All of the other algorithms, in contrast, take 1 to 2 minutes. The relative performance of the algorithms is similar to that with high locality, but the assign-early algorithms pay an even greater penalty for inserting into the logical OID index out of order.

We therefore ran some experiments to see how the algorithms perform with physical OIDs. Figure 22 show the results of these experiments. 2pass-invsort^{++} and assign-early depend on logical OIDs and

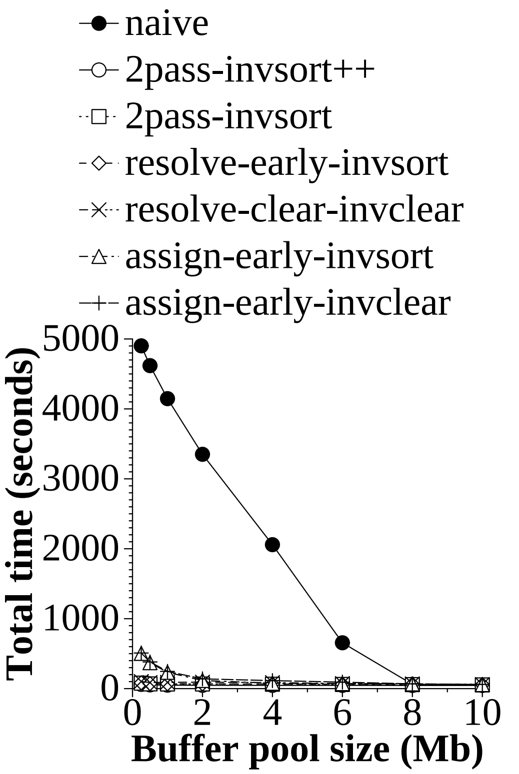


Figure 20: 5 Mb database with no locality.

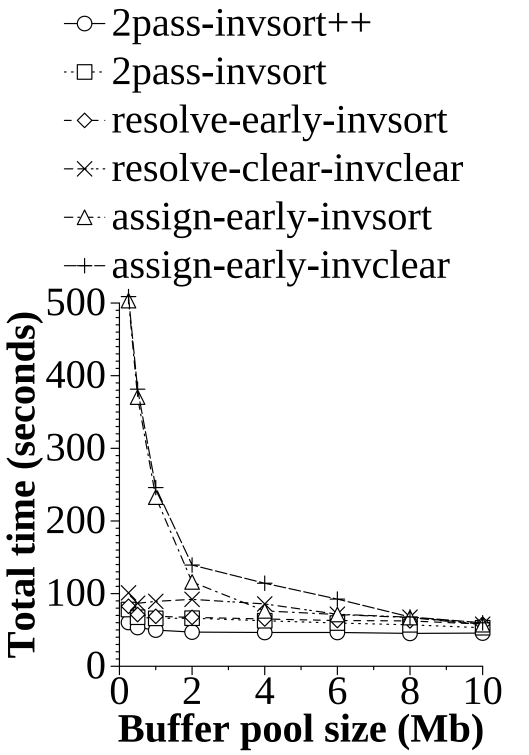


Figure 21: 5 Mb database with no locality, without naive.

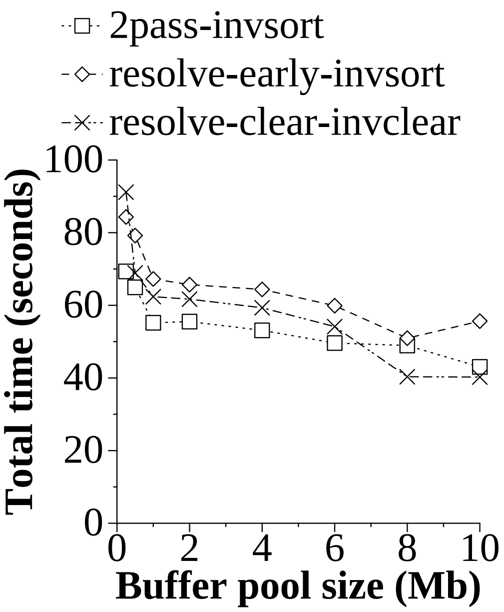


Figure 22: 5 Mb database with physical OIDs.

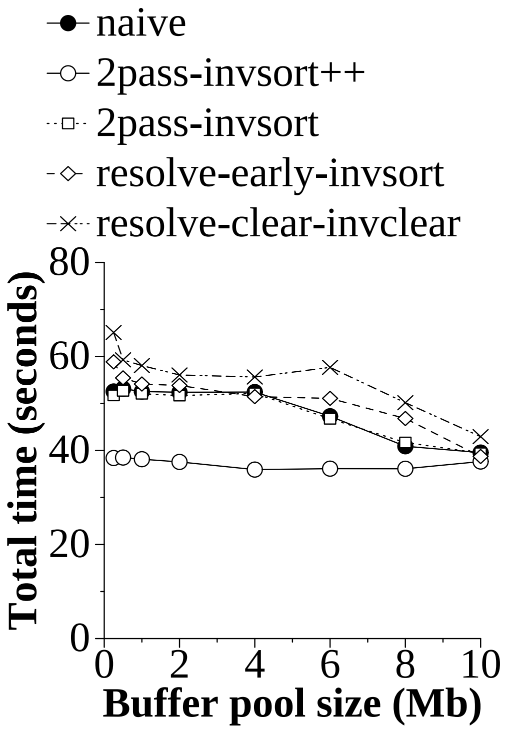


Figure 23: 5 Mb database without inverse relationships.

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.[4] The size of the objects thus grew from 200 bytes to 280 bytes to store the same information. Where 2pass-invsort^{++} used 16 Mb of disk space with logical OIDs, with physical OIDs it used 19 Mb. The database itself grew from 7 Mb (including the logical OID index) to 7.9 Mb. 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.

For the final experiment in this chapter, we ran tests to load objects containing 10 relationships but no inverse relationships. We present the results in Figure 23. We found that the analytic model was correct: all of the algorithms run much faster. For example, 2pass-invsort^{++} loaded the 5 Mb database with 0.5 Mb of Memory in 38 seconds; naive took 52 and resolve-early-invsort ran in 55 seconds. The corresponding times to load the same database with inverse relationships were 51, 79, and 2158 seconds,

[4]Shore logical OIDs have two components, an 8-byte volume id and an 8-byte "serial number," which is unique only within a single volume. Shore stores only the serial number inside objects, because most objects share relationships with other objects on the same volume. When an object references an object on a different volume, a local serial number is used as an alias for the remote volume and serial number pair. Shore physical OIDs have fields for a volume id, file id, page id, and slot number, for a total of 12 bytes.

respectively.

3.7.1 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 does 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 2pass-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 resolve-clear-invclear would beat 2pass-invsort^{++} with high locality, the CPU time involved in clearing made up for the savings in disk I/Os, and 2pass-invsort^{++} is faster.

2pass-invsort^{++} is clearly the best choice according to the implementation results. Its superior performance is due to the fact that objects are never updated; they are completely filled in when they are created during the last phase of the algorithm. This results in 50% fewer disk I/Os and 20-30% better overall performance.

Of the other algorithms, both 2pass-invsort and resolve-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 resolve-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 resolve-clear-invclear. We therefore recommend that users implement 2pass-invsort^{++} if pre-assigning of OIDs is possible, and either 2pass-invsort or resolve-early-invsort if not.

3.8 Conclusions

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 repetitive 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 2pass-invsort^{++} if logical OIDs are available, and either 2pass-invsort or resolve-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 individual **insert** or **new** statements, since inverse updates cannot be batched.

Our primary conclusions from this study are that a two-pass algorithm, using pre-assigned logical OIDs, is the best technique for handling forward references, and that the inverse-sort technique is best for creating inverse relationships. In subsequent chapters, we will begin with these techniques in our algorithm, and extend the algorithm to address the challenges of translating surrogates to OIDs when the id map is large and of creating relationships to objects already in the database.

Chapter 4

The partitioned-list approach

4.1 Introduction

In Chapter 3, we presented techniques for creating relationships in OODBs. In addition to dealing with relationships, a good load algorithm 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, as we showed in Chapter 3. Also, a load utility can decrease client-server interaction to speed up 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, the load algorithms proposed in Chapter 3 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 the study in Chapter 3, 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 chapter we propose the partitioned-list algorithm, which, unlike our previous algorithms, provides good performance even for this range of problem sizes.

The third class of data sets is the most challenging. 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 Gb 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 Mb.

- 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 Gb 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 Mb.

In this chapter, we propose a new algorithm, the partitioned-list algorithm, that extends the 2pass-invsort^{++} algorithm we recommended in Chapter 3 to handle large data sets. We will use the same loading example we presented in Section 3.2 to illustrate the new algorithm. After describing the new algorithm, we present a performance study comparing it to simpler modifications of the 2pass-invsort^{++} algorithm.

4.2 Load algorithms

In this section, we present each of the load algorithms we study in this chapter. The first two algorithms are the naive and 2pass-invsort^{++} algorithms from Chapter 3. Then we describe two simple modifications to the id map data structure which allow it to exceed memory size but do not alter the basic load algorithm. We use these modifications to adapt the 2pass-invsort^{++} algorithm for large data sets. Finally, we introduce a new algorithm, the *partitioned-list* algorithm, which is optimized for the case when the id map is significantly larger than the amount of memory available.

4.2.1 Naive algorithm

The naive algorithm reads the data file twice, so that forward references in the data file can be handled correctly. Updates due to inverse relationships are performed as they are encountered; there are no todo lists. If the inverse relationship is many-to-one, or many-to-many, then updating the inverse object will change its size. The id map is an in-memory open addressing hash table hashed on surrogate and is kept separate from the database buffer pool.

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

The 2pass-invsort^{++} algorithm also reads the data file twice. However, it uses an inverse todo list to handle updates caused by bidirectional relationships. The algorithm also takes advantage of Shore's logical OIDs to pre-assign OIDs to objects on the first pass while it fills in the id map and creates the inverse todo list. Then it sorts the inverse todo list so that the updates are in the same order as the objects appear in the data file. On the second pass over the data file, the updates are merged into the object creations and each object is completely filled in when it is created. As in the naive algorithm, the id map is an in-memory open addressing hash table. The remaining algorithms are all modifications of the basic 2pass-invsort^{++} algorithm.

4.2.3 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.

We 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.

4.2.4 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.

4.2.5 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 repetitive and random disk I/O. The goal of this algorithm is to eliminate the repetitive I/O.

In the 2pass-invsort^{++} algorithm, 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, to retrieve the OIDs for each relationship. These lookups were interspersed with reading the data file and creating objects. However, by separating the lookups from reading the data file, into a different todo list, these lookups can also be performed as a join between the todo list and 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 build 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 24: Id map with 2 partitions.

Partition 0	
OID for object to update	Surrogate for object to store
OID5	202
OID7	102
OID7	202

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

Figure 25: Todo list with 2 partitions.

Partition 0	
Surrogate for object to update	OID to store
202	OID5
102	OID7
202	OID7

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

Figure 26: Inverse todo list 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 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 other information (not shown) indicating where and how 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 other information (not shown) indicating where and how 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; one page of each partition of the id map, todo list, and inverse todo list must fit in the buffer pool at the same time. Figure 25 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 5, except that they are now partitioned as shown in Figures 24 and 26. In this example, the hash partitioning function is surrogate mod 2.

Note that although the todo and inverse todo lists are conceptually different lists, their entries are very similar. More specifically, each inverse todo entry shares a surrogate and OID with a todo entry; only the information (not shown) about how to update each object is different. These two lists could instead be stored as one list, where each entry has four fields: the surrogate, the OID, and the update information for each of the two objects. (If the relationship is unidirectional, then the update information for the inverse object is left blank.) However, once the entries are joined with the id map, two separate update entries must be created, since each update will have a different sort key.

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 other information for storing it (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 other information (from the inverse todo entry).

    Note that all of the 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 data file sequence order, which in Shore is the same as logical OID order) 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.

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 is necessary for all of the algorithms.

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 no longer any repetitive I/O being performed on behalf of the algorithm, because there is no longer random access to any load data structure.

