Incremental Loading of Object Databases[*]

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

Abstract

Object-oriented and object-relational databases (OODB) need to be able to load the vast quantities of data that OODB users bring to them. Loading OODB data is significantly more complicated than loading relational data due to the presence of relationships, or references, in the data. In our previous work, we presented algorithms for loading new objects that only share relationships with other new objects. However, it is frequently the case that new objects need to share relationships with objects already in the database.

In this paper we propose using queries within the load data file to identify the existing objects and suggest using parameterized functions to designate similar queries. We then propose a novel evaluation strategy for the queries that defers evaluation until all the queries can be evaluated together. All of the instantiations of a single query function can then be treated as a join between the function parameters and the collection over which the function ranges, rather than evaluated as individual queries. We implement both traditional one-query-at-a-time evaluation strategies and our new strategy in a load algorithm for the Shore persistent object repository and present a performance study showing that the new strategy is at least an order of magnitude better when there are many relationships to objects already in the database.

1 Introduction

As object-oriented and object-relational databases (OODB) attract more and more users, the problem of loading the users' data into the OODB becomes more and more important. Loading is currently a bottleneck in many OODB applications [CMR+94], and the current methods of loading are inadequate.

Although relational database systems provide a load utility to efficiently load large numbers of objects, the current commercial object-oriented databases [Ont94, Obj92, LLOW91, Ver93] do not. Relationships between objects complicate the process of loading object-oriented data, and render relational solutions inadequate. In our previous work [WN94, WN95], we addressed many of the problems posed by relationships in loading a new database. 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.

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

[†]Author's current address: Dept. of Computer Science, Stanford University, Stanford, CA 94305 or wiener@cs.stanford.edu

- 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 [WN95].

In this paper, 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 2; the research OODB systems Iris [FBC+90] and Postgres [RS87] both support query functions, as does Illustra [Ill94].

The main contribution of this paper 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.

1.1 Related work

Our previous work [WN94, WN95] focuses on loading objects into OODBs, but as we mentioned, we ignored the problem of connecting new objects to existing objects until now.

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.

1.2 Structure of the paper

The remainder of the paper is structured as follows. In Section 2, we present an example schema and give examples of query functions. Then we present our algorithms for query evaluation, and describe how to modify a load algorithm to incorporate query evaluation in Section 3. Section 4 describes our implementation, and we present our performance study and results in Section 5. We conclude and outline our future work in Section 6.

2 Loading example

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

2.1 Example database schema

```
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 1: University database schema definition in ODL.

The example schema defines the relevant portions of the university database mentioned in Section 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 1 using the Object Definition Language proposed by ODMG [Cat93].

2.2 Example query functions

In Figure 2, 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]

[1]The range clause should specify a homogeneous collection, but the collection need not be a type extent; another top-level collection or a collection inside another object might be used instead.

```
define query function find_student(X) as
select s
from Student s
where s.name = X;

define query function find_course(Y,Z) as
select c
from Course c
where c.dept.name = Y and c.number = Z;
```

Figure 2: Query function definitions.

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

Figure 3: Example instantiation of a query function.

However, we have replaced the constants in the query predicates with parameters of the query function. The parameters (X, Y and Z, above) represent values that will be filled in by an instantiation of the query function. Figure 3 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.

The order of the queries in the data file must not be significant. That is, we require that the answer to a query should not change based on whether or not certain objects described in the data file have already been created. 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 one object is returned. For our experiments, if more than one object matched the query, we chose one at random. Also, because queries are intended to select one object, we only consider equijoin predicates. Different algorithms would be required for non-equijoin queries. 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.

2.3 Example data file

In Figure 4, we show a portion of an example data file which uses the query functions defined in Figure 2. The complete data file also contains the query function definitions in Figure 2. 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

```
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 4: Example data file using query functions.

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.

3 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 our previous work [WN94, WN95] 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 [WN95], 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.

3.1 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 present the partitioned-list algorithm, modified to use immediate-evaluation to handle queries in the data file. A more complete description of the partitioned algorithm (without queries) is available in our earlier work [WN95]. Parts of step 1 and step 3 are new; the rest of the algorithm remains unchanged. We continue to use an *id map*, *todo list*, *inverse todo list*, and *and update list* to keep track of relationships between new objects.

