Context-Based Prefetch for Implementing Objects on Relations

Philip A. Bernstein Shankar Pal David Shutt

Microsoft Corporation
One Microsoft Way, Redmond, WA 98052-6399
{philbe, shankarp, dshutt}@microsoft.com

Abstract

When implementing persistent objects on a relational database, a major performance issue is prefetching data to minimize the number of round-trips to the database. This is especially hard with navigational applications, since future accesses are unpredictable. We propose using the context in which an object is loaded as a predictor of future accesses, where context can be a stored collection of relationships, a query result, or a complex object. When an object O's state is loaded, similar state for other objects in O's context is prefetched. We present a design for maintaining context and using it to guide prefetch. We give performance measurements of its implementation in Microsoft Repository, showing up to a 70% reduction in running time. We describe variations that selectively apply the technique, exploit asynchronous access, and use application-supplied performance hints.

1 Introduction

One way to implement persistent objects is to map them to a relational database system (RDBMS). This approach has two main benefits: it provides persistent object views of existing relational databases; and it allows an RDBMS customer to build new object-oriented databases without introducing a new database engine, which avoids changes to database administration procedures and interoperability problems with existing applications. The approach is even more attractive with object-relational DBMSs, which support more of the desired object functionality in the database engine itself. The main disadvantage of mapping objects to relations is performance, which for many common usage scenarios is well below that of object-oriented database systems (OODBs) that use storage servers designed explicitly for object-oriented access.

An important feature of persistent object implementations (on any kind of storage system) is the ability to load persistent objects as active main memory objects, using the object model of the application environment (e.g., C++, Java, Smalltalk, OMG CORBA, or COM). This minimizes the impedance mismatch between the language and DBMS [11], but creates performance challenges for the database implementation, especially when mapped to an RDBMS rather than a custom storage system.

One major performance problem is that application object models are inherently navigational. That is, objects have references or relationships to other objects, which applications follow one at a time. Caching of recently-accessed objects is helpful to avoid accessing the RDBMS too often. But even with caching, if each access to a non-cached object entails a round-trip to the RDBMS, performance will be unbearably slow.

To get a feeling for the performance penalty of round-trips to an RDBMS, consider the following simple experiment. Define a relational database consisting of one table, whose 100,000 rows are 100 bytes each. Each row has a 16-byte ObjectID column which has a clustered index, three 24-byte string-valued columns, and three 4-byte integer-valued columns. Suppose the application knows which ObjectID values it wants, and it retrieves the rows for 100 randomly selected keys in batches of 1, 20, or 100 rows. Using a warm server cache to factor out the cost of disk accesses, we ran the experiment on an RDBMS product and retrieved 580 rows/second, 2700 rows/second, and 3200 rows/second for batch sizes 1, 20, and 100 respectively.[1] This corresponds to a retrieval time of 170 ms (milliseconds), 37ms, and 31ms for 100 rows.

Proceedings of the 25th VLDB Conference
Edinburgh, Scotland, 1999

[1] All experiments in this paper use commodity hardware. The hardware and software configurations are left unspecified, to avoid the usual legal and competitive problems with publishing performance numbers for commercial products. All performance measurements are averages over multiple trials, with more trials for higher variance measurements.

In this case, it is up to 5.5 times faster to get rows (i.e., object states) in a batch of 100 rows rather than a-row-at-a-time.

To minimize the performance penalty of database round-trips, applications often issue a query to identify the objects of interest, and then scan the resulting cursor, one object at a time. This tells the DBMS which objects to retrieve in batch, but often leaves open which pieces of the objects' state are desired, and hence which tables to access. Moreover, for many simple popular navigational patterns, such as following a relationship, it's a nuisance to issue a query. The application programmer would be happier to navigate from one object to the next, accessing the objects as it needs them, letting the DBMS automatically determine what data to prefetch. Programming interfaces for most OODBs, such as the ODMG model [7], satisfy this desire by offering both navigational and query access. But how can the DBMS figure out what to prefetch in response to navigational access? This is the central question addressed by this paper.

Some OODBs use page servers [15]. When accessing an object, the page-oriented OODB (POODB) retrieves the page containing the object, and therefore prefetches other data on the same page. Thus, by clustering data that will be accessed together on the same page, a POODB ensures effective prefetching. That is, the clustering approach amounts to a static prefetching algorithm. Clustering and prefetching are dual problems.[2]

Suppose that whenever a navigational application gets an object O for the first time, it shortly thereafter accesses most of the objects on O's page — the best case for a POODB, whose performance in this case is hard to beat. By comparison, when implementing objects on an RDBMS, there's an additional query processing step to find the same records that the POODB clustered on O's page. Even if the RDBMS clusters the records the same way, it still takes more time to find them and gather them up for transmission to the application than the POODB, which simply ships the page.

Even if the POODB's clustering strategy is optimal for the average workload, access patterns vary and the page-oriented prefetching will make mistakes. Sometimes it will make useless prefetches, where the fetched data isn't subsequently accessed. At other times it will miss prefetch opportunities, because a predictable access pattern hits objects that are mostly on different pages. These mistakes are inherent in the architecture: static clustering of records and page-oriented accesses.

A system that maps objects to an RDBMS (we'll call it an *OMRDB*) probably cannot match the POODB for access patterns that follow the POODB's physical data clustering. However, it may be able to earn back some of that lost performance, in two ways. First, since the OMRDB uses a row server, not a page server, it can prefetch an arbitrary combination of rows. That is, it can use knowledge of recent application behavior to identify combinations of rows worth prefetching, and then use its powerful query processor to find and retrieve those rows in one round-trip, whether or not the data is physically clustered. By contrast, a POODB generally retrieves pages from the server on demand. For a given data layout, both OMRDB and POODB will retrieve the same number of pages. But query-based prefetching allows an OMRDB to prefetch rows before they're referenced, reducing latency. Second, if the density of desired rows on each page is low, then the OMRDB will make better use of client cache than a POODB (since it prefetches only desired rows) and will use less network bandwidth to transfer prefetched rows. Of course, OODBs that use object servers rather than page servers can use this OMRDB tactic, so they too can benefit from the techniques described in this paper.