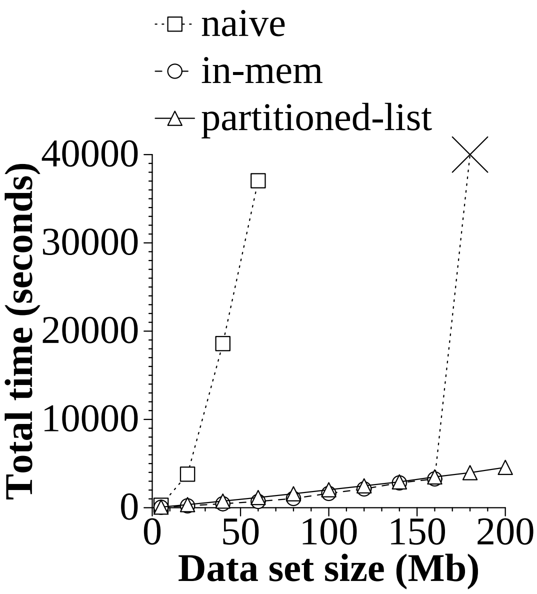
4.3 Performance results

We implemented all of the algorithms in Shore and ran a series of performance experiments to show how quickly (or slowly!) each algorithm loads different size data sets. In this section, we will use the names in Table 3 to refer to each algorithm.

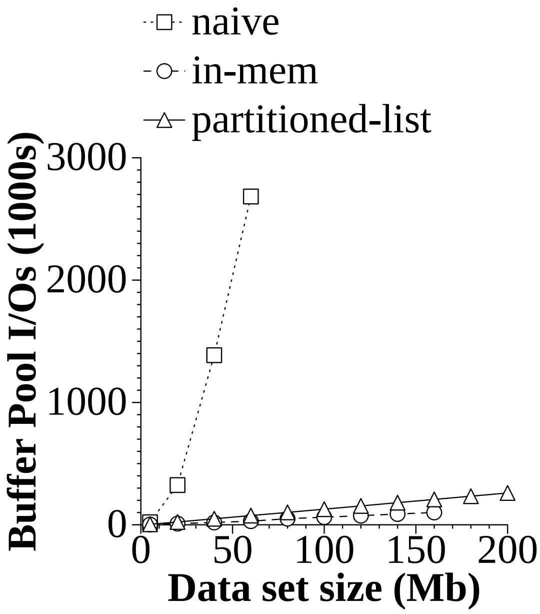
Algorithm	Major data structures	Described in
naive	in-memory hash table id map immediate inverse updates	Section 4.2.1
in-mem	in-memory hash table id map inverse todo list	Section 4.2.2
btree	B^+-tree id map inverse todo list	Section 4.2.3
cache	B^+-tree plus cache id map inverse todo list	Section 4.2.4
partitioned -list	partitioned list id map todo and inverse todo lists	Section 4.2.5

Table 3: Algorithm names used in performance graphs.

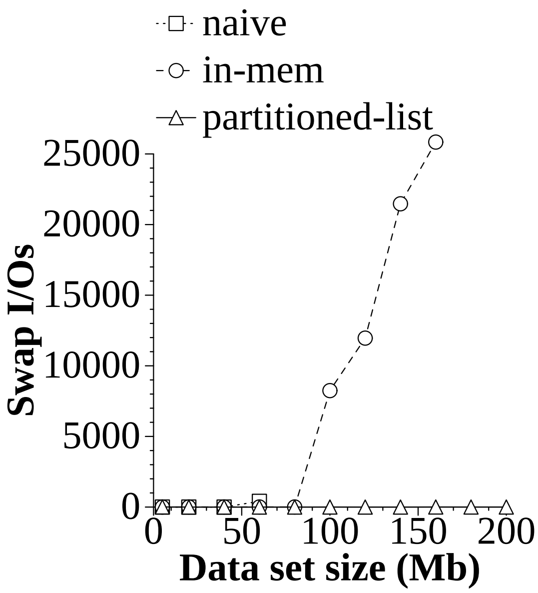
4.3.1 Comparing algorithms with different classes of data set sizes



(a)



(b)



(c)

Figure 27: 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 uses 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 27(a) shows the total (wall clock) time for the naive, in-mem, and partitioned-list algorithms to load data sets of 2 to 200 Mb. Figures 27(b) and 27(c) show the number of I/Os that were performed by the same experiments: 27(b) depicts the number of I/Os in the buffer pool, and 27(c) depicts the number of I/Os performed as virtual memory is swapped in and out of physical memory.[1]

Naive performs comparably to in-mem on the 2 Mb data set; nearly the entire data set fits in the 4 Mb buffer pool and 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 total time and the buffer pool I/Os. At 5 Mb, naive is already taking 5 times as long to load: it takes 277 vs. 49 seconds for in-mem. 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 12 minutes. Naive is clearly unsuitable for loading once the

[1] We measured the virtual memory page swaps with the *getrusage* system call; Shore provided the buffer pool I/O statistics.

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 completed less than 10% of step 1 of the algorithm. (In-mem loaded the entire 160 Mb data set in under 1 hour.) 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 [DKO$^+$84] 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 and read.

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

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 28 shows the total 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

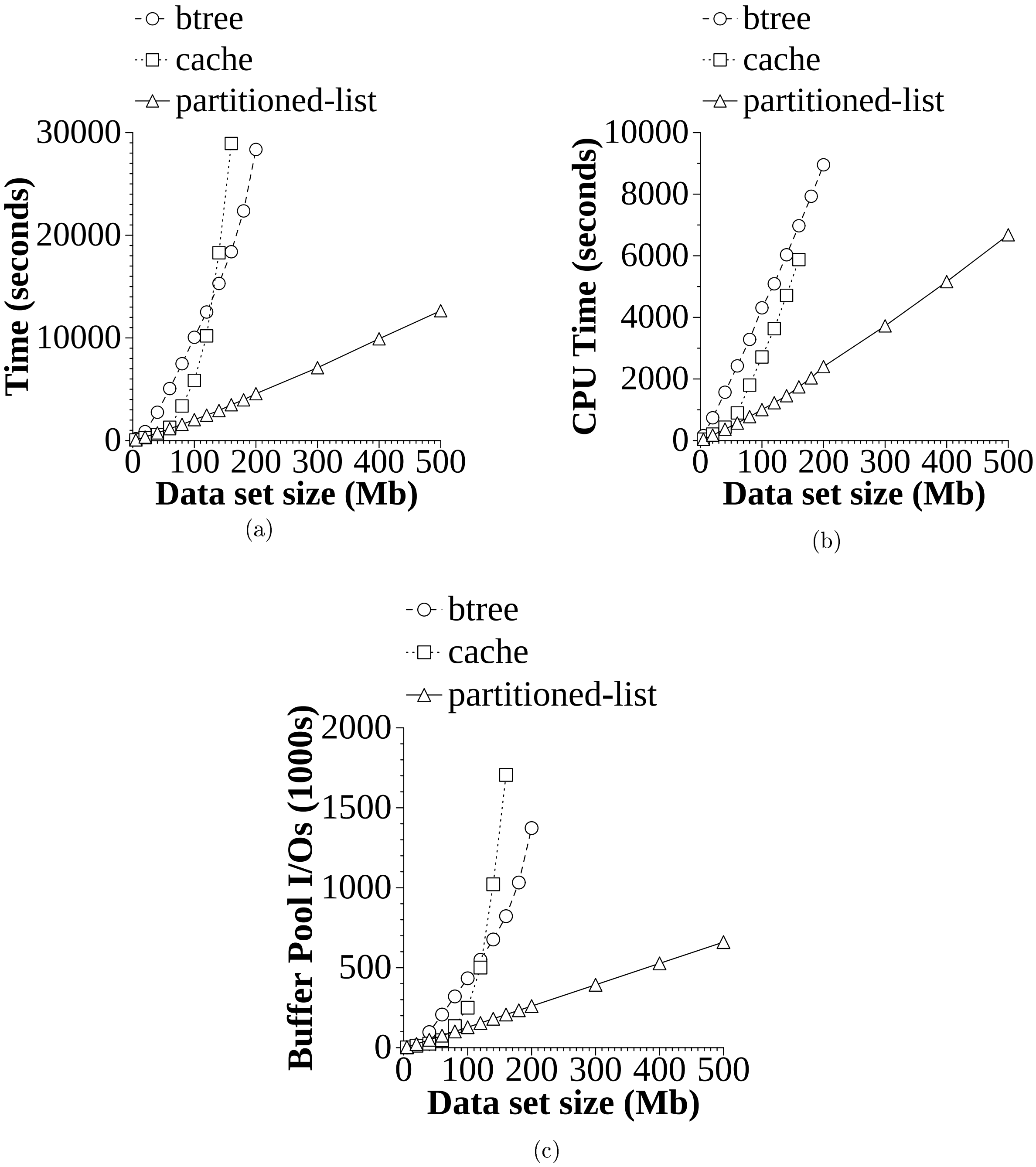


Figure 28: Comparing algorithms for loading data sets with very little memory available.

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 total time and CPU time, with the fewest number of I/Os. Comparing the total time to the CPU time for partitioned-list reveals that approximately 50% of the total 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 total 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 over 8 hours (while partitioned-list completes the load in 1 hour).

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 btree 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) 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

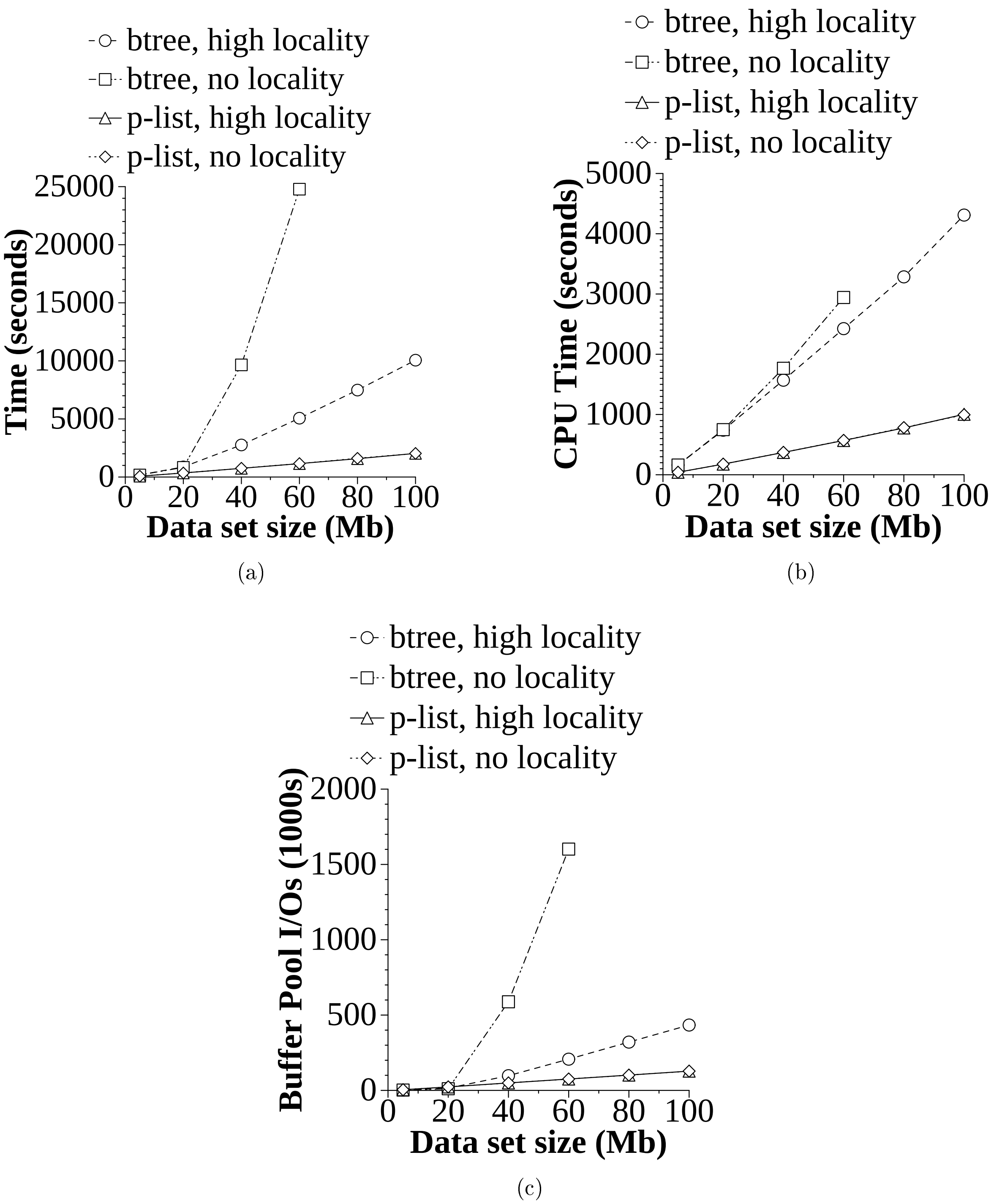


Figure 29: Comparing the sensitivity of the btree and partitioned-list algorithms to the locality of reference in the data set.