The id map correlates the surrogates and OIDs of new objects; when an OID is assigned to an object, we insert a new entry into the id map containing the object's surrogate and OID. The todo list keeps track of surrogates seen in the data file, that must be translated to the corresponding OIDs to store in objects. When a surrogate is read in the data file as part of an object's description, we generate a todo list entry containing the OID of the object to update (the object being described), the surrogate indicating what to store in the object, and some information about where and how to store it. The inverse todo list is similar: it keeps track of surrogates corresponding to objects to update, and the OIDs that must be stored in them. The partitioned-list algorithm translates the surrogates into OIDs by joining the todo list and inverse todo list with the id map. The resulting entries, containing OIDs of objects to update and OIDs to store in them, comprise the update list.

In addition, we use a separate update list to keep track of updates to existing objects caused by new bidirectional relationships they share with new objects. These entries similarly contain the OID of the object to update, the OID to store in it, and information (such as offsets) describing how and where to store it.

The complete algorithm, with details of the new steps, is as follows.

1. Read the data file, and create the id map, todo list, and inverse todo list entries.
    - 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. 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.
3. If necessary, repartition the id map, todo list, and inverse todo list.
4. Join the todo list and inverse todo list with the id map to create the update list for new objects.
5. Sort the update list.
6. Read the data file and sorted update list concurrently. Create the new objects; each object will contain data for all of its relationships when it is created — no updates to new objects are necessary.

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 work, we looked for ways to reduce or eliminate the redundant and random I/O; this led to the deferred-evaluation strategy.

[2]If the objects have physical OIDs, we use the OIDs in the update entries as sort keys. If not, we 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. Then we use the physical locations as sort keys.

3.2 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* 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 complete load algorithm using deferred-evaluation is now as follows; only the detailed steps are new.

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.

    - For each query function, allocate an instantiation table.
    - 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.

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.

   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.

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; each object will contain data for all of its relationships when it is created — no updates to new objects are necessary.

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, the actual object may not be in memory when the join is performed, and physically consecutive objects will probably not be listed consecutively in the original update list, especially if the object collection is partitioned for the join. 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.

3.3 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.

4 Experimental configuration

4.1 Implementation

We implemented all of the algorithms in C++. The database was stored under the Shore storage manager [CDF+94]. We ran all of the algorithms on a Hewlett-Packard 9000/720. The database volume was a 2 gigabyte Seagate ST-12400N disk controlled exclusively by Shore. The data file resided on a separate disk and thus did not interfere with the database I/O. The buffer pool for each experiment was 4 Mb.

For these tests, we turned logging off, as would be expected for a load utility [Moh93]. By applying our resumable load algorithm [WN95], we are able to turn off logging of existing data safely, i.e., such that the load can be undone after a system crash or transaction abort, or can be resumed from the last checkpoint. In general, the log documents both the actions of each transaction, and the order in which they occurred. Since the load data file completely describes the actions of the load, and the objects being updated by the load cannot be read or written by any other concurrent transactions, the log becomes redundant to the load file and checkpoint record. A small set of state information (which is kept in the checkpoint record) plus the created objects and other data structures on disk sufficiently describe how far the load has progressed and how to resume or undo it if the load is interrupted before completion.

4.2 Data characteristics

Both the new and the existing database objects were 200 bytes. The schema for each new object contained ten bidirectional relationships. We listed five relationships explicitly with each object in the data file. The other five relationships were their inverses. Four of the five explicit relationships were always shared with another new object. The fifth relationship was either to an existing object (represented by a query instantiation) 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 number of objects to control the sizes of the existing database and the data set being loaded. Each data set had 5000 objects per megabyte (Mb) of data. We also built an index on the existing database collection, so that immed-eval-index could use it. The data files for the new data 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.

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 1: Abbreviations for query evaluation strategies.

We use the abbreviations in Table 1 to refer to the load algorithms containing each of the query evaluation strategies.