When prefetching objects, an OMRDB has two related decisions to make: which objects to prefetch and which portions of those objects' state to prefetch. To illustrate our technique, we focus first on the latter question with a simple example (see Figure 1). Suppose an application accesses a relationship R on object O, which returns a set of objects, S. Suppose the state of each object in S is spread across multiple tables. The application may not access all of that state of each object. To avoid prefetching state that the application does not need, the OMRDB delays deciding which state to prefetch. Instead, it simply retrieves the object IDs of the objects in S (making a round-trip to the RDBMS) and waits to see what the application does next. Suppose the application selects an object O′ in S (which is now in application cache) and accesses attribute A in O′. This requires another round-trip. But rather than just retrieving A for O′, the OMRDB retrieves (prefetches) A for all objects in S. This is useful if the application later accesses A for many of those other objects in S, a very common access pattern in workloads we have observed.

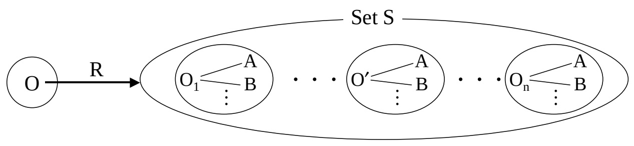


Figure 1 Simple Example of Context-Based Prefetch

Notice that the prefetch decision is based on the application's access pattern — it is not statically determined. O′ could be a member of many collections in the database. The decision to prefetch A for all objects in S is based on the fact that O′ was fetched as part of S, and not

[2] Pointed out to us by Michael Franklin.

some other collection. The OMRDB must remember this fact, to use S as the basis for prefetching A. This is the core idea of our prefetching technique: The OMRDB uses the context in which each object is accessed as a predictor of other objects that will be accessed later.

To implement this approach, the OMRDB creates a *structure context* for each object, which describes the structure in which the object was fetched. Examples of “structures” are stored collections of relationships, query results, and complex objects. When accessing some state of an object O, the OMRDB prefetches the same pieces of the state for other objects in O’s structure context, as in the example of accessing O′’s attribute A for all objects in O′’s structure context S. We call this approach *context-controlled prefetch*. As in the above example, the approach is beneficial whenever all objects in a context undergo similar manipulation.

The rest of the paper is organized as follows. Section 2 summarizes related work on the general problem of implementing OMRDBs efficiently. Section 3 presents the basic mechanisms for context-controlled prefetch. Since the technique is not cost effective in all situations, Section 4 proposes performance hints to selectively enable the optimizations. A summary of our implementation in Microsoft[3] Repository version 2.0 (in Microsoft SQL Server 7.0) is discussed in Section 5. Performance measurements in Section 6 show up to a 3-fold speedup due to these optimizations. Section 7 describes extensions for asynchronous prefetch, lazy loading of objects, and prefetching across paths. Section 8 is the conclusion.

2 Related Work

Although much has been published on the implementation of persistent objects, very little of it uses an RDBMS as the underlying store. Keller *et al.* provide a good overview of the issues [14]. Most papers assume that the set of objects to be retrieved is defined by a query, such as [16, 17, 18, 23], where the issues are running the query efficiently and assembling the objects in the OMRDB, and caching the query result for reuse with later queries [13]. Navigational access is applied to the result of the query, so there’s nothing to prefetch. Descriptions of some commercial products that map objects to relations can be found in [19, 22, 24, 27].

Proponents of OODBs have published many white papers to show that their products outperform a similar implementation on RDBMSs, but little of this has made it into the scientific literature. A useful bibliography is [9].

Prefetching architectures based on recent reference behavior are described in [12, 21]. Palmer and Zdonik use pre-analyzed reference traces to guide prefetch [21]. Curewitz, Krishnan and Vitter use compression algorithms to guide page prefetch based on recent reference behavior [12]. Both approaches work when the exact same objects or pages are retrieved again and again. By contrast, our approach works for any reference sequence that conforms to our generic navigational pattern, even the first (and possibly only) time the objects are accessed, and allows application programmers to influence prefetch decisions. The approaches appear to be complementary and could potentially be used in the same system, a possible subject for further investigation.

3 Using Structure Context

3.1 Object Model

To describe details of the approach, we need to define an object model. We choose one that is similar to those in common use, such as the ODMG model [7], COM [26], and UML [3, 25]. The approach is largely insensitive to the details of the model used here. It should work well for any model that groups objects into structures.

Each persistent object has a persistent *state* that consists of attributes. Each attribute value can be a *scalar*, an *object*, or a *set*.

- Each scalar-valued attribute conforms to a *scalar type*, which gives the name of the attribute and its data type, such as string, integer, or Boolean.
- Every object has a scalar-valued ObjectID attribute that uniquely identifies the object.
- Each object-valued attribute is one side of a *binary relationship*. That is, each relationship consists of two objects that refer to each other.
- Each set-valued attribute contains an object of type set, which in turn contains a set of either scalar values or object references. The concept of set is a representative example of a generic structure type. Other structure types would be handled analogously to sets, such as sequence, array, table, or record structure, but we do not consider them here.

Each object conforms to an object type. Each *object type* has a name and a set of attribute types it can contain. Each binary relationship conforms to a *relationship type*, which gives the name of the relationship, the two object types that can be related, and for each of the two object types, the attribute name by which the reference is accessed.

A *class* is a body of code that implements one or more object types. It includes a class factory that produces objects that are instances of the class. It also includes code that implements the usual read and write operations

[3] Microsoft is a trademark of Microsoft Corporation.

on all of the attributes and structures of the object types that the class implements.

3.2 Operations

The set of navigational object-oriented operations that we consider are GetObject, GetAttribute, GetNext, and ExecuteQuery. These are meant to be a representative sample of the kinds of navigational operations found in programming interfaces for persistent object systems.

- GetObject(ObjectID) returns a running copy of the persistent object whose unique identifier is ObjectID.
- O.GetAttribute(AttrName) returns the value of the attribute AttrName from object O. (The notation O.M means execute method M on object O.) The result is a scalar, object, or set, depending on the attribute type. A set has an associated cursor, initially pointing to the set's first element.
- S.GetNext either returns the scalar or object identified by set S's cursor and advances the cursor, or, if the cursor points beyond the end of S, returns Null.
- ExecuteQuery(Q) returns the set of objects that satisfy query Q's qualification (as in OQL [7, 10]).

3.3 Database Schema

An OMRDB maps objects to rows of tables. A class maps to a table whose columns represent its single-valued attributes. Our optimizations are applicable independent of the rules used to map attributes of a class to a particular table. However, for completeness, we give a few details of mappings that are commonly used.