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.

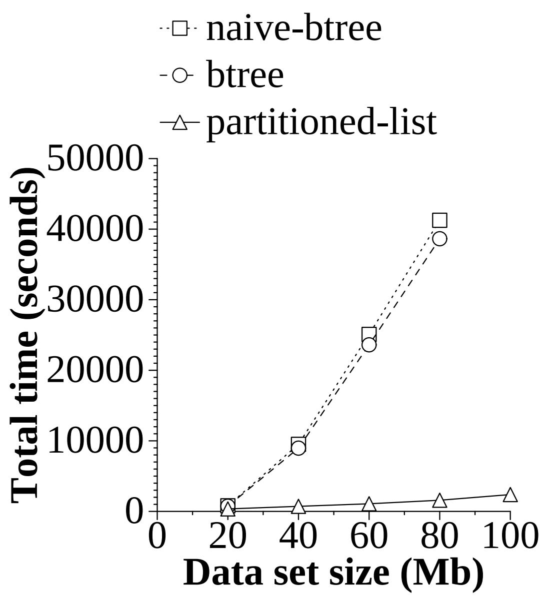
Loading some data sets with no locality of reference validated this theory, and we show the results in Figure 29. There was no difference in the partitioned-list algorithm when loading data sets with and without locality of reference; the times varied so slightly that the lines appear to overlay each 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.

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

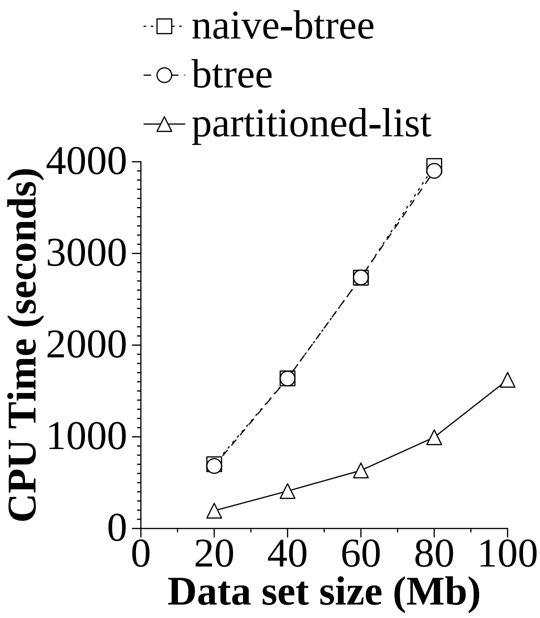
Although all of the major commercial OODB vendors support inverse relationships, many object-relational databases do not (e.g., Illustra [Ube94] and UniSQL [Kim94]) and/or users may choose not to use them. As a final comparison of the algorithms for handling large data sets, we altered the data file 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 30 shows the results. Naive-btree and btree are nearly indistinguishable on the graphs (btree is actually 6-8% 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 1/2 hour while btree takes over 10 hours.

4.3.4 Discussion

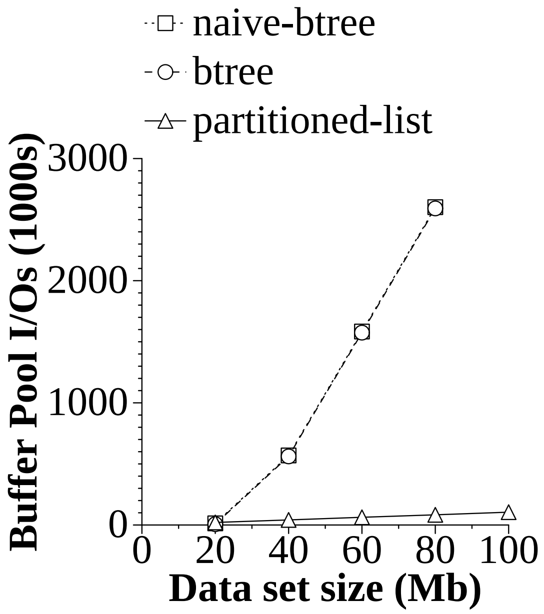
When the id map fits in memory, partitioned-list is less than twice the cost of in-mem, which neither creates a todo list nor writes the id map to disk. Using hybrid hash join instead of Grace hash join to join the id map with the todo list and inverse todo list would eliminate the extra cost of writing the



(a)



(b)



(c)

Figure 30: Comparing algorithms when there are no inverse relationships.

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 21 times faster than btree on a 60 Mb data set! Even when there are no inverse relationships in the schema, 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 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.

4.4 Conclusions

Loading new data, especially large quantities of 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 repetitive accesses to both the database and any other secondary storage data structures.

In this chapter, 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 repetitive 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 linearly 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.

The partitioned-list algorithm incorporates the techniques introduced in Chapter 3 for efficiently handling inverse relationships. However, it 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 conclude that eliminating repetitive I/O is the key to achieving good performance in a load algorithm, as demonstrated by the effectiveness both of sorting the inverse object updates and of joining the id map to the items that needed to be looked up, instead of performing associate lookups. We recommend the partitioned-list algorithm unconditionally for loading sets of new data. In Chapter 5 we begin with the partitioned-list algorithm, and extend it to load new data sets whose objects share relationships with objects in the database.

Chapter 5

Incremental loading

5.1 Introduction

In Chapters 3 and 4, we addressed many of the problems posed by relationships in loading a new database. We introduced efficient techniques for creating relationships among new objects, and for creating the inverses of those relationships. However, we ignored the problem of creating relationships between new objects and objects that already exist in the database.

Yet users need to incrementally load data. Consider the following examples, all of which require that new objects share relationships with existing objects.

- A university database models students, courses, and department information. Each semester, new sections of courses are offered, and students register for those sections. A typical load contains the new sections, each of which has a relationship to its existing course description object, and the new registrations (one per student per course), each of which has a relationship to the new section and to the (existing) student who registered.

- A scientific experiment database captures information about the input and output of various soil science experiments. A complex object connecting the hundreds of input parameters is constructed to describe an experiment. Then the experiment is run, possibly many times, and the results must be loaded into the database and connected to the appropriate experiment object and its inputs. For example, objects describing the growth of each plant must be connected to the objects describing the plants.

Although the load utilities offered by the object-relational systems Illustra [Ill94] and UniSQL [Kim94] solve some of the problems posed by relationships, they are incapable of connecting new objects to existing objects during the load. Instead, those relationships must be created by update statements after the load is complete. Although batch update statements may be used, the statements must identify exactly the new objects to be updated, and to which existing objects each should be connected — which can be quite difficult when there are many such objects. If individual update statements are used, evaluating them is necessarily inefficient, often orders of magnitude worse than if the updates were batched, as we showed in Chapter 3.

In this chapter, we focus on how to identify existing objects in the database and on how and when to create relationships to them, as part of the load. In the data file describing the new objects, each new object description specifies the existing objects to which it should be connected. We propose using queries as the most natural means of identifying existing objects. For example, it is already possible to use queries to connect objects when creating them with individual insert statements in Illustra [Ill94], although not when creating them during a bulk load.

We further suggest using query functions to define similar queries. A query function defines a query where some or all of the query constants are parameters to the function. Query functions are common in relational database systems, such as Sybase [Syb92] and Informix [Inf94]. Using query functions has two advantages. First, query optimization is only necessary per query function, rather than per query. Second, use of the query functions allows easy identification of similar queries. We give examples of query functions in Section 5.3; the research OODB systems Iris [FBC+90] and Postgres [RS87] both support query functions, as does Illustra [Ill94].

The main contribution of this chapter is our observation that similar queries can be evaluated *together* during the load, and that doing so provides huge performance gains. We address both how to evaluate the queries, and how to integrate the query evaluation into a load algorithm. We also discuss when to update the existing objects that share bidirectional relationships with new objects, and hence need to be modified. After describing our new technique, we present a performance study comparing it to simpler techniques.

5.2 Related work

In the context of scientific experiments, Cushing et al. [CMR+94] propose using proxy objects to keep track of an experiment's input objects and to facilitate loading the result objects, including linking them to the input objects. However, the process of identifying the existing input objects is just pushed earlier in time, from when the results are loaded to when the proxy is constructed. (They propose using a graphical tool to identify the existing objects, which will not scale well.) Also, they do not discuss how or when during the load to create the relationships.

Sellis [Sel88] and others [PS88] look at how to optimize and evaluate multiple queries. However, they do not consider queries that have the same form but different constants in their predicates, which is exactly the set of queries we would like to optimize together. Furthermore, much of their work focuses on recognizing related queries. We let the explicit use of query functions solve the recognition problem.

Evaluating multiple instances of query functions is also related to evaluating correlated subqueries. The techniques we apply to evaluate multiple queries together are similar to those for decorrelating subqueries [Kim82, Day87, GW87], and attain similar improvements in performance. However, the work on decorrelation focuses on when to rewrite a single query; our "rewriting" groups together multiple queries that are submitted (as part of the load) at the same time.

5.3 Loading example

We use an example database schema and data file, including queries, to illustrate the query evaluation techniques in our loading algorithms.

5.3.1 Example database schema

The example schema defines the relevant portions of the university database mentioned in Section 5.1. In this schema, each Course has a one-to-many relationship with a Department, and a many-to-one relationship with Sections of the course. Enrollment has a one-to-one relationship with both a Student and a Section, representing the Student's enrollment in that Section. We define the schema in Figure 31 using the Object Definition Language proposed by ODMG [Cat93].

5.3.2 Example query functions

In Figure 32, we define two query functions over the university database. Note that except for giving each query a name, the queries look very much like ordinary SQL queries (with object-oriented extensions).[1] However, we have replaced the constants in the query predicates with "*"s. The "*"s

[1]The range clause should specify a homogeneous collection, but the collection need not be a type extent; another

```
interface Course {
    attribute char name[20];
    attribute int number;
    relationship Ref<Department> dept
        inverse Department::courses;
    relationship Set<Section> sections
        inverse Section::course;
};
interface Section {
    attribute char semester[5];
    attribute int year;
    attribute int section_number;
    relationship Ref<Course> course
        inverse Course::sections;
    relationship Set<Enrollment> enrolled
        inverse Enrollment::section;
};
interface Student {
    attribute char name[40];
    relationship Set<Enrollment> classes
        inverse Enrollment::students;
};
interface Enrollment {
    relationship Ref<Section> section
        inverse Section::enrolled;
    relationship Ref<Student> student
        inverse Student::classes;
    attribute int num_credits;
    attribute char grade[2];
};
```

Figure 31: University database schema definition in ODL.

represent values that will be filled in by an instantiation of the query function. Figure 33 shows an example instantiation, and the full query to which it is equivalent. The query retrieves the course offered by the CS Department whose course number is 101.

Note that each query should select exactly one object. Since a query may select potentially many objects, some method must be chosen to ensure that only object is returned. For our experiments, if more than one object matched the query, we chose one at random. Also, although we only implemented support for queries with one collection in the range clause, allowing multiple collections (i.e., joins) in the range clause is a straightforward extension of the implementation.

top-level collection or a collection inside another object might be used instead.

```
define query function find_student as
select s
from Student s
where s.name = *;

define query function find_course as
select c
from Course c
where c.dept.name = * and c.number = *;
```

Figure 32: Query function definitions.

```
find_course("CS", "101") ⇒ select c
                           from Course c
                           where c.dept.name = "CS"
                           and c.number = "101";
```

Figure 33: Example instantiation of a query function.