5.1 Varying the size of the existing object collection

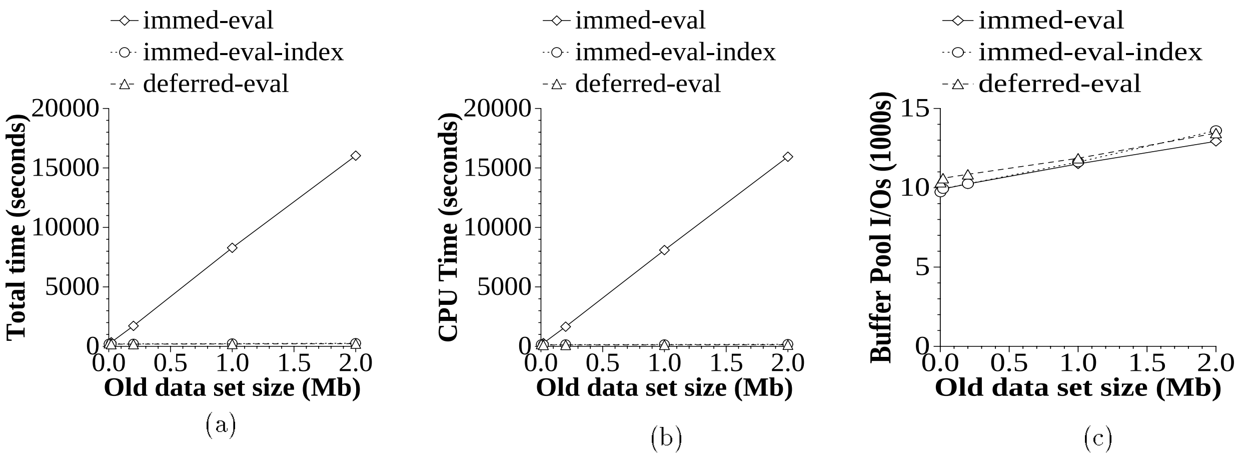
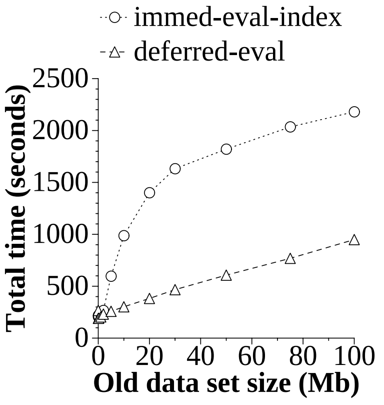


Figure 5: Comparing all of the query evaluation strategies for different existing database sizes.

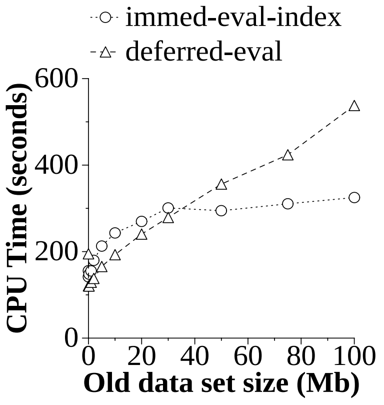
For the first set of experiments, we varied the size of the existing object collection from 100 objects (0.02 Mb) to 10 Mb. We held the size of the data set being loaded constant at 10 Mb. Each new object description contained exactly one query, which targeted an existing object chosen at random from the collection.[3] Figure 5 shows the results for all three query evaluation strategies when the existing object collection fits in the buffer pool, and Figure 6 shows the results for immed-eval-index and deferred-eval when it does not.

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-

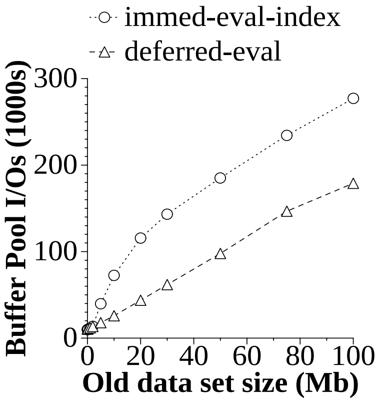
[3]By chosen at random, we mean that when the data file was generated, the constants in each query instantiation were chosen at random from the range of values held by existing objects. Each query in the data file targeted a specific object in the existing object collection.