The simplest mapping is to map a class to exactly one table that contains all of the class's attributes. But more complex mappings are also popular. For example, suppose class B inherits from a class A. If both classes are concrete (i.e., have instances), then there are separate tables for B and A. A's columns, which B inherits, may be stored in both A's and B's tables (Figure 2(ii)), or only in A's table (Figure 2(iii), sometimes called "vertical partitioning" [14]). In the latter case, B's state is reconstructed by joining A's and B's tables. If A is abstract, then its columns might only be stored in tables of concrete classes that inherit from it, in which case it has no corresponding table (i.e., in Figure 2, only store Table T_B, sometimes called "horizontal partitioning").

We assume each many-to-many relationship type is represented in a "junction" table. There could be a separate junction table for each relationship type, with columns SourceObjectID and TargetObjectID, or a generic junction table for all relationship types with columns SourceObjectID, RelationshipTypeID, and TargetObjectID. Each one-to-many relationship can be represented either in a

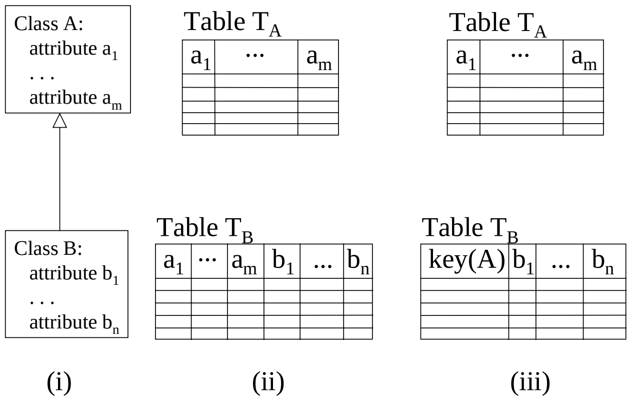


Figure 2 Mapping Classes to Tables

junction table or as a foreign key on the "many side." E.g., if the one-to-many is parent-child, then the foreign key to the parent is stored in the child.

An attribute consisting of a set of scalars can be stored in a table with columns ObjectID, AttributeID (short form of the attribute name), and Value. If the set's maximum cardinality N is known, it could instead be stored as columns of the class's table, such as AttrName_1, …, AttrName_N. Since these two table structures are isomorphic to one-to-many relationships and single-valued attributes (respectively), the prefetch scenarios for a set of scalar attributes are isomorphic to those other two cases as well and therefore are not treated further in this paper.

3.4 The Prefetch Pattern

As discussed in Section 1, the main approach is to maintain a structure context (or, more simply, a *context*) for each object, which describes the structure in which the object was loaded, and to use that context to guide later prefetch decisions. In this section, we explain one usage of the approach in detail. We reapply this usage to other operations in the next section.

Consider the following operation sequence:

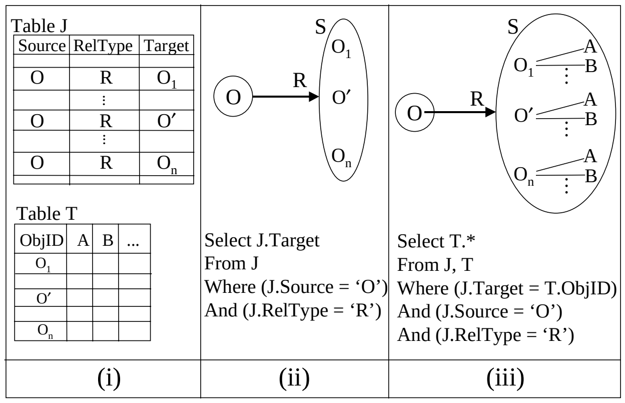
a. S = O.GetAttribute(R), which returns a set S of objects, which is the value of relationship attribute R.

b. O′ = S.GetNext, which returns an object O′ in S.

c. V = O′.GetAttribute(A), which returns the value V of scalar-valued attribute A of O′.

This is the scenario of Figure 1. Attribute A corresponds to a column of a table, T, containing (some of) the state of O′'s class, C. Unless T is very wide (e.g., has lots of long columns), it costs little more to retrieve all of T's columns that are part of C's state than to retrieve only A. This is because most of the cost is in the disk accesses, which retrieve all of the columns from disk whether or not they are fetched by the OMRDB. Thus, if there is a good chance that some of those columns will be accessed, then it is worth prefetching all of those columns of T for O′. Moreover, since O′ was retrieved via S, if we expect other objects in S to be accessed similarly to O′, then we should prefetch all of those columns of T for all objects in S, not just for O′. This avoids later round-trips to the database for each object in S.

The optimization is illustrated in Figure 3. Table T is shown in Figure 3(i), with relationships, such as R, implemented by a junction table J. Steps (a) and (c) above are illustrated in Figure 3(ii) and (iii) respectively. Notice that step (c) uses the same selection clauses for J as step (a), and then joins with T to get the columns of T for *all* objects in O.GetAttribute(R), not just for O′.



Source	RelType	Target
O	R	O_1
	⋮	
O	R	O′
	⋮	
O	R	O_n

ObjID	A	B	...
O_1			
O′			
O_n			

```
Select J.Target
From J
Where (J.Source = ‘O’)
And (J.RelType = ‘R’)
```

```
Select T.*
From J, T
Where (J.Target = T.ObjID)
And (J.Source = ‘O’)
And (J.RelType = ‘R’)
```

Figure 3 Prefetching Scenario

The experiment in Section 1 suggests this prefetch is profitable if at least 4-5 items in the collection are later accessed (since batch retrieval is 4-5 times the cost of a single-row retrieval). However, the addition of object-level processing reduces the fractional contribution of DB round-trips to total cost. Thus, in our experiments, across a variety of workloads and database profiles, the prefetch is profitable if at least 3 items are subsequently accessed (for our combination of data server, network, etc.). The more items that are accessed, the more round-trips that are saved by the prefetch, hence the greater the benefit.

If objects in S have state in two tables, T and T′, is it worth getting attributes from both of them? Usually not. For example, we extended the experiment in Section 1 by duplicating the table, to model T and T′. Getting 100 rows from both tables in one round-trip added 50% to the execution time over retrieving only from T. So the cost of needlessly getting T′’s columns is high. Also, the benefit is modest, since retrieving those rows from both tables was only 19% slower in two round-trips vs. retrieving the join in one round-trip. Thus, one should only retrieve T′’s columns if they are almost certain to be needed.