5.3.3 Example data file

```
Section(semester, year, section_number, course) {
    15:  "Fall", 1995, 1, find_course("CS", "101");
    16:  "Fall", 1995, 2, find_course("CS", "101");
}
Enrollment(section, student, num_credits) {
    21:  15, find_student("Sally"), 3;
    22:  15, find_student("George"), 3;
    23:  15, find_student("Alice"), 4;
    24:  16, find_student("Manish"), 3;
}
```

Figure 34: Example data file using query functions.

In Figure 34, we show a portion of an example data file which uses the query functions defined in Figure 32. The complete data file also contains the query function definitions in Figure 32. Since the Student and Course objects are already in the database, query function instantiations are used to represent relationships between them and the new Section and Enrollment objects. Relationships between the new objects are represented by the integer surrogates which precede each object description. Both the surrogates and the query instantiations must be resolved to the correct OIDs to store in each new object. In the case of the query instantiations, resolving them to OIDs involves evaluating the queries they represent. In addition, because the relationships are bidirectional, the inverse (existing) objects must also store the OID(s) of the corresponding new object(s). In the next section, we discuss how to evaluate the query instantiations, and when to update the existing inverse objects.

5.4 Query evaluation in the load algorithm

The simplest method of evaluation is to evaluate each instantiation of a query function separately, when it is first encountered. We first present two variants of this strategy, which we call *immediate-evaluation*. One variant is always applicable, while the other requires an appropriate index. This strategy corresponds to a nested-loops join between the query instantiations (as the outer relation) and the collection over which the query ranges (as the inner relation).

Then we present another strategy, *deferred-evaluation*, which defers evaluating any of the query instantiations until they have all been seen, so that they may be evaluated *together*. Therefore, at evaluation time, any join technique may be used to join the query instantiations and the query collection.

In all cases, we defer updates to the existing objects (which are needed when the relationship is bidirectional) so that they can be processed efficiently, without random and repetitive I/O. We showed in Chapters 3 and 4 that batching and sorting updates to objects, instead of applying them as they are discovered, can decrease the total load time by several orders of magnitude.

We integrate each strategy into our previous load algorithm, the *partitioned-list* algorithm, and present the complete resulting algorithm. At the end of this section, we discuss the timing of the steps involved in handling queries in the load, and examine the performance and concurrency trade-offs involved in performing those steps earlier or later in the load algorithm.

5.4.1 Immediate evaluation

In this strategy, whenever a query instantiation appears in the data file, it is immediately evaluated and resolved to the OID of a matching (existing) object. We looked at two common techniques for evaluating single queries with one or more selection predicates over a collection.

Scan the collection The simplest way to evaluate a query which ranges over a collection is to scan the collection, examining each object to see if it matches the query predicates. Once a match is found, the scan halts. The cost of evaluating a single query is proportional to the size of the collection, C. Because each query is evaluated separately, the cost of evaluating N queries is $O(C * N)$.

Use an index to probe the collection If an index is available on one or more of the attributes in the query predicates, then an obvious variation of immediate-evaluation is to use the index to probe the collection, rather than scanning it. If the collection spans more than a few pages, using an index to find potential matches can be much faster than scanning the collection. The cost of evaluating a single query is then the cost of an index lookup, plus the cost of checking the objects retrieved via the index until a match is found. If the height of the index is h, then the cost of using the index to evaluate the query is proportional to $h + 1$ (assuming a single object is retrieved by the index lookup). The cost of evaluating N queries is thus $O(h * N)$. Generally, h is much smaller than C.

We now adapt the partitioned-list algorithm from Chapter 4 to accept queries in the data file and use immediate-evaluation to evaluate them. We add a separate update list to keep track of updates to existing objects caused by new bidirectional relationships they share with new objects, and process it separately in a new step. We also add a new component to step 1. The following changes are made.

1. Insert into step 1.

   - For each relationship described by a query instantiation, instantiate the query function and evaluate it. (For example, suppose that the description of A contains a query identifying existing object B. Evaluate the query to retrieve B's OID.) Create an update entry for A on the update list, containing the OID just pre-assigned to A, the OID that is the query result (B's OID), and some information about where to store the result OID.

     – If the relationship has an inverse, make an entry on a separate update list for existing objects, indicating that the query result object B should be updated to contain the OID of A.

2. Insert a new step directly after step 1.

- Sort the existing object update list so that the updates will be applied to existing objects in physically sequential order,[2] and grouped together by object to update. Update the existing objects.

  After this step, the existing database objects will not be accessed again. Performing the updates this early hopefully catches some of the existing objects still in the buffer pool. However, locks must still be maintained on the objects in case the load transaction must be rolled back.

Note that since the load data structures are written and then read once each, sequentially, the only redundant or random I/O in the algorithm is due to query evaluation. Scanning a collection for each query causes multiple reads of each object in the collection, and using an index causes multiple reads of random index pages, and reads of random pages in the collection. As in our previous chapters, we looked for ways to reduce or eliminate the redundant and random I/O; this led to the deferred-evaluation strategy.

5.4.2 Deferred evaluation

With immediate-evaluation, although each query may be evaluated by the most efficient technique for a single query, it is not clear that using this strategy for each query individually is the best global strategy. For example, the first variant of immediate-evaluation scans the collection once for each query, checking each object to see if it matches a set of predicate constants. However, it cost virtually nothing extra (if the predicates are simple) to check each object against *two* sets of predicate constants. That is, the cost of evaluating two queries together is nearly the same as that of evaluating one query.

This observation that two queries may be evaluated together more cheaply than separately is the intuition behind deferred-evaluation: if we defer evaluating queries, we can evaluate the queries that differ only in their predicate constants (i.e., are different instantiations of the same query function) together. We use an *instantiation table* to keep track of all of the instantiations of a query function; the table contains one entry per instantiation, and there is a separate table for each query function. Another way to view the evaluation of all of the instantiations of a single function is then as a *join*

[2] If the objects have physical OIDs, use the OIDs in the update entries as sort keys. If not, retrieve and store each object's physical location in the update entry at the time the entry is generated, when the object is still pinned by the query evaluation. Use the physical locations as sort keys instead.

between the instantiation table (containing the sets of predicate constants) and the collection over which the query function ranges.

Deferred-evaluation, therefore, defers evaluating any query instantiation until all of the instantiations are known. Then one join is performed per query function. We chose to implement (and describe) hybrid hash join [DKO+84] for the join, but any join technique could be used. The following changes are made to the load algorithm.

1. Insert into step 1.

    - For each query function, allocate an instantiation table.

2. Insert into step 1, instead of the step to do immediate query evaluation.

    - For each relationship described by a query instantiation, make an entry in the appropriate instantiation table. The entry contains a representation of the query instantiation (i.e., the predicate constants for the query), the OID of the object to update (which was just pre-assigned), and information about how to update it and whether the relationship has an inverse.
    - If an instantiation table grows too big to keep in memory, write it to disk. In our implementation, the table is a hash table and we write the entries out in hash partitions in preparation for a hash join. The partitioning function and the join use a simple hash function on the predicate constants.

    Note that duplicate query instantiations are easily detected and are grouped together in the instantiation table. In fact, we save only the object update information for duplicates; we do not save two copies of the instantiation. Then during the join, duplicates are handled together. In direct contrast, both variants of immediate-evaluation will evaluate each duplicate instantiation separately, since each query instantiation is handled separately.

3. Between step 1 and updating the existing objects, insert a new step to join the query instantiation table and the query collections. Since we had to hand-code the join algorithm, we chose to implement a single join algorithm, a hash join with the instantiation table as the build relation, and the collection over which the query ranges as the probe relation.

For each query function:

If the instantiation table is still in memory:

- Scan the collection over which the query ranges. For each object in the collection, probe the table.
  - For each matching query instantiation (including duplicates), make an entry on the (new object) update list with the saved OID of the object to update, the saved information about the relationship, and the OID of the collection object to store in the object.
  - If the relationship has an inverse, make an entry on the update list for existing objects, indicating that the collection object should be updated with the OID of the new object.

If the instantiation table has been written to disk:

- Repartition the instantiation table if necessary.

  If any of the partitions is too large to fit in memory, split all of the partitions by rehashing on the instantiations' predicate constants. Repeat until all partitions can (individually) fit in memory, which is necessary for building the hash table below.
- The instantation table has been written to disk in hash partitions. Allocate an equal number of partitions for the collection. Scan the query collection.
  - For each object, hash on its values for the query predicates and write (a copy of) the object to the appropriate partition. Note that the entire object need not be written; we write only its OID and the values needed to evaluate the query.
- Join each corresponding pair of instantiation table and collection object partitions.
  - Read the instantiation table partition into memory.
  - Scan the collection partition. For each object in the partition, probe the table and handle each match as above.

The complete load algorithm using deferred-evalution is now as follows.

1. Read the data file, and create the id map, todo list, and inverse todo list entries. Allocate and fill in the query instantiation tables.

2. Join each query instantation table with the collection over which the query ranges. Create entries on the existing object update list and the new object update list.

3. Sort the update list for existing objects and update the objects.

4. If necessary, repartition the id map, todo list, and inverse todo list.

5. Join the todo list and inverse todo list with the id map to create the update list for new objects.

6. Sort the update list.

7. Read the data file and sorted update list concurrently. Create the new objects.

From step 3 onwards, the load algorithm is the same for both immediate-evaluation and deferred-evaluation. Step 3 is desirable for updating the existing objects, instead of performing the updates as they are discovered, for two reasons. First, if the collection is partitioned, then the actual object may not be in memory when the join is performed, and physically consecutive objects will probably not be in the same partition. Second, more than one query function may range over the same collection, and contribute updates to the same objects. Therefore, it is better to defer the updates until they are all known, and sort them into physically sequential order. A single sequential pass over the portion of the database containing the query collections then suffices to update all of the existing objects.

We note that deferred-evaluation could use any join technique. Since Shore does not currently have query processing capabilities, we had to write the join code and we chose to implement only hash join. However, if the instantiation table were stored as a generic table (i.e., as a relation or a set of objects), then it would be feasible to use any join operator to perform the join. If appropriate statistics on the instantiation table were gathered as the instantiations were added, it might also be possible to let a query optimizer pick the appropriate join strategy. (There are cases where the roles of build and probe relation should be reversed, for example.) However, storing the instantiation table generically does carry some liabilities: for example, each instantiation would necessarily carry the overhead of an object; we avoided such overhead in our hard-coded approach. Also, the join operator would need to feed the join results into the update lists — which is easy if and only if the update lists are also stored internally as tables, and not as (potentially more efficient) load-specific data structures.

The cost of evaluating all of the instantiations of a query function using deferred-evaluation is only $O(C)$, the cost of scanning the collection once, if the instantiation table fits in memory. If the instantiation table gets written to disk, then the cost is $O((2 * P + 1) * C + 2 * (R + 1) * N)$, where R is the number of times the instantiation table is repartitioned and P is the number of passes needed to partition the collection. We expect R to be 0 or 1 and P to be 1 in most cases, so that this cost formula simplifies to $O(3 * C + 2 * N)$. We therefore predicted that deferred-evaluation would perform very well whenever there were more than a trivial number of query instantiations, and would always perform better than the scanning variant of immediate-evaluation.

5.4.3 Timing of query evaluation

We now examine the concurrency and performance trade-offs involved in processing queries during the load. Clearly, none of the newly loaded objects should be visible until the load is complete; otherwise, the user will see relationships to objects that have not yet been created and it will not be possible to abort and rollback the load.

Similarly, once the existing objects have been updated with relationships to new objects, the existing objects must remain exclusively locked for the remainder of the load. However, before the existing objects are updated, other transactions may read them, and before the existing objects are retrieved, i.e., before the queries are evaluated, the existing objects may be updated by other transactions. Therefore, to get the best concurrency, it is desirable to defer evaluating the queries and updating the existing objects as long as possible.

For the best performance, on the other hand, it is desirable to evaluate the queries and update the objects as early as possible. By evaluating the queries early, the "updates" to the new objects can be sorted together with the "updates" that come from the todo and inverse todo lists. Therefore, these updates can be applied as the objects are created, and do not involve updating the object at all. If the queries are evaluated after the objects are created, then another pass over the new database is necessary to perform the updates.

