Scalable and Recoverable Implementation of Object Evolution for the PJama$_1$ Platform

M.P. Atkinson, M. Dmitriev, C. Hamilton, and T. Printezis

{mpa,misha,craig,tony}@dcs.gla.ac.uk
Department of Computing Science, University of Glasgow,
17 Lilybank Gardens, G12 8RZ, Scotland

Abstract. PJama$_1$ is the latest version of an orthogonally persistent platform for Java. It depends on a new persistent object store, Sphere, and provides facilities for class evolution. This evolution technology supports an arbitrary set of changes to the classes, which may have arbitrarily large populations of persistent objects. We verify that the changes are safe. When there are format changes, we also *convert* all of the instances, while leaving their identities unchanged. We aspire to both very large persistent object stores and freedom for developers to specify arbitrary conversion methods in Java to convey information from old to new formats.

Evolution operations must be safe and the evolution cost should be approximately linear in the number of objects that must be reformatted. In order that these conversion methods can be written easily, we continue to present the pre-evolution state consistently to Java executions throughout an evolution. At the completion of applying all of these transformations, we must switch the store state to present only the post-evolution state, with object identity preserved. We present an algorithm that meets these requirements for eager, total conversion.

This paper focuses on the mechanisms built into Sphere to support safe, atomic and scalable evolution. We report our experiences in using this technology and include a preliminary set of performance measurements.

1 Introduction

An effective schema evolution technology is essential for any persistent platform. One of the dominant effects of time is change, and in enterprise applications[1], this manifests as changes in requirements, changes in understanding of the application, and changes to correct mistakes [38]. In the case of an object-oriented persistent platform, these changes result in a requirement to change the descriptions of existing objects, both their content and behaviour, to change their instances to conform with the new descriptions, and to introduce new descriptions that will later generate new objects.

[1] Applications that are typically long-lived, complex and large scale, referred to as "Persistent Application Systems" in [7].

G.N.C. Kirby, A. Dearle, and D.I.K. Sjøberg (Eds.): POS-9, LNCS 2135, pp. 292–314, 2001.

Zicari explored this issue in the context of O_2 [44,10]. Odberg identified a classification of object-oriented schema evolution techniques [36] and Wegner and Zdonik have examined the issues of schema edits in OODBs [43,40]. Our work contrasts with theirs, as for orthogonality and consistency PJama$_1$ stores the methods in the store [25,26,5], which provides us with the opportunity of supporting developers by verifying the mutual consistency of the new classes *including the code in methods*. As far as we know, commercial (O)RDBMs and OODBMs that support schema edit, such as GemStone [20], do not attempt such consistency checking. We describe our approach to schema editing covering:

❶ the way that changes are specified,
❷ the set of changes supported,
❸ the consistency checking undertaken, and
❹ the set of object population reformattings that may ensue from such a change.

In previous papers [15,17] we reported our rules for verifying that a set of changes were consistent and our evaluation of that approach using an algorithm that was not scalable, as it depended on building the new store state in main memory. The new contribution of this paper is an algorithm that safely, i.e. after validation checks and atomically, carries out eager evolution. We believe that it has good scalability properties. The algorithm depends on particular properties of our new store technology, Sphere, and is a partial validation of their value.

The challenge that we sought to overcome is presented by the combination of three factors. The algorithm has to be safe, that is any failure must leave the store in its original or final state. It has to be scalable, which means that it cannot rely on major data structures or evolved object collections fitting into main memory. And it has to be complete and general purpose. By this we mean that it has to accommodate any changes to any population of objects. That in turn requires that we must be able to run conversion code developed by application developers during evolution. In order that this code is tractable for application developers, the initial state of the store must remain stable and visible throughout the evolution. The new state appears atomically after all of their code has been executed.

The principal topics of this paper, are the algorithm and the support it obtains from Sphere and measurements demonstrating scalability. This support from Sphere includes incremental scanning, atomicity and durability. We provide a discussion of evolution models and a summary of our previous work (⇝2). We present PJama$_1$, an orthogonally persistent platform (⇝3), followed by an introduction to Sphere (⇝4). This is followed by a description of our *eager* evolution algorithm (⇝5). We present initial performance measurements (⇝6) before reviewing related work (⇝7) and offering our conclusions (⇝8).