Since the technique heavily uses cache, the interactions of prefetching and cache management need careful attention. E.g., if the cache is nearly full or if the data to prefetch is very large, then the prefetch may not work well. We will see other examples of this later.

To generate the SQL query shown in Figure 3(iii), the OMRDB needs the context of O′. The context should include the information that was used to create the set S initially, namely, the object ID of the relationship’s source, O, and the relationship name, R. These are the parameters that are needed to construct the SQL query, which retrieves the desired attributes of all objects in the context.

We expect it is worth supporting variations of this pattern where only attribute A or a predefined subset of attributes is retrieved, and not all the attributes of its table. However, as this is only a slight variation of our main idea, we don’t consider it further in this paper.

3.5 Case Analysis

We generalize Section 3.4 to other navigational access patterns that suggest future navigational behavior. These suggestions lead naturally to prefetch recommendations, which retrieve the data needed to service the later accesses before those accesses occur. Of course, recent navigational accesses don’t guarantee that those later accesses will occur, so prefetch recommendations must be applied selectively, an issue we will discuss in Section 4. For now, we simply describe potentially useful prefetches and how to implement them, for each of the operation types in Section 3.2.

In the following descriptions, we omit the initial test to determine if the requested data is already in cache and therefore does not need to be fetched.

GetObject(ObjectID) – Prefetch some or all of the object’s state. Note that to load the object into main memory, the OMRDB only needs to know the object’s class (to know what class to instantiate) and that the object exists (to know whether to return an error). Prefetching other object state is optional at this stage. Set the object’s context to null, which means it was loaded directly, not as part of a larger structure.

O.GetAttribute(AttrName) – where AttrName is the name of an attribute in O’s class. There are six cases to consider, Case (i) – Case (vi) below:

Case (i) If AttrName is a single-valued object reference, and O’s context is null, then get the reference to the object from the database. If the object reference is stored as a foreign key in a table containing other attributes of O, then retrieve those other attributes too.

For example, suppose AttrName is a many-to-one relationship R from O's class C to class D (see Figure 4). Suppose C's table T_C(ObjID, A_1, …, A_n, FKey) contains scalar attributes A_1, …, A_n of C and foreign key attribute FKey that contains the object ID of an object in D related via R. Since getting the reference to O_D in D involves accessing the FKey column of O's row in T_C, prefetch the other attributes A_1, …, A_n of that row too.

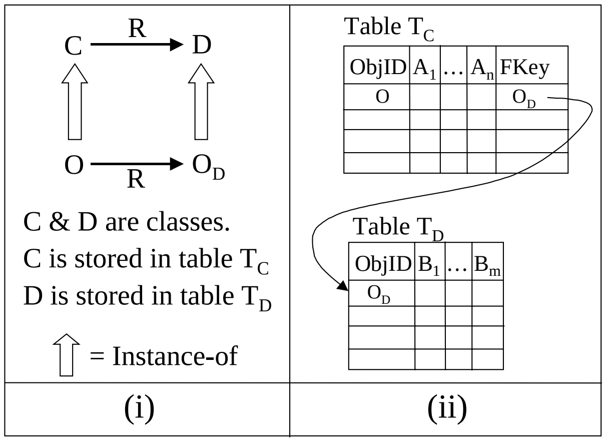


Figure 4 Prefetching Attributes with a Foreign Key

If T_C is indexed on the compound key [ObjID, FKey], rather than just ObjID, then the query processor can get O_D without accessing O's row, making it cheaper to retrieve FKey by itself. If the index on [ObjID, FKey] is non-clustered, this raises the incremental cost of getting A_1, …, A_n, which should therefore be prefetched too only if there's a high probability of subsequent access.

Case (ii) If AttrName is a scalar and O's context is null, then simply retrieve the attribute. As in Case (i), retrieve other attributes of O's state that are in the table containing AttrName's column. Here and in Case (i), prefetching those other attributes is cost-beneficial if some of them are subsequently referenced.

Case (iii) If AttrName is an object reference, and O's context is a set S′, then prefetch the reference for *every* object O′ in S′, not just for O. That is, run one SQL statement that returns the ObjectID value of AttrName for all objects in S′. As in Case (i), if the object references are stored as foreign keys in a table containing other attributes of O′, then retrieve those other attributes too. Prefetching the object reference for other objects in S′ is cost-beneficial if at least several other objects in S′ are subsequently accessed. This case is essentially the same as Figure 4, except that multiple rows of T_C are retrieved (one for each object in S′) instead of just one (for O).

Case (iv) If AttrName is a scalar and O's context is a set of objects, then O.GetAttribute(AttrName) corresponds to step (c) in Section 3.4. Do the prefetch described there and in Figure 3(iii).

Case (v) If AttrName is set-valued and O's context is null, then retrieve the set's content for O, which is a set S of either scalar values or objects. In the latter case, assign S to be the context of each object in S.

If S is a set of objects and the object references are stored as foreign keys in the referenced objects, then retrieve other attributes of the referenced objects stored in the same table as the foreign key. Modifying the example of Figure 4 so that AttrName is a one-to-many (rather than many-to-one) relationship R from O's class C to class D, we get Figure 5, where B_1, … B_m can be retrieved along with objects in D referenced by O. As usual, prefetching those other attributes is cost-beneficial if some of them are referenced later for at least several other objects in S.