Suppose the queries are evaluated early, as proposed above, but the updates to the existing objects are delayed until after the new objects have been created. This may yield good performance and definitely affords better concurrency. However, there may be a performance advantage in updating the objects immediately after retrieving them (they may still be in the buffer pool), or in sorting the update list immediately after creating it (sorted runs of updates may still be in the buffer pool).

Since concurrency is not something we can measure with single-user tests on a single machine, and the algorithms described in detail above will definitely yield the best performance, these are the algorithms we implemented. However, if concurrency is a critical objective, this decision should be revisited.

5.5 Performance results

We ran a series of experiments to show how the query evaluation algorithms performed with different existing database and load data set configurations. We varied the size of the existing object collection over which the queries ranged, the size of the data set being loaded, the percentage of object descriptions that contained queries, and the number of distinct queries (i.e., the number of existing objects that were the result of a query). In each performance test, we held three of the above factors constant, and varied only one of them. The buffer pool for each experiment was 4 Mb.

Abbreviation	Query evaluation strategy	Variant
immed-eval	Immediate-evaluation	Scan collection
immed-eval-index	Immediate-evaluation	Use index over collection
deferred-eval	Deferred-evaluation	

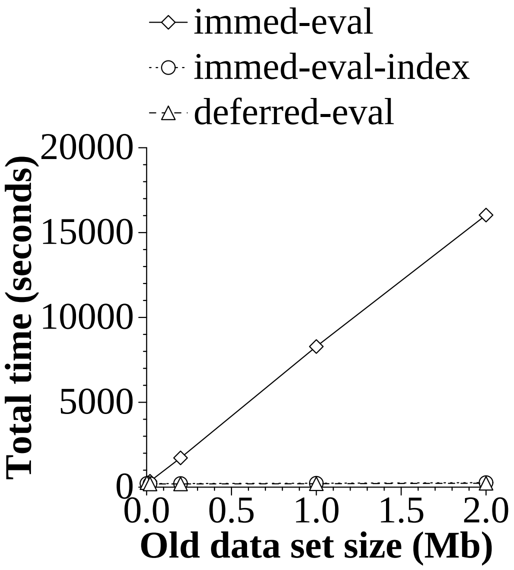
Table 4: Abbreviations for query evaluation strategies.

We use the abbreviations in Table 4 to refer to the load algorithms containing each of the query evaluation strategies.

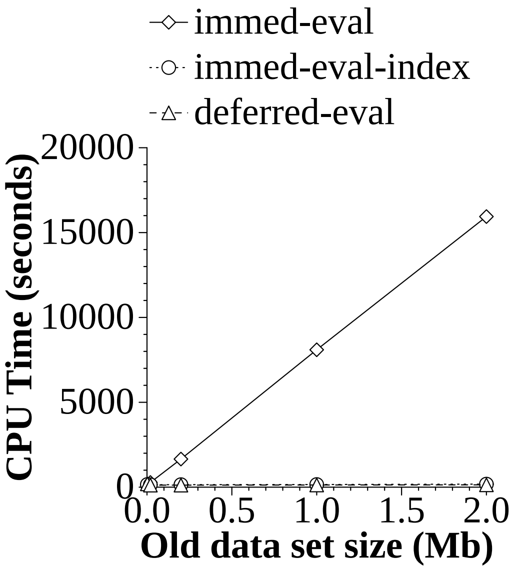
5.5.1 Varying the size of the existing object collection

For the first set of experiments, we varied the size of the existing object collection from 100 objects (0.02 Mb) to 10 Mb. We held the size of the data set being loaded constant at 10 Mb. Each new object description contained exactly one query, which targeted an existing object chosen at random from the collection.[3] Figure 35 shows the results for all three query evaluation strategies when the existing object collection fits in the buffer pool, and Figure 36 shows the results for immed-eval-index and deferred-eval when it does not.

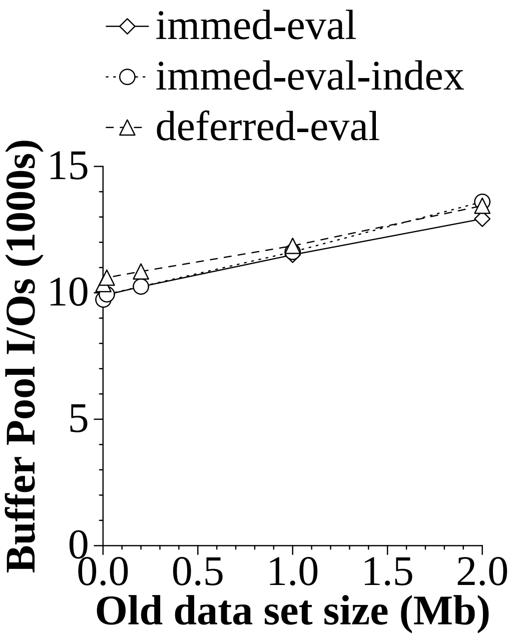
[3] By chosen at random, we mean that when the data file was generated, the constants in each query instantiation were chosen at random from the range of values held by existing objects. Each query in the data file targeted a specific object in the existing object collection.



(a)



(b)



(c)

Figure 35: Comparing all of the query evaluation strategies for different existing database sizes.

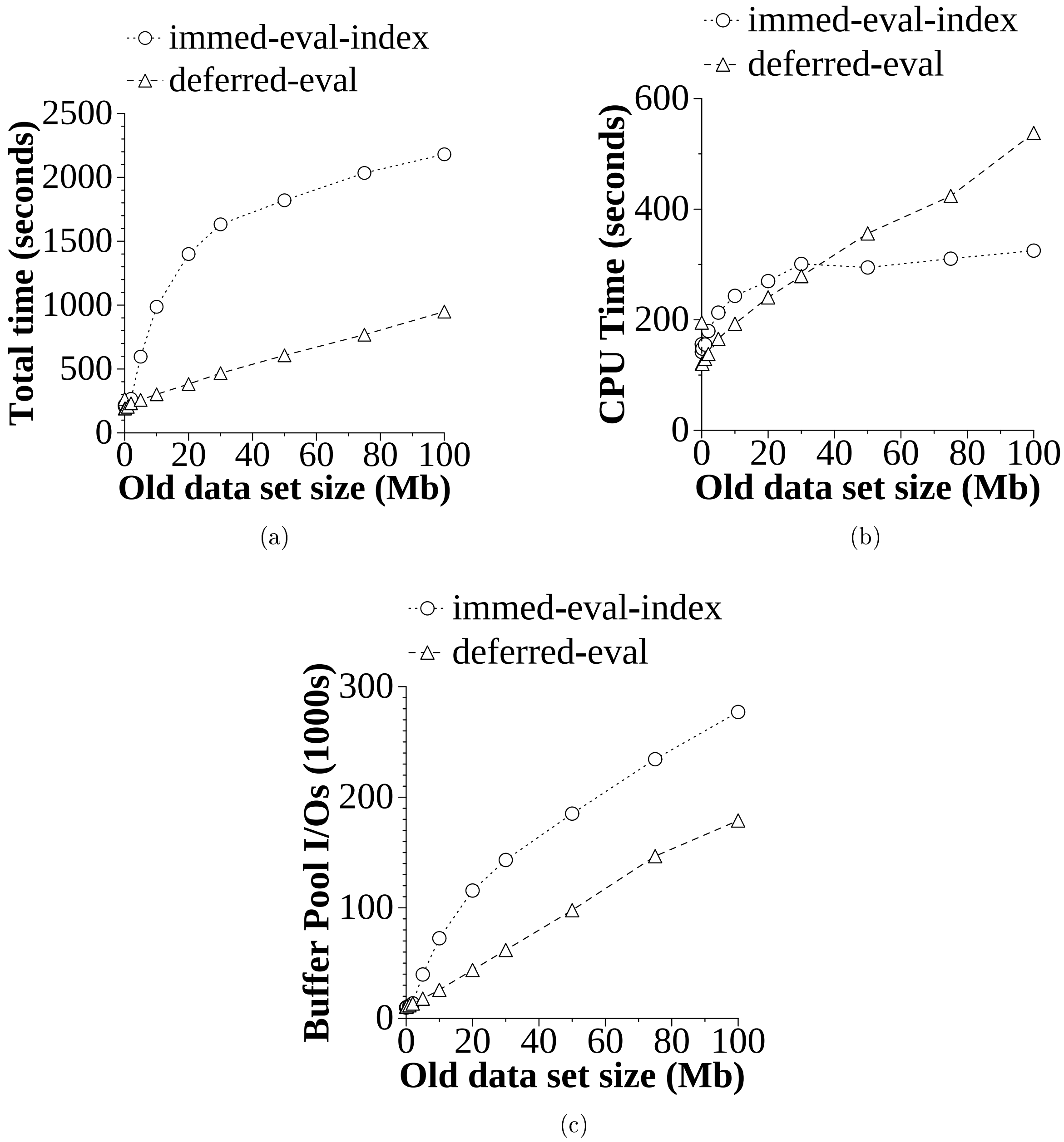


Figure 36: Comparing the better query evaluation strategies for different existing database sizes.

It is clear that immed-eval does not scale to handle queries over large collections, regardless of whether they fit in the buffer pool. Even when the existing collection only contains 100 objects on 3 pages, immed-eval is 50% slower than immed-eval-index and deferred-eval. As the size of the existing object collection grows to 1 Mb, the amount of CPU time incurred by pinning and examining each object in the collection once per query causes immed-eval to take an order of magnitude longer than either immed-eval-index or deferred-eval, and at 2 Mb, immed-eval takes nearly 2 orders of magitude longer (16,032 vs. 231 seconds) to finish the load. When the extra I/O involved to scan a collection larger than the buffer pool is coupled with the high CPU costs, immed-eval becomes completely impractical to run.

Immed-eval-index is much less sensitive to the size of the existing object collection than immed-eval. However, it is only as good as deferred-eval when the entire existing collection fits in the buffer pool. The total time needed by immed-eval-index to load is directly correlated with the number of I/Os performed. The number of I/Os needed to locate each existing object rises sharply with the 5 Mb collection, which does not fit in the 4 Mb buffer pool. The number of I/Os continues to rise steeply as less and less of the index fits in the buffer pool, and fewer index and collection pages are still in the buffer pool from a previous query when they are needed again. After 30 Mb, the CPU time for immed-eval-index levels off, since a constant number of pages are pinned to answer each query. However, immed-eval-index still takes more than twice as long to load as deferred-eval once the index doesn't fit im memory, e.g., taking 1820 vs. 606 seconds to load when the object collection is 50 Mb.

In direct contrast to immed-eval and immed-eval-index, deferred-eval shows a total load time that grows slowly and linearly with the size of the existing object collection. Since deferred-eval scans the entire collection, as the size of the collection grows, scanning it begins to dominate the number of I/Os performed. However, the total time is only moderately affected, since prefetching helps lower the cost of the I/Os, the collection is only scanned once, and the query evaluation is only one part of the load.

Note that the CPU time for deferred-eval grows linearly with the size of existing object collection, because a constant amount of work is done for each object in the collection. A hash key for the object is built and used to probe the instantiation table. The CPU time for immed-eval-index, on the other hand, is proportional to the number of queries, and irrespective of the size of the existing object collection (once the index no longer fits in the buffer pool — not having to pin index pages does save CPU time).

However, the I/O costs of the algorithms dominate the total time, and therefore, deferred-eval continues to be the faster algorithm even when it requires more CPU time.