(a)



(b)



(c)

Figure 6: Comparing the better query evaluation strategies for different existing database sizes.

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 addition, if immed-eval-index were really calling a query processor to perform each query, its cost would have been much higher [Abi95], especially relative to deferred-eval, which would call the query processor only once. We did not have a query processor for Shore, so all of those calls (and their corresponding costs) were circumvented.

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.

5.2 Varying the number of new objects loaded

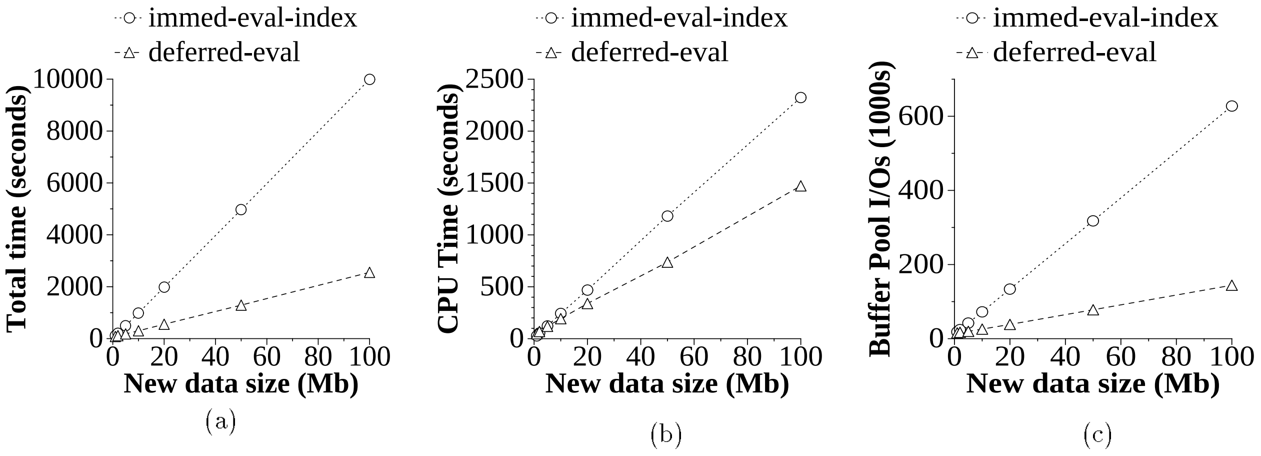


Figure 7: Comparing query evaluation strategies for different new database sizes.

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 7 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 7(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.

5.3 Varying the number of queries

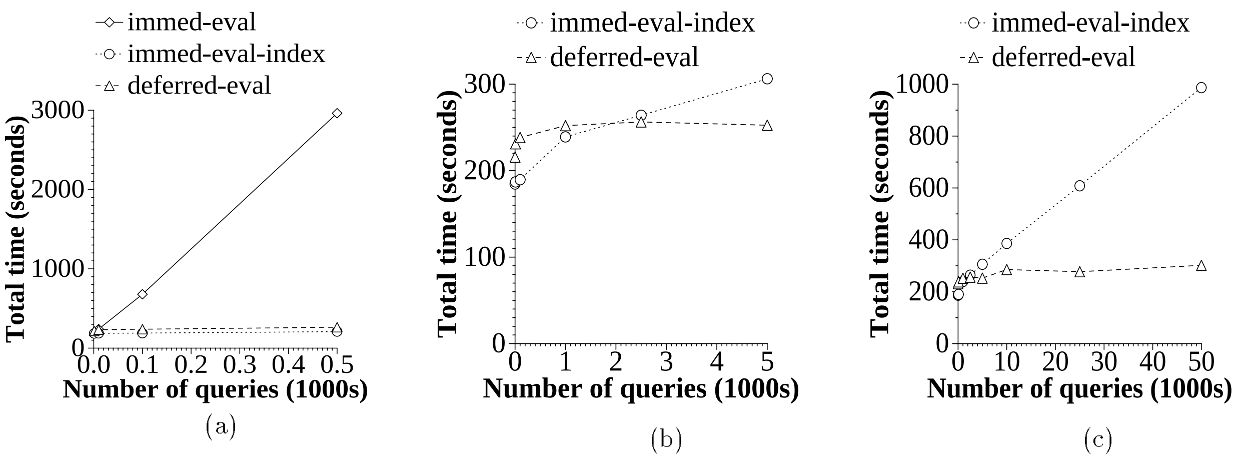


Figure 8: Comparing query evaluation strategies when varying the number of queries.