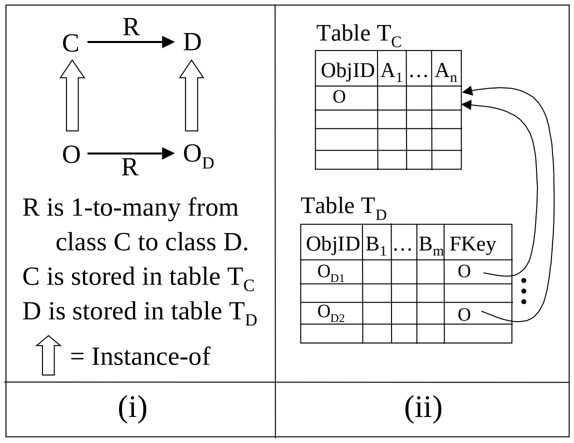


Figure 5 Prefetching Attributes of Referenced Objects

Case (vi) If AttrName is a set (of scalars or objects) and O's context is a set S′, then prefetch the AttrName set for every object in S′. For example, suppose AttrName is a relationship R to a set of objects. Then execute *one* SQL statement that returns a table of <$ObjID_1$, $ObjID_2$> pairs, where $ObjID_1$ denotes an object in S′ and $ObjID_2$ denotes an object referenced by $ObjID_1$ via R, thereby prefetching Object-IDs of members of the R-set of *every* object in S′. This is analogous to Section 3.4, where the scalar attributes of all objects in S′ are prefetched. For the R-set S_i referenced by each $ObjID_{1,i}$ in S′, assign S_i to be the context of each object in S_i. The prefetch is cost-beneficial if those R-sets are accessed for at least several other objects in S′.

Continuing the example of Figure 5, if R is one-to-many from O's class C to class D, and is represented as a foreign key in D's table T_D, then the attributes of objects in D can be prefetched too, modulo the additional cost, depending on whether FKey is part of the compound index on T_D.

ExecuteQuery(Q) – We apply context-based prefetch to the set that results from the execution of a query, just as we do for a set of stored object references in cases (iii) and (iv). We could do this by executing the query, saving the resulting ObjectID's in a set, and sending the ObjectID's to the RDBMS on each prefetch. However, this is inefficient for large query results, and problematic since the query that does the prefetch could exceed the maximum size of a SQL statement or stored procedure call. We could re-execute the query on each prefetch, but this too would be inefficient unless the query is cheap. Therefore, we store the context on the server in a temporary table.

So, assume each session has a temporary table TEMP(SetID, ObjectID) in the DBMS. Associate a unique SetID, s, with the query. Map the query into an Insert statement that appends rows <s, o> to TEMP for each ObjectID o in the result of the query. Execute the Insert and return the ObjectID's of the query result into a set, S, also identified by s.

Creating this context is cost-beneficial if attributes of the objects in S are subsequently accessed, making prefetches of those attributes cost-beneficial, and if the query is too expensive to re-execute to perform the prefetch.

TEMP can be created any time after starting the database session, but no later than the first call to ExecuteQuery.

Like any query result, a query context in TEMP can be used across transaction boundaries only if the application is using read-committed, not repeatable-read, isolation.

S.GetNext – Return the designated element of S. Prefetching was accomplished when S was loaded, i.e., in GetObject(ObjectID) or O.GetAttribute(AttrName).

3.6 Managing Structure Context

Structure context is part of the state of a loaded object. The context's lifetime is governed by that of the object. Thus, when the object is released, the context is deallocated too. This includes context information that is maintained with the object by the OMRDB, typically in main memory. Also, when releasing a set that was the result of an ExecuteQuery, it includes deleting rows of the temporary table containing the cached result of the query. Like any deallocation, the latter can be done lazily and asynchronously with respect to other processing.

Like persistent database tables, the performance characteristics of temporary tables must be carefully analyzed when using them to cache query results. For example, depending on how space is managed, it may be important to preallocate temporary table space. It may or may not be valuable to have an index on SetID, depending on how queries are processed, the size of query results, and how many query results are concurrently active.

Suppose an object is loaded multiple times via different navigational paths. For example, an application might load object O via O′.GetAttribute(AttrName), where AttrName is a set S containing O, and later reload O via GetObject(o), where o is O's ObjectID. In some programming interfaces, the later operation does not return the same loaded object, but rather returns another copy of the same persistent object. If so, then one would expect the two objects to share their cached persistent object state, since they refer to the same object. However, since the two objects were loaded via different paths, they should have different contexts. In the example, the first object's context is S, while the second's is null.

4 Performance Hints

Since the prefetch optimizations of Section 3.5 are not always cost-effective, it is important to be able to enable and disable them. Enabling an optimization is essentially a performance hint to the underlying OMRDB. This general approach of application-supplied hints has been adopted by other object-to-relational mapping systems, such as [20], and in other commercial products [8].

The main optimizations that are worth controlling in this way are the following:

1A. **A**ttributes for **1** object - When accessing an object O, prefetch all of O's scalar attributes.

1R. **R**elationships for **1** object - When accessing an object O, prefetch all of O's references to other objects.

MA. **A**ttributes for **M**any objects - When accessing a scalar attribute A of an object O, prefetch all of the attributes in A's table for all objects in O's set context.

MR. **R**elationships for **M**any objects - When accessing a relationship attribute A of an object O, prefetch the objects related via A for every object in O's set context.

Some contexts are very large — too large to prefetch the attributes or relationships for every object in the context. One could specify a threshold for context size, above which the MA or MR prefetch is disabled. Or, it may be better to stream in the prefetched data in batches. This requires finding a query that retrieves a subset of the data to be prefetched plus a remainder query that identifies the remaining data to be prefetched. This isn't always doable. For example, in MA, to prefetch attribute values for a large unordered set, we would need a column value that partitions the set into non-overlapping subsets.

It is often appropriate to enable a prefetch optimization for an application's entire execution, but sometimes it is not. For example, if an application accesses all members of some sets but only one member of others, MA will speed up accesses to the former and slow down accesses to the latter. Thus, it is beneficial to enable and disable the optimizations dynamically during its execution.

Performance hints can be added as tags in an information model to specify the default prefetch behavior of a class. For example, a class C could be tagged to enable MR but disable MA. Such default behavior can be overridden by the application.

Although prefetch is enabled and disabled dynamically, it is still beneficial to maintain context for each loaded object. This makes it possible to perform context-based prefetch on an object that was loaded when prefetching was disabled. Since maintaining context is cheap, there is little benefit to disabling it, with one possible exception: The result of a query is cached in a temporary table for use as the context of each object in that result. Since adding the result to the temporary table has non-negligible cost, it is probably worth disabling in cases where it is known to be ineffective.

Although the manual control that performance hints offer is worthwhile, it would be even better if the system could automatically decide when to enable and disable prefetching. One approach is to run the application on sample data to generate traces, and then analyze the traces to determine whether each type of optimization is cost effective for a given run. Another approach is to statically analyze the application to predict certain access patterns. For example, a common pattern is to call GetAttribute(R), where R is a relationship that returns a set of object references, and then loop through the result, accessing attributes of each object. In this case, the program would be modified to enable MA before entering the loop. Even finer grained control is possible here, since the exact set of attributes that will be referenced could be made known in the hint, thereby reducing the cost of the prefetch.