5.5.2 Varying the number of new objects loaded

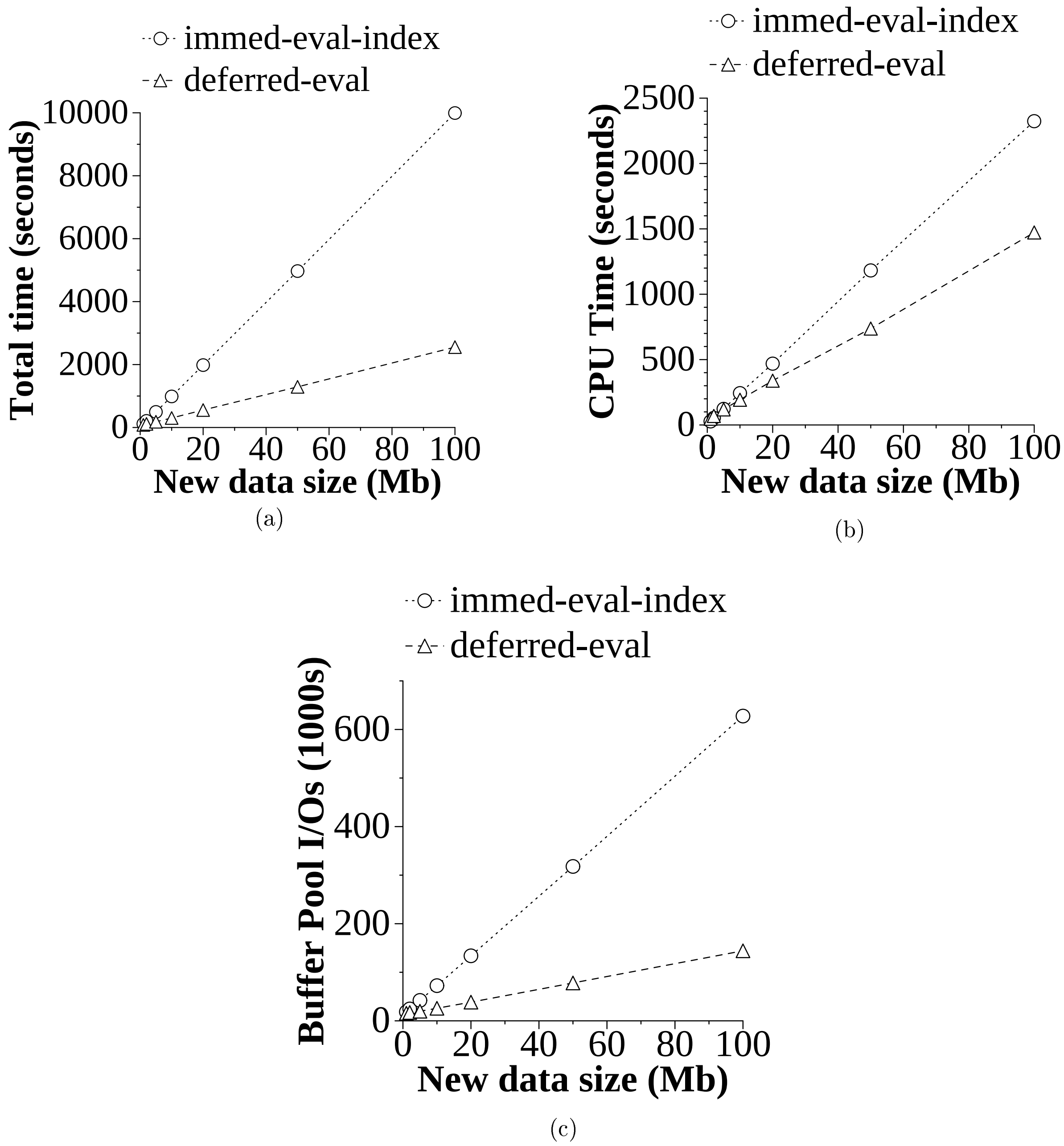


Figure 37: Comparing query evaluation strategies for different new database sizes.

In the next experiment, we held the size of the existing object collection constant at 10 Mb and varied the size of the new object data set from 100 objects (0.2 Mb) to 100 Mb. Each new object again contained one relationship to an existing object, targeted at random from the existing object collection. Figure 37 shows the total time and number of I/Os incurred by immed-eval-index and deferred-eval. We did not run immed-eval, since it was apparent from the first set of experiments that immed-eval could not scan a collection of 10 Mb in a competitive length of time.

The cost of immed-eval-index is a constant multiple of the number of queries to evaluate, since each query is evaluated separately and has a constant cost. With a 10 Mb existing object collection, the height of the B^+-tree index is two. The root page stays in the buffer pool, and the cost of evaluating each query is two I/Os: one index page and one object page. Since each new object description contains one query, the cost of immed-eval-index grows linearly with the number of new objects.

The cost of deferred-eval also grows linearly with the number of new objects. However, the cost of scanning the existing object collection stays constant. It is the cost of saving the query instantiations in the instantiation table, and then building a hash table on them, which grows with the number of new objects. Because many (approximately 140) query instantiations can fit on one page of the instantiation table, and each page is written and then read just once, the number of I/Os required to handle an increasing number of queries grows much more slowly than the number of queries. Figure 37(b) illustrates the different rates of I/O growth exhibited by immed-eval-index and deferred-eval. The different I/O growth rates underly the total times displayed by each algorithm; both the number of I/Os and the total time incurred by immed-eval-index are roughly four to five times those incurred by deferred-eval.

5.5.3 Varying the number of queries

For the third experiment, we varied the total number of relationships to existing objects. We held the size of the existing object collection constant at 10 Mb, and also held the size of the new data set constant at 10 Mb. The number of queries ranged from 10 to 50,000, which is 1 per new object created. The total time used by each algorithm is shown in Figure 38: in Figure 38(a) we show the times for all the strategies for 10 to 500 queries, in Figure 38(b) we show the times for immed-eval-index and deferred-eval for 10 to 5000 queries, and in Figure 38(c) we show the times for immed-eval-index and deferred-eval for 100 to 50,000 queries, which is one query per new object.

Our first observation is that when there are only 10 queries, the cost of scanning the entire existing

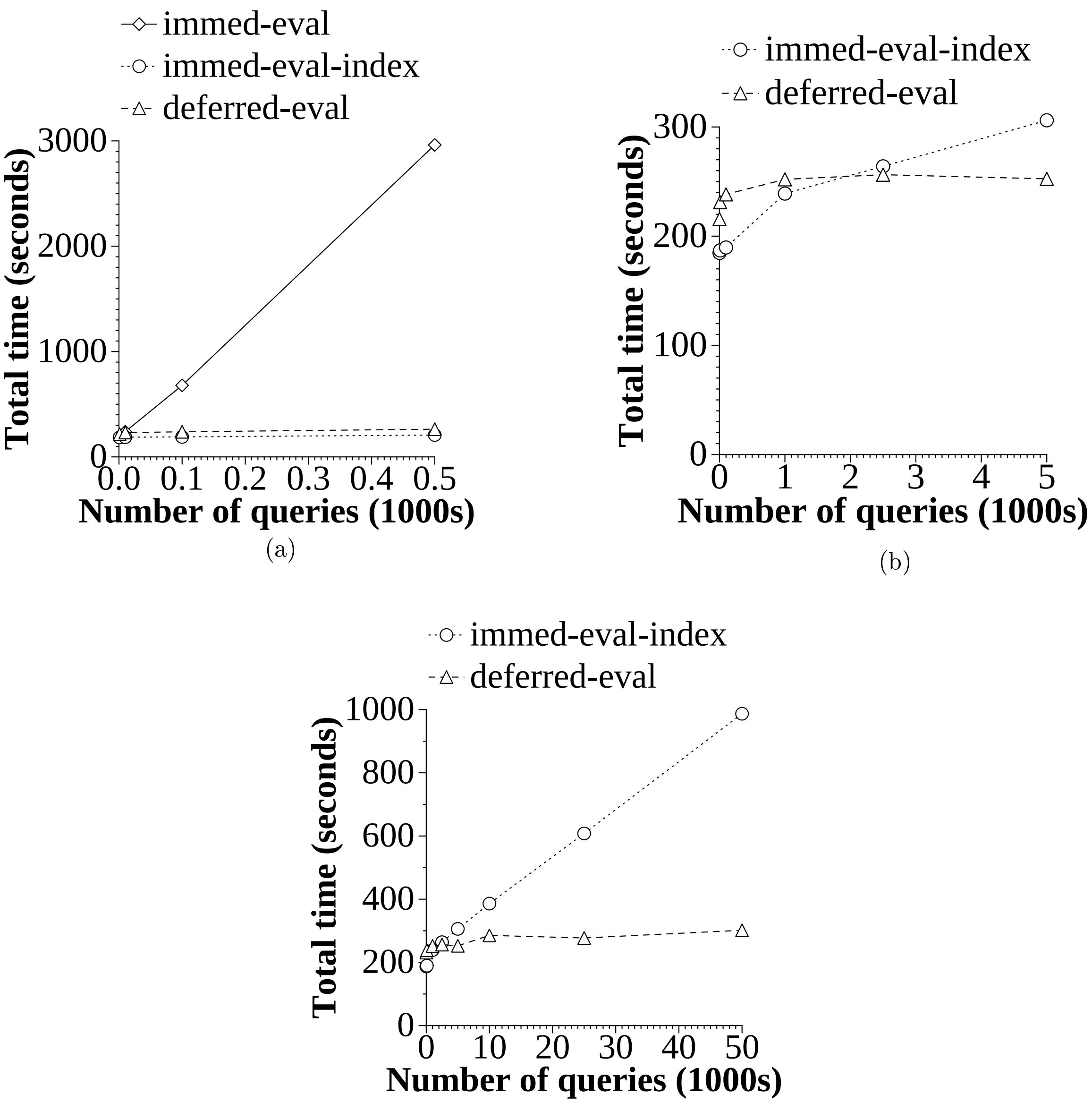


Figure 38: Comparing query evaluation strategies when varying the number of queries.

object collection for each query is not that high relative to the cost of writing the query instantiations to disk and reading them back again. In fact, since the collection scan in immed-eval ceases once a match is found, on average only half of the collection is scanned. Therefore, immed-eval is a reasonable algorithm for evaluating 10 queries (although immed-eval-index is better). However, for 100 queries, the CPU cost of scanning the collection begins to dominate immed-eval's performance, and with 500 queries (one query per hundred new objects) immed-eval takes more than an order of magnitude longer (2961 vs. 207 seconds) to complete the load than either immed-eval-index or deferred-eval.

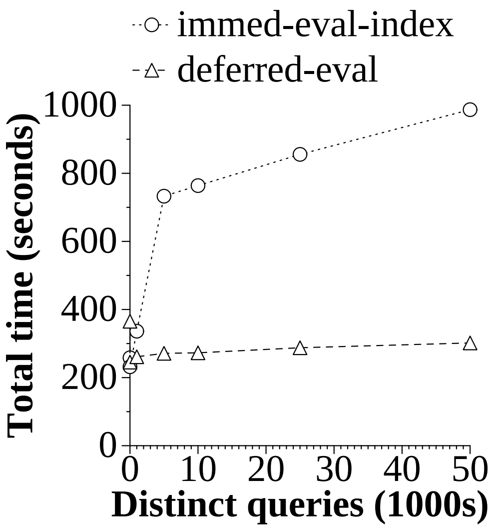
For small numbers of queries (10 to 1000), immed-eval-index is as good as or slightly better than deferred-eval. This is because the cost of 1 or 2 I/Os per query for immed-eval-index, on the one hand, is balanced by the cost of scanning all 1250 pages of the existing object collection for deferred-eval, on the other hand. (Recall that most of the index for the 10 Mb existing object collection stays in the buffer pool, so the index lookups are fairly cheap. If the collection, and hence the index, were larger, deferred-eval would become the faster algorithm sooner, with fewer queries.) Once the number of queries exceeds the number of pages in the existing object collection, deferred-eval is always the best algorithm.

5.5.4 Varying the number of distinct queries

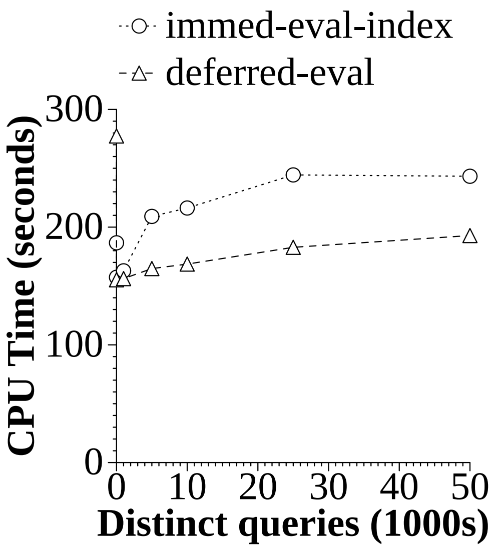
For the final experiment, we held the number of queries constant, but varied the number of distinct queries from 10 to 50,000 (the number of distinct objects). That is, when there were few distinct queries, many of the queries were duplicates. The number of objects in the existing collection and the data set remained constant, both at 50,000 or 10 Mb. We present the results in Figure 39.