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 8: in Figure 8(a) we show the times for all the strategies for 10 to 500 queries, in Figure 8(b) we show the times for immed-eval-index and deferred-eval for 10 to 5000 queries, and in Figure 8(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 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.

5.4 Varying the number of distinct queries

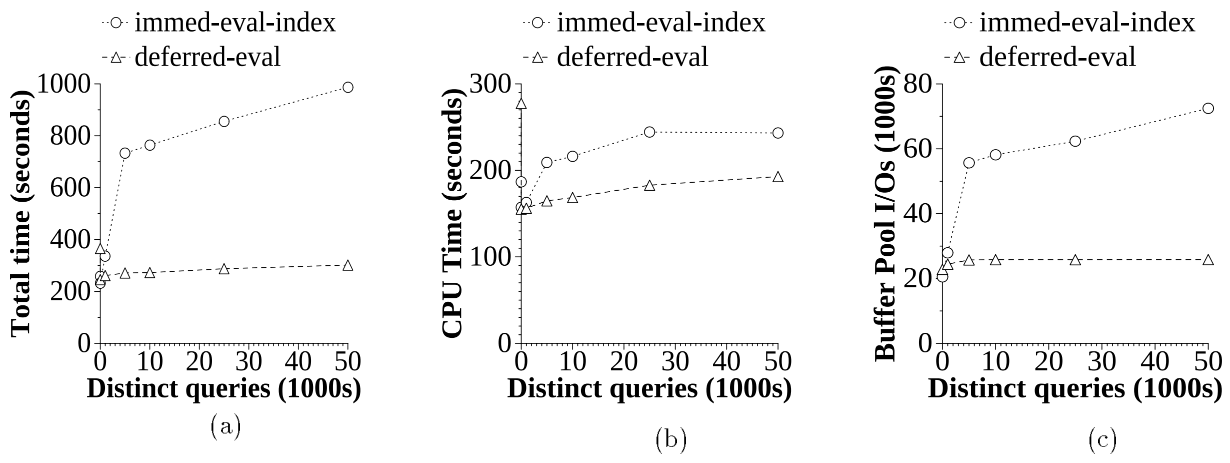


Figure 9: Comparing query evaluation strategies when varying the number of distinct queries.

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 9.

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

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.

6 Conclusions

A bulk loading utility is critical to users of OODBs with significant amounts of data. 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 paper, 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.

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.

Our future work on loading includes investigating when it would be better to dump and load an index (along with the rest of the database) rather than regenerating the index entries during the load; our preliminary cost model shows that it may be cheaper to dump the index. We are also looking into reclustering algorithms that dump and then reload objects and adding smart integrity checking to the load algorithms. We are also interested in algorithms for loading objects in parallel on one or more servers with multiple database volumes. In addition, we are integrating the load implementation with the higher levels of Shore and turning it into a utility to be distributed with a future release of Shore.

7 Acknowledgements

We would like to thank Dave Maier for originally suggesting that loading into OODBs would make an interesting resesarch topic, and for encouraging us to use queries to identify existing objects. A valuable discussion with Odysseas Tsatalos and Mark McAuliffe led to the deferred-evaluation strategy, and Mike Carey noted that, as a load data file writer, he would much prefer to write query function instantiations than full queries throughout the data file. Serge Abiteboul and the OGI OO Database Reading Group provided substantive comments on an early draft of this paper. We are also indebted to Mike Zwilling, C.K. Tan, Nancy Hall, and Dan Schuh for help with our Shore implementation.

References

[Abi95] Serge Abiteboul, September 1995. Observation based on experiments with the O2 OODB.

[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.

[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.

[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.

[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.

[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.

[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.

[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.

[Moh93] 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.

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

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

[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.

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

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

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

[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.