A cache control mechanism would also be beneficial, to reduce the amount of prefetching when the OMRDB's cache is stressed. The cache manager could track the amount of free space, for this purpose.

5 Implementation in Microsoft Repository

The prefetch optimizations of Section 3 are implemented in the latest version of Microsoft Repository, whose storage engine is a persistent object layer implemented on top of Microsoft SQL Server. The product's object model, table layout, and API do not include all of the alternatives in Sections 3.1 - 3.3. The differences that are relevant to the prefetch optimizations being considered here are:

- All relationships are currently stored in a single junction table, like Table J in Figure 3(i), not as foreign keys in class-oriented tables.
- The object model is COM, where classes implement interfaces and interfaces have single-valued scalar properties and relationship-valued collections. Each interface's scalar properties are stored in (i.e., are columns of) one table, each of whose rows contain state for an object whose class implements the interface. Each table can store the properties of many interfaces, but a class's interfaces need not all be stored in one table.
- There is an additional method, ObjectInstances, that gets the set of all objects that are instances of either a given class or a class that supports a given interface. Objects that are retrieved by this method have the retrieved set as their context, which is represented by the identity of the class or interface. This is very similar to Case (v) in Section 3.5, O.GetAttribute(AttrName), where AttrName designates a set of object references.
- Each time a persistent object P is loaded, a new COM object is created, which shares its cached state with other COM objects that represent P. The context is stored in the COM object, since each load operation for P may be via a different navigational path and therefore have a different context to guide prefetch.

The object model, table layout, and API are described in detail in [1, 2].

6 Performance Measurements

The prefetch optimizations as implemented in Microsoft Repository have been useful for many customer applications we have tested, so they are frequently enabled — especially MA, which we have found is almost always beneficial. However, it is hard to report on these in a way that bears scientific scrutiny, since each application yields a varying workload that is hard to characterize succinctly. We therefore ran some more controlled experiments to show how the optimizations work in practice.

6.1 Experiment 1 – The 007 Benchmark

A good test to show the benefits of MA is the 007 benchmark [4, 5, 6], since it is a highly regular workload that is representative of many persistent object applications (a brief summary is in the Appendix). We ran the 007 queries and traversals using Microsoft Repository on the medium sized 007 database (225MB). We ran with a cold server cache, to ensure the optimizations are fully penalized for useless prefetches. We did not run 007 structural modifications, since their behavior is not affected by our optimizations.

Queries Q1-Q3 and Q7 are all of the form: retrieve a set of objects (in our case using ExecuteQuery) and access the objects' state. Q8 also accesses state, to form a subquery for each object in an outer query (it retrieves a set of object pairs). So they all benefit from optimization

MA, which prefetches the objects' state in one round-trip based on the cached query result. Figure 6(i) reports the percentage improvement in running time over an execution with no prefetch optimizations enabled (i.e., ((old-new)/old)×100). The benefit varies based on the size of the result and the algorithm for joining the temp table (containing cached query results) with the attribute table at the server. Query Q4 is of the same form as Q1-Q3, but the query plan for the prefetch is sub-optimal, making MA ineffective. None of the queries benefit from 1R or MR, because they don't access any relationships, nor from 1A, because only one attribute of each object is accessed. Query Q5 simply counts the number of objects in the result of a query, so there's no benefit to prefetching properties or relationships.

007 Test	Benefit of MA
Q1	87 %
Q2	17 %
Q3	54 %
Q4	-39 %
Q5	0 %
Q7	73 %
Q8	42 %

(i)

007 Test	Benefit of MA
T1	1 %
T2a	-11 %
T2b	34 %
T2c	31 %
T3a	-12 %
T3b	34 %
T3c	39 %
T6	0 %

(ii)

Figure 6 Benefit of Prefetch in 007

Traversals T1-T6 navigate relationships starting from a root. Like the queries, traversals T2b,c and T3b,c access attributes and therefore benefit from MA. T1 and T6 do not access attributes, so MA has no effect. T2a and T3a access only one object in each collection, so prefetching attributes for all objects is a cost that has no compensating benefit. Thus, MA decreases their performance.

In principle, traversals T1-T3 should benefit from MR, but they actually do not. The traversals visit the entire bill of materials hierarchy of a base assembly, which includes 655,000 atomic parts. This overflows the client cache, causing prefetched objects to be replaced before being accessed. Currently we do not handle this situation very gracefully in that we let the cache overflow. In a future version, we will have a more adaptive algorithm.

6.2 Experiment 2 – XML Export

Another application with a highly regular workload is the XML export utility that ships with the product. We wrote a program that uses the utility to export a set of objects reachable from a root, by doing a breadth-first traversal from the given root and writing them out as an XML stream. We exported a UML model consisting of 3100 objects and 3900 relationships. The execution time was 267 ms with no optimizations, 117ms with 1R enabled, 220ms with MA, and 78ms with both 1R and MA. That is, the execution was 56%, 18%, and 71% faster with optimizations 1R, MA, and 1R+MA, respectively.

1R helps because in UML each object has many relationship types, all of which are accessed, either to export them or determine that they're empty. MA helps because if a relationship is non-empty, then attributes of all related objects are exported. These benefits are nearly independent, in that the benefit of 1R+MA (71%) is nearly the sum of the benefits of 1R and MA (56%+18%=74%).

6.3 Experiment 3 – Measuring MR

To measure the benefits of MR in a controlled setting, we created an object hierarchy stored as relationships in a junction table like Table J of Figure 3(i), with a clustered index on its key (Source, RelType, Target). The hierarchy has a top level fan-out of 100, and a second level fan-out of 20. We traversed the hierarchy by running SQL. The baseline ran a SQL query to get the 100 children of the root followed by a query for each child to get its 20 children. To model MR, we replaced the 100 queries for grandchildren by one query to get all grandchildren. The latter retrieves the same amount of data as the former, but replaces the fixed overhead of 100 queries by that of just one query. The resulting execution time was 71% faster with MR enabled than disabled.