Since immed-eval is insensitive to the number of distinct queries (it scans the entire collection for each query), and it takes several orders of magnitude longer than the other algorithms when there are 50,000 existing objects and 50,000 queries, we do not present it.

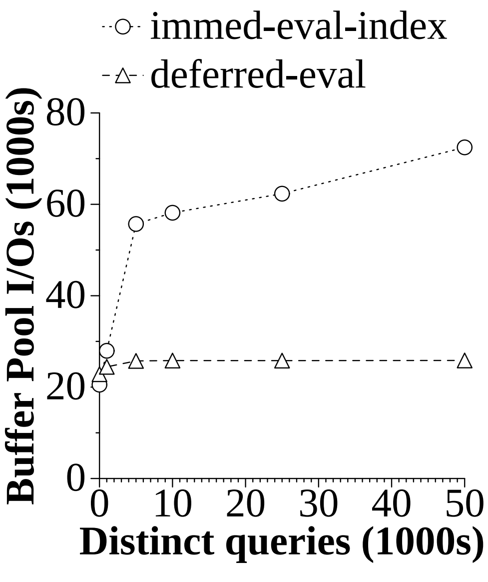
Deferred-eval is also relatively insensitive to the number of distinct queries. The instantiation table contains the same number of entries in all cases, and the existing object collection is scanned just once. Immed-eval-index, on the other hand, is very sensitive to the number of distinct queries. When there are very few, then the relevant index and object pages remain in the buffer pool, and the index lookups are very cheap. However, once there are more distinct query targets than fit in the buffer pool,



(a)



(b)



(c)

Figure 39: Comparing query evaluation strategies when varying the number of distinct queries.

immed-eval-index again becomes roughly four times as expensive as deferred-eval.

5.5.5 Discussion

Immed-eval provides acceptable query evaluation performance only for very small collections of existing objects (1-5 pages in size) or very few queries (tens of queries, regardless of the number of new objects). Immed-eval-index is a much faster algorithm than immed-eval in all cases, and also quite simple to implement. However, immed-eval-index relies on the existence of an appropriate index, and is still quite expensive for large numbers of queries. Nonetheless, if the only algorithms available are immed-eval and immed-eval-index, and there are a non-trivial number of queries, it is probably worth the cost of building an index to be able to run immed-eval-index.

Although deferred-eval is significantly more complicated to implement, if many queries are expected, or the size of the existing object collection(s) is large, then it is worth the extra effort. Once deferred-eval has been implemented, it is not necessary to also consider immed-eval-index: even when immed-eval-index is slightly faster, deferred-eval's performance is within 20% of it.

We also note that if the instantiation table is implemented as a set of tuples, and joins between sets of tuples and other object collections are supported, then implementing deferred-eval is greatly simplified. Each query instantiation is converted to a tuple as it is read in the data file, and a join operator is invoked to perform the join. In fact, a query optimizer may be invoked to choose the best join algorithm. If there are multiple joins involved in the query (i.e., other joins are explicitly specified by the query), then the optimizer can also choose the optimal join order. The current state of Shore has neither a query optimizer nor query operators, so we had to hand-code the join as part of the load implementation.

5.6 Conclusions

Loading data incrementally is important to many users, whose data is generated incrementally. However, the newly loaded data must be able to share relationships with previously loaded data. In this chapter, we addressed the problem of creating relationships between new objects and objects that already exist in the database. We proposed using queries in the data file to identify the existing objects and showed that although evaluating each query separately provides a functional solution, it may not provide adequate performance for more than a small number of queries over a small portion of the existing database. We proposed evaluating all of the queries together instead, using the deferred-evaluation strategy, and demonstrated that this approach scales with the number of queries and size of the existing database to provide much better performance.

We explained the deferred-evaluation strategy as a join between the query function instantiations and the collection over which the query ranges. Immediate-evaluation using an index corresponds to a nested-loops-with-index join. This observation helps explain the performance results that find that immediate-evaluation using an index is good for small numbers of queries, or small collection sizes. It is known that nested-loops-with-index is the algorithm of choice when only a small percentage of the inner relation (the existing object collection) participates in the join [DNB93]. This is the case both when there are few tuples in the outer relation (few queries) and when there are few distinct tuples in the outer relation (few distinct queries).

Since all strategies handle updates to existing objects the same way, the difference in their relative performance would not be affected if the relationships were unidirectional and did not require updates to the existing objects. We recommend implementing the deferred-evaluation strategy as the strategy of choice for evaluating all queries in the data file. Even when it was slower in the performance tests, it was only 20% slower, and this was due to the particular join algorithm we used. In nearly all cases it was faster, and in general, deferred-evaluation may employ any join technique, not just hash join. Optimally, deferred-evaluation would be able to invoke a query optimizer to choose the best join technique, and then invoke the join operator of choice; we believe that in this case, deferred-evaluation would always be the fastest evaluation strategy.

Although in this study we focused on creating relationships that are described with the new objects, simple extensions to the data file syntax would also allow the creation of (unidirectional) relationships from existing objects to new objects. Similar syntax extensions would further permit the creation of new relationships among existing objects. (The object to be updated would be described by a query rather than by a surrogate.) The same strategies we presented would apply to evaluating the queries and to updating the existing objects.

We recommend the partitioned-list algorithm augmented with deferred-evaluation as the algorithm of choice for an OODB load utility. In the next chapter, we consider how to make the partitioned-list algorithm resumable, so that a long running load transaction may be paused and restarted with a minimal loss of work.

Chapter 6

Resumable load

6.1 Introduction

In the previous chapters, we proposed techniques to load objects and incorporated them into an efficient load algorithm. However, even the fastest load algorithm may still run for hours when loading large data sets — gigabytes or terabytes of data. It is highly desirable not to lose all of the effort already spent in loading a database when the load is interrupted. Interruptions always occur when the system crashes; a load may also be interrupted by an urgent query needing the CPU power or buffer pool space, or because disk space was temporarily unavailable.

In this chapter, we show how to make the partitioned-list algorithm we presented in Chapter 4 resumable. First, we identify the specific information to save in a restart checkpoint of the algorithm and discuss how to resume from a given checkpoint. We show that only minimal information needs to be saved in the restart checkpoints, the checkpoints can be taken relatively frequently, and only the work done after the last restart checkpoint is lost if the system crashes. Second, we survey the features that a system like Shore must provide to support a resumable load. The system features are actually quite general, and can be used to suport other resumable transactions.

6.2 Related work

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. Gray and Reuter give a useful overview of ways to reestablish the context of a long-running transaction, such as a load, in order to resume it [GR93].

6.3 Restart checkpoints

In the worst case, a load is interrupted by a system crash, and the load algorithm does not receive a warning of the impending crash that would allow it to halt in a coherent state. 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.

Normal logging is not the solution to making the load resumable, however, for the following reasons: First, when loading new data, the log would need to contain at least as much information about the new data as the database itself, and probably more. (The log must include not only an image of the data created but also metadata about where the data is stored, what transaction created it, the previous action of the transaction, etc.) Furthermore, the log would need to be forced out to disk constantly. Therefore, many commercial relational database systems allow logging to be turned off for loading, e.g., Teradata's NCR 3700 [WCK93], Tandom's NonStop SQL [MHL+92], Sybase's SQL Server [Syb92], IBM's DB2 [Moh93a]. We similarly would like to avoid logging during load.

Second, normally, database systems *undo* the effects of an uncommitted transaction during restart recovery [GR93]. We want to resume an uncommitted load. Third, even redoing the load from the log would be unsatisfactory — it would take at least as long, if not longer, than restarting the load from the beginning! In addition, we would still need a way to resume the load from where it left off.

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.

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 the partitioned-list load algorithm present in Chapter 4, 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 overwriting the previous contents 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, overwrite its contents 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.

6.4 System support needed for a resumable load

Although we discuss the following system features in the context of a resumable load, we emphasize that they are needed to make any long-running transaction be resumable. While the features are not specific to Shore, they do assume a system architecture similar to that of Shore.

- Although we do not want to log all of the data created in the recovery log, it is necessary to log page allocation and updates to existing objects and indices.[1] It is also necessary to log the "structure-keeping" pages of large objects, i.e., the pages that potentially point to other pages of the large object. Otherwise, a structure-keeping page might reference another page that was never written to disk before a system crash. After the crash, traversing that reference could cause the storage manager itself to fail. The system must provide a logging mode that provides this level of logging, rather than simply turning all logging off. The same granularity of logging is needed to add data to existing files without logging all of the data, since it must be possible to identify and remove the new data after a crash without compromising the previously existing data. Sybase,

[1] An index built as part of the load should be created in bulk, from the bottom up [MN92], and would be checkpointed in a manner similar to the load objects.

for example, can load relational data using this granularity of logging *only* if there are no indices [Syb92].

- After committing the current state of the load transaction and taking a restart checkpoint, the load needs to continue from where it was prior to the checkpoint. Therefore, the load must continue to hold its resources, e.g., locks, after the commit. Either chained transactions or persistent checkpoints (also called Phoenix transactions) allow a transaction to keep its resources after committing [GR93]. Teradata, for example, provides this capability to its parallel relational resumable load [Wit94].

- The load must be resumed. The checkpoint record's existence must be recognized, the resources, e.g., locks, of the load must be regained, and the load process restarted. If the database server is responsible for resuming the load, then the restart checkpoint record can be placed in the recovery log, where it will be found during restart recovery. The server can then regain resources on behalf of the load and restart it. The Aries recovery algorithm [MHL+92], for example, contains features that allow the server to perform all three of the above requirements on behalf of a resumable transaction.

  If, on the other hand, the user is responsible for resuming the load, the system must provide a stable state change for the files being loaded and/or used by the load, so that no other user can inadvertently read from those files before the load is resumed and the files again locked. Also, the user or the load program must "remember" where the restart checkpoint records are saved. The records can either have a user-specified tag, be related to the name of the data file, or hard-coded into the load program. Teradata's users are responsible for resuming the load, but the restart checkpoints are remembered by the system [Ter91]. Both Teradata and DB2 provide stable state changes [Wit94, RZ89] to files, which protect them from being read.

Some of the above system features are only necessary to protect the user from seeing incorrect or incomplete data. For example, the first created object will contain the OIDs of not yet created objects. If locks are not continuously held on all of the objects, then the user may see these OIDs and try to locate the objects. However, if the user is willing to assume responsibility for trying to access data in the process of being loaded, and forfit the ability to rollback the load transaction, then neither chained transactions nor stable state changes are essential. Correct logging and a well-known location for the checkpoint records are needed in all cases.

6.5 Conclusions

Only a small amount of extra effort is needed to implement a resumable load algorithm. In this chapter, we presented the design of such an algorithm, and showed that checkpoints can be taken frequently with very little extra overhead. It requires very little extra code. The key requirements for checkpointing a load are the ability to flush pages of the buffer pool to disk and the availability of the system features we identified as necessary for a resumable transaction: logging of bookkeeping and integrity-preserving data (only), chained or Phoenix transactions, and either checkpoint records in the log or a memorable location for the checkpoint record and stable state changes for the files affected by the load. These features, together with the algorithm for taking restart checkpoints, allow a single load transaction to be paused and resumed many times, for any reason, with a minimal loss of work.

Chapter 7

Conclusions

7.1 Summary of thesis results

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 OODB data is significantly more complicated than loading relational data due to the presence of relationships, or references, in the data. In this thesis, we proposed efficient techniques to overcome the obstacles posed by relationships, and presented an integrated algorithm that combines the best of these techniques. For each obstacle, we devised alternative techniques and compared them in a performance study. We note that the key to good performance in each case involved eliminating repetitive and random I/O patterns, by redesigning the algorithms both to use fewer actions that cause I/O, and to use sequential I/O patterns instead. The next four subsections each recap one of the obstacles and the proposed solution. We recommend that a load utility incorporate all of the solutions; the complete load algorithm is described in detail in Chapter 5.