MR is sensitive to the size of the context to which it's applied, since that affects the number of queries it saves. For example, running the previous experiment on a hierarchy with top level fan-out of 20 and second level fan-out of 100, the benefit of MR is only 33%, less than half as much as the previous case. The same amount of data is retrieved as before, but only 20 queries for grandchildren are replaced by one query. This effect also explains, in part, the lack of benefit of MR in 007 traversals, since all the fan-outs are of size 3. In XML, MR actually hurts performance, probably because the contexts are small and the prefetch requires an extra expensive join.

7 Extensions

7.1 Asynchronous Prefetch

To improve response time further, it is desirable to do the minimal work necessary to return from each data access call and prefetch any additional data asynchronously with respect to the call. To some extent, an application can do this manually by reordering its logic so that the OMRDB gets data before the application actually needs it. Beyond this rather crude recommendation, there are two approaches to making the prefetch asynchronous: pipeline the prefetch or prefetch after processing the request.

In the pipelined approach, the OMRDB issues a query to retrieve the requested and prefetched data together. When the first packet arrives from the DB server, the OMRDB starts populating its cache. As soon as the requested data arrives, the OMRDB returns to the application, but it continues processing packets asynchronously until all the prefetched data have arrived.

If the DBMS supports multiple active statements on a database session, then a second application request can be executed while a previous prefetch is still active. Most DBMSs do not support this, since it requires parallel nested transactions so that concurrent SQL statements can be independently backed out. Without this feature, a second session is needed to avoid delaying an application call while a prefetch is tying up the session. In any case, if the application later asks for an item that is still being prefetched, it is blocked.

In the non-pipelined approach, the OMRDB retrieves the requested data plus a subset of data to be prefetched. After receiving this data, it returns to the application and asynchronously issues a query for the remaining data to prefetch. However, as discussed in Section 4, finding a query that retrieves a subset of the data to be prefetched plus a remainder query that gets the remaining data to be prefetched is sometimes infeasible.

In some cases, it is unavoidable to fetch the entire result before returning to the caller. For example, Microsoft Repository stores sequenced collections as rows linked by back pointers [1]. If an application requests an item of the collection other than the first, then all of the collection elements must be loaded into cache for the engine to determine the requested item. This defeats both the pipelined and non-pipelined approaches.

Finally, statistics are needed to determine how much to prefetch. The goal is to fetch only a small amount to avoid delaying the application much by the prefetch, but fetch enough to keep the application busy while the second, asynchronous prefetch is running.

7.2 Prefetching Across Paths

We can extend context-based prefetch to apply to paths of relationships. For example, consider a persistent object base that contains a database schema definition object, which contains table definition objects, each of which contains column definition objects, each of which is related to a data type definition object. When accessing a column of the first table definition, MR will prefetch column definitions of all table definitions. We showed in Section 6.3 that this can yield significant improvements. It may also be beneficial to prefetch the scalar attributes of those column definitions (i.e., Section 3.5, Case (v)) and their relationships to data type definitions (adding another hop to the path). To estimate the benefit, recall in Section 3.4 that getting 100 rows from two identical tables with 100-byte rows in 2 round-trips is 19% slower than getting the join in one round-trip. One could extend the hints of Section 4 to specify when to prefetch across such a path.

An issue with this approach is the representation of the inherently hierarchical result of the prefetch, when a relational query is used. As observed in [17], in a parent-child relationship, when retrieving all the children of a parent, the parent information is repeated for every child due to the normalization inherent in relational queries. Possible solutions are to return the result in a nested table, a tree structure, or an XML stream (that represents the tree structure), all of which require some extension to the underlying RDBMS.

7.3 Lazy Loading of Sets

Suppose O.GetAttribute(AttrName) returns a set, S, of objects (i.e., Case (iv) and Case (vi) in Section 3.5). One could execute GetAttribute by storing only its definition (i.e., "O.GetAttribute(AttrName)") and not its instances in main memory. If S is used only to insert new objects, then the existing state of S never needs to be loaded, a significant optimization. If existing objects in S are retrieved, then its instances can be loaded on the first invocation of S.GetNext.

Suppose it is known that all objects in S are instances of the same concrete class C (rather than different specializations of C). Now even GetNext can avoid loading instances of S, by creating a hollow instance O′ of C. This can be done using only cached type information, without accessing database instances. The state of O′ is populated only when one of its attributes is accessed. At this point, MA kicks in, which retrieves that attribute (and possibly others in the same table) for all objects in S. This prefetch gets the object IDs of all objects in S as a side effect, which allows S finally to be populated with instances. Thus, the initial round-trip to the RDBMS to get the object IDs of objects in S is entirely avoided, leaving only one round-trip to get the attributes of all objects in S, thus halving the number of round-trips in this scenario.

While this benefit is appealing, unfortunately the line of logic to attain it is not quite sound: If S is empty, then S.GetNext should return Null. If this is known to be impossible (e.g., because the set has an enforced integrity constraint saying it has non-zero cardinality), then the technique works fine. Otherwise, if it can return Null, then it is not valid for it to return a hollow instance of C. Therefore, to benefit from this optimization, a change is needed in the programming interface. There are several options: a hint could be issued before the first call to GetNext, to tell the system what attribute(s) to prefetch; the GetNext call itself could optionally include a list of attributes of interest; or the semantics of GetNext could be modified so that the first GetNext on an empty set returns an object and an exception is raised only when attempting to access one of that object's attributes. The use of a hint strikes us as the best of the alternatives.

8 Conclusion

We described a technique for predicting useful prefetches when a navigational object-oriented interface is implemented on a relational DBMS. We presented a design for the technique and measured its performance in a commercial product, Microsoft Repository 2.0. We proposed a number of extensions, some of which would benefit from further work, such as automatically issuing hints to enable the prefetch optimization and prefetching across paths of relationships.

Overall, there has been much published about efficient implementations of persistent objects. Having worked on an implementation of persistent objects on a relational database for the past several years, we feel that the problem of optimizing the performance of such a system is only partially understood and would benefit from much more research. Given the advent of object-relational DBMSs, and the need to offer persistent object interfaces on top, the importance of this problem is growing.

Appendix – Summary of 007

The 007 benchmark is based on a bill-of-materials database. In the medium database, each assembly has 3 sub-assemblies, and so on through 7 levels. Each assembly has also has 3 composite parts, each of which has an associated document and has 200 interconnected atomic parts. The following queries and retrievals are taken from [6], paraphrased to save space:

Q1 – Given 10 random atomic part id's, get the atomic parts (that exist) and the number retrieved.

Q2 – Given a range of dates containing the last 1% of dates in atomic parts, retrieve the atomic parts.

Q3 – Given a range of dates containing the last 10% of dates in atomic parts, retrieve the atomic parts.

Q4 – Given 100 random document titles, for each document, find all base (i.e., level 1) assemblies that use the composite part corresponding to the document. Also return the number of such base assemblies.

Q5– Find all base assemblies that use a composite part whose build date is later than that of the base assembly. Also report the number of base assemblies found.

Q7 – Scan all atomic parts.

Q8 – Find all pairs of documents and atomic parts where the atomic part's document id equals the document's id. Return the number of pairs found.

T1 – Traverse the assembly hierarchy. For each base assembly, visit its unshared composite parts. For each composite part, do a depth first search on its atomic parts. When done, return the number of atomic parts visited.

T2 – Same as T1, but swap attributes x and y for some of the objects:
a. Update one atomic part per composite part.
b. Update every atomic part encountered.
c. Update each atomic part in a composite part four times.

T3 – Same as T2, but update the (indexed) date field.

T6 – Same at T1, but for each composite part, visit only its root atomic part.

Note that there is no Q6, T4, or T5 in 007. Traversals T7 and T8 are omitted because they run very fast and therefore lead to inaccurate (high variance) measurements.

Acknowledgments

We thank Thomas Bergstraesser and Jason Carlson for many stimulating discussions about the performance of Microsoft Repository. We are also very grateful to Jayaram Mulupuru and Daniel Reib for implementing the approaches described in this paper and to Jim Hance and Gary Miao for measuring them, all of which helped uncover many of the detailed problems we addressed.

References

1. Bernstein, P.A., B. Harry, P.J. Sanders, D. Shutt, J. Zander, "The Microsoft Repository," *Proc. 23rd VLDB Conf.*, 1997, pp. 3-12.

2. Bernstein, P.A., T. Bergstraesser, J. Carlson, S. Pal, P.J. Sanders, D. Shutt, "Microsoft Repository Version 2 and the Open Information Model," *Information Systems 24(2)*, 1999, to appear.

3. Booch, G., J. Rumbaugh, I. Jacobson, *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, MA, 1998.

4. Carey, Michael J., David J. DeWitt, Jeffrey F. Naughton, "The 007 Benchmark," *Proc. 1993 ACM SIGMOD Conf.*, pp. 12-21.

5. Carey, Michael J., David J. DeWitt, Chander Kant, Jeffrey F. Naughton, "A Status Report on the 007 Benchmark," *Proc. OOPSLA 1994*, pp. 414-426.

6. Carey, Michael J., David J. DeWitt, Jeffrey F. Naughton, "The OO7 Benchmark," technical report, ftp.cs.wisc.edu., Univ. of Wisconson, Jan. 1994.

7. Cattell, R.G.G., D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H.

Strickland, D. Wade. *The Object Database Standard: ODMG 2.0*, Morgan Kaufmann Publishers, 1997.

8. Chang, E. E., Randy H. Katz, "Exploiting Inheritance and Structure Semantics for Effective Clustering and Buffering in an Object-Oriented DBMS," *Proc. 1989 ACM SIGMOD Conf.*, pp. 348-357.

9. Chaudhri, Akmal B., "ODBMS Resources," http://www.soi.city.ac.uk/~akmal/html.dir/resources.html

10. Cluet, S., "Designing OQL: Allowing Objects to be Queried," *Information Sys. 23(5)*, 1998, pp. 279-306.

11. Copeland, G. and D. Maier, "Making SmallTalk a Database System," *Proc. 1984 ACM SIGMOD Conf.*, pp. 316-325.

12. Curewitz, K.M., P. Krishnan, J. S. Vitter, "Practical Prefetching via Data Compression," *Proc. 1993 ACM SIGMOD Conf.*, pp. 257-266.

13. Keller, A., J. Basu, "A Predicate-based Caching Scheme for Client-Server Database Architectures," *VLDB Journal 5(1)*, Jan. 1996, pp. 35-47.

14. Keller, A., R. Jensen, and S. Agrawal, "Persistence Software: Bridging Object-Oriented Programming and Relational Database," *Proc. 1993 ACM SIGMOD Conf.*, pp. 523-528.

15. Lamb, Charles, Gordon Landis, Jack A. Orenstein, Daniel Weinreb, "The ObjectStore System," *CACM* 34(10), 1991, pp. 50-63.

16. Lee, Byung Suk and Gio Wiederhold, "Outer Joins and Filters for Instantiating Objects from Relational Databases Through Views," *IEEE Trans. On Knowledge and Data Eng.* 6(1), 1994, pp. 108-119.

17. Lee, Byung Suk and Gio Wiederhold, "Efficiently Instantiating View-Objects From Remote Relational Databases" *VLDB Journal* 3(3), 1994, pp. 289-323.

18. Mitschang, Bernhard, Hamid Pirahesh, Peter Pistor, Bruce G. Lindsay, and Norbert Sdkamp, "SQL/XNF - Processing Composite Objects as Abstractions over Relational Data," *Proc. 1993 Int'l Conf. On Data Eng.*, pp. 272-282.

19. Ontos, http://www.ontos.com

20. Orenstein, Jack A. and D. N. Kamber, "Accessing a Relational Database through an Object-Oriented Database Interface," *Proc. 21st VLDB Conf.*, 1995, pp. 702-705.

21. Palmer, M. and S. Zdonik, "Fido: A Cache that Learns to Fetch," *Proc. 17th VLDB Conf.*, 1991, pp. 255-264.

22. Persistence Software, http://www.persistence.com

23. Pirahesh, Hamid, Bernhard Mitschang, Norbert Sdkamp, and Bruce G. Lindsay, "Composite-object views in relational DBMS: an implementation perspective," *Information Sys* 19(1), 1994, pp. 69-88.

24. POET Software, POET SQL Object Factory, http://poet.com/factory.htm

25. Rational Software Corp. Unified Modeling Language Resource Center. At http://www.rational.com/uml

26. Rogerson, D. *Inside COM*. Microsoft Press, 1997.

27. RogueWave Software, DBTools.h++, http://www.roguewave.com/products/dbtools/