7.1.1 How to refer to other new objects

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, OIDs for new objects are not available at all when the load file is written, because the corresponding objects have not yet been created.

We proposed using surrogates to identify new objects in the data file. 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 of the surrogates into the corresponding OIDs. We used a data structure called an *id map* to map each surrogate to its corresponding OID as the objects are loaded.

In Chapter 4, we studied how to represent the id map, and how to perform the lookups on the id map required to translate surrogates for relationships into OIDs. The simplest implementation of the id map, a random access data structure that allows associative lookups at any time (a hash table or B^+-tree), exhibits very poor performance when the id map does not fit in memory or in the buffer pool. As an alternate solution, we implemented the id map as a sequential list. Instead of allowing associate lookups to the id map, each lookup was written on a "to do" list, including what to do with the result of the lookup. We were then able to join the lookups with the id map, using a standard join algorithm. This approach, the partitioned-list approach, yielded orders of magnitude better performance for id maps that exceeded the size of memory, and, unlike the more naive approach, scaled linearly with the number of objects loaded.

7.1.2 How to refer to existing objects

Previously loaded objects already have OIDs which should be used in relationships to them, but the objects must somehow be identified in the data file. In Chapter 5, we proposed using queries in the data file to identify the existing objects and showed that although evaluating each query separately as it is encountered provides a functional solution, it does not scale well with the number of queries or with the size of the database that must be searched for each query. Instead, we deferred evaluating queries so that similar queries could be evaluated together. We let query functions designate similar queries, and built tables of the instantiations of each query function. Evaluating the queries is then equivalent to performing a join between each query instantiation table and the collection of objects over which the corresponding query function ranges. Standard join algorithms may be used for the join, and an optimizer can choose the best join algorithm for the circumstances.

7.1.3 How to handle forward references in the data file

When surrogates are used to represent relationships in the data file, the surrogate seen in any given object description may be a forward reference to an object described later in the data file. In Chapter 3, we found that the best technique for resolving such surrogates involved reading the data file twice. On the first pass over the data file, OIDs are allocated to each object so that, on the second pass over the data file, it is possible to resolve all surrogates to OIDs. If the OIDs can be pre-assigned to objects during the first pass, then allocating OIDs does not involve any I/O and after the objects are created on the second pass, no updates are necessary. This has the important advantage of not causing any object to change its size, which can impact the clustering of objects. If pre-assignment of OIDs is not possible, the objects are created during the first pass and updated during the second.

One consequence of this technique is that it is easy to integrate with the lookup mechanism for the id map that we proposed in Chapter 4: on the first pass, a list of surrogates to look up in the id map is built, and between the first and second passes, the list is joined with the id map. The list is subsequently sorted to regain its original order.

7.1.4 When to create inverse relationships

Inverse relationships cause not just one object to be updated, but two. Whenever the description of an object in the data file contains a relationship to another object, if the relationship has an inverse, then that other object (the inverse object) must also store the relationship. The simplest method of maintaining inverse relationships is to update both objects at the same time. However, this leads to randomly ordered updates to the inverse objects, and the subsequent repetitive and random I/O causes extremely poor performance if the entire set of objects does not fit in the buffer pool.

We explored an alternative approach in Chapter 3, where information about each required update is written to a "to do" list when the update is discovered. The list is then sorted by the object to update, which groups together updates to the same object. Additionally, if possible, sorting also groups the updates so that the objects are updated in physically sequential order — updates to objects on the same page are grouped together, and the pages are updated in physically sequential order. Obtaining a sort key that yields physically sequential order is not hard.

For new objects, the sequence order of the objects in the data file is the sort key. Inverse updates to the new objects should be identified during the first pass over the data file, and sorted before the second pass. Then reading the "updates" may be merged with reading the data file during the second pass, and the updates applied before the objects are actually created. For existing objects in the database, the physical location of the objects should be retrieved when the objects are identified and still pinned in the buffer pool, and used as the sort key.

For even modest numbers of inverse relationships, sorting the updates resulted in two orders of magnitude improvement in performance, and continued to grow with the size of the database and number of updates.

7.1.5 How to resume a long-running load

In Chapter 6, we presented a resumable load algorithm, so that a load interrupted by a system crash or urgent query may continue rather than starting over. First, 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. A checkpoint may be taken at virtually any time during the load. Second, we specified the combinations of features that a system like Shore must provide to support a resumable load. Not all of the features are mandatory; each provides some protection to the user: from corrupting the database, from seeing incompletely loaded data, or from forgetting to resume the load. Together, the features allow a single load transaction to be paused and resumed many times, for any reason, with a minimal loss of work.

7.2 Recommendations

We recommend that a load algorithm incorporate all of the techniques we list above, and the complete algorithm in described in detail in Chapter 5. In addition, there are several other requirements the load utility must meet in order to achieve our performance results. First, the utility must be implemented in the server, to minimize communication and data transfer overhead. Ideally, the utility should be able to read data directly from the buffer pool, so that large data structures such as the id map and todo lists are not copied after they are read from the disk.

Second, the load must be able to control the maintenance of inverse relationships. Sorting the updates will do no good if the system is also performing the updates, one update at a time, as the other half of the relationships are created.

Third, access to functions not generally available to users must be provided to the load utility. For example, it is uncommon for the user to be able to pre-assign OIDs. The maximal performance of the load algorithm depends on it, however, and the function is easy to supply.

Fourth, if a resumable load is desired, the system must provide the features we identified in Chapter 6: a no-logging option that does not compromise the database's integrity yet allows updates to system data structures, chained transactions or persistent checkpoints so that intermediate states of the load may be committed without losing locks on the loaded data, and a mechanism to remember, during recovery, that a load was in progress at the time of the crash.

Finally, we note that implementing an efficient load algorithm using our techniques will be simple in any OODB that supports value-based joins. If the todo and update lists and query instantiation table are implemented as tables, then existing code may be used to create the lists, to join the id map and todo lists, to sort the updates, and to optimize and evaluate the query functions. Writing the load algorithm is thereby reduced to writing high-level code to perform and coordinate all of the steps.

7.3 Future work

Loading data often involves more than just creating the new objects; there may be integrity constraints to check, triggers to evaluate, and indices to build. It is clear from experiences with relational systems that delaying integrity checks until the end of a load and sorting them before applying the checks, similar to the way we delayed and sorted updates to inverse objects, can save time and reduce duplicate checks [Moh93a]. Although integrity checks may be significantly more complicated in an object-oriented system, and may require evaluating complex path expressions, it is likely that batching integrity checks and trigger evaluations will also yield significant performance improvements.

The complement of loading a database from a file is dumping it to a file. Finding good algorithms for dumping a database poses different challenges than for loading one. Relationships are easy to handle; the current OIDs become surrogates and no translation is necessary. However, the order in which to dump the database objects is an issue. While physically sequential order will probably yield the fastest dump time, it may not yield the desired order of objects in the dumped file. The trade-offs between organizing the objects as they are dumped and processing the data file (e.g., sorting it by the new clustering criterion) after it has been dumped need to be explored.

Furthermore, if the database is dumped, the question of whether to dump indices need to be revisited. In relational systems, the conventional wisdom is to recreate the index when the database is reloaded. However, the primary justification is that it would be too hard to translate the old pointers to objects into new pointers. In an OODB, these pointers are the objects' OIDs, and we already provide a mechanism for translating old OIDs (the surrogates) into new OIDs. The possibility of exploiting this mechanism, the id map, leaves open the question of whether to dump an index or recreate it.

In a larger framework, dumping and reloading a database is one possible method of reclustering. However, there may be better methods of reclustering. To date, there are no studies of reclustering for OODBs, and the possibilities, especially for databases that use logical OIDs and can easily relocate objects, are wide open.

Finally, as users start to have gigabytes and terabytes of data, even the best single-threaded load algorithm will take far too long to load the database. Parallel load algorithms need to be developed, leaving open the questions of how to partition the work and when to coordinate various phases of the load. In addition, there will be trade-offs between performing steps of the load at the sites where the objects will be stored, and distributing the work as evenly as possible over all of the resources.

Bibliography

[Cam95] John Campbell, January 1995. Personal correspondence.

[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.

[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. In *Proceedings of the Seventh Intenational Conference on Scientific and Statistical Database Management*, Charlottesville, VA, September 1994.

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

[Day87] U. Dayal. Of Nests and Trees: A Unified Approach to Processing Queries that Contain Nested Subqueries, Aggregates, and Quantifiers. In *Proceedings of the International Conference on Very Large Data Bases*, pages 197–208, 1987.

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

[DKO+84] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation Techniques for Main Memory Database Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1–8, 1984.

[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.

[DNB93] David J. DeWitt, Jeffrey F. Naughton, and Joseph Burger. Nested Loops Revisited. In *Proceedings of the Symposium on Parallel and Distributed Information Systems*, San Diego, CA, January 1993.

[FBC+90] D.H. Fishman, D. Beech, H.P. Cate, E. C. Chow, T. Connors, J. W. Davis, N. Derrett, C. G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M. A. Neimat, T.A. Ryan, and M. C. Shan. Iris: An Object-Oriented Database Management System. In S. B. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*, pages 216–226. Morgan-Kaufman, Inc., San Mateo, CA, 1990.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan-Kaufman, Inc., San Mateo, CA, 1993.

[GW87] R. A. Ganski and H. K. T. Wong. Optimization of Nested SQL Queries Revisited. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 23–33, 1987.

[Ill94] Illustra Information Technologies, Inc. *Illustra User's Guide*, June 1994.

[Inf94] Informix Software, Inc. *Informix Guide to SQL*, December 1994.

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

[Kim82] W. Kim. On Optimizing an SQL-like Nested Query. *ACM Transactions on Database Systems*, 7(3):443–469, September 1982.

[Kim94] W. Kim. UniSQL/X Unified Relational and Object-Oriented Database System. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, page 481, Minneapolis, MN, 1994.

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

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

[MHL+92] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.

[MN92] C. Mohan and I. Narang. Algorithms for Creating Indexes for Very Large Tables Without Quiescing Updates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 361–370, 1992.

[MN93] C. Mohan and I. Narang. An Efficient and Flexible Method for Archiving a Data Base. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 139–146, 1993.

[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.

[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*, pages 167–185. Morgan-Kaufman, Inc., 1990.

[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.

[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. 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.

[PS88] J. Park and A. Segev. Using common subexpressions to optimize multiple queries. In *IEEE Conference on Data Engineering*, pages 311–319, 1988.

[RS87] L. A. Rowe and M. Stonebreaker. The POSTGRES Data Model. In *Proceedings of the International Conference on Very Large Data Bases*, pages 83–96, 1987.

[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, September 1989.

[Sel88] T.K. Sellis. Multiple Query Optimization. *ACM Transactions on Database Systems*, 13(1):23–52, March 1988.

[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.

[Syb92] Sybase, Inc. *Command Reference Manual*, release 4.9 edition, 1992.

[Ter91] Teradata Corporation. *Fast Load User's Guide for Network-Attached Systems*, release 4.1.1/4.1.2 edition, October 1991.

[Ube94] M. Ubell. The Montage Extensible DataBlade Architecture. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, page 482, Minneapolis, MN, 1994.

[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.

[WCK93] A. Witkowski, F. Cariño, and P. Kostamaa. NCR 3700 — The Next-Generation Industrial Database Computer. In *Proceedings of the International Conference on Very Large Data Bases*, pages 230–243, 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.

[Wit94] Andrew Witkowski, October 1994. Personal correspondence.

[WN94] J. L. Wiener and J. F. Naughton. Bulk Loading into an OODB: A Performance Study. In *Proceedings of the International Conference on Very Large Data Bases*, pages 120–131, Santiago, Chile, 1994. Morgan-Kaufman, Inc.

[WN95] J. L. Wiener and J. F. Naughton. OODB Bulk Loading Revisited: The Partitioned-List Approach. In *Proceedings of the International Conference on Very Large Data Bases*, Zurich, Switzerland, 1995. Morgan-Kaufman, Inc. To appear.