2 Categories of Evolution

There are two important contexts in which evolution is required: *development evolution* (⇝2.1) and *deployed evolution* (⇝2.2).

2.1 Development Evolution

When developers are working on a persistent application they will frequently want to test it against persistent data. They therefore work on a collection of data and classes that represents the currently relevant aspects of the eventual enterprise application. At this time, they usually require up to a few hundred megabytes of data to achieve representative behaviour and run trials. For example, when testing the geographic application of persistence (GAP, ⇝6.3) we use a few counties of the UK or California, rather than the whole UK or USA data.

It is typical of this phase of use, which may include user trials, that changes occur very frequently. The developer then needs to change classes and install them and the consequential changes in the experimental store. As any users involved are aware that this is a development system, it is acceptable to interrupt the prototype service if one exists. Normally, using the evolution technology proves much faster than rebuilding the store. However, we have observed developers still rebuilding the store as opposed to using the evolution technology, and conclude that convenience is crucial to the use of evolution technology in this context. We are therefore working on making the evolution technology easier to use e.g. integrating it with build and compilation technology [16].

2.2 Deployed Evolution

Once developers ship a version of an enterprise system we expect it to be in constant use at a large number of customer sites. These customers, or other bespoke software vendors, will develop their own software (classes) and populate their stores with the shipped classes, their own classes and instances of both sets of classes. Meanwhile, the original developers will have fixed bugs or provided new facilities, and will need to ship the revised classes to the stores of customers that want to obtain bug fixes or a new release. This requires a different treatment from development evolution.

❶ *Validation and Preparation*: As much as possible of the validation and preparation must be completed at the developer's site, based on configuration management information recording the exact versions of classes shipped to each customer.
❷ *Defining the Transformations*: Some optimised "pre-compiled" form of all of the transformations of previously shipped classes would be assembled at the developer's site. The developer would take responsibility for information migrating to the new forms, just as an office-tool or CAD-tool vendor does today.
❸ *Packaging the Change*: The results of the previous two steps have to be packaged into a "self-installing" unit, that can be shipped to a customer site and activated by the customer.
❹ *Installing the Change*: Once activated, the change must install against very large collections of data, without excessive disruption of a customer's workload. That is, the evolution must operate safely and concurrently with the

existing workload and the customer must be able to meet an urgent requirement by limiting evolution's resource consumption, interrupting or terminating the evolution if necessary.

As far as we know, it is not yet possible to perform *validated, safe and non-disruptive, deployed* evolution in any system. It would, of course, need to be incremental, to limit disruption, and in most cases would need to be partial, so that some code can continue to work with earlier versions. This paper is concerned only with *development* evolution, which we believe can usefully be eager and total, provided that it is scalable and safe.

2.3 Development Evolution Requirements

The goals for development evolution technology have been described [15,17]. Here we present a summary.

- ❶ *Durability*: A primary aspect of safety is that evolution should never leave a persistent object store (POS) in an inconsistent state. That is, any failure must either leave the POS unchanged or an evolution must complete i.e. evolution should be treated as atomic. Either of these may utilise recovery on restart.
- ❷ *Validity*: Best efforts must be made to detect developers' mistakes as early as possible, in order that preventable errors are not propagated into the POS. In particular, the classes in a PJama_1 store after an evolution must be completely *source compatible* [21], and all instances must conform to their class's definition.
- ❸ *Scalability*: Any evolution step should be completed in a reasonable time, and with reasonable space requirements, taking into account the nature of the changes requested. For example, the resources used should, at worst, be approximately proportional to the volume of data that must be updated in the POS.
- ❹ *Generality*: Whatever change developers discover they need, they should be able to achieve it. This includes any transfer of information between the pre-evolution and post-evolution state that they require, even if its representation is fundamentally different.
- ❺ *Convenience*: Developers should be able to achieve evolution of a POS using tools and procedures that are as close as possible to their normal methods of working. The amount of additional concepts that they have to understand and the number of additional steps they have to take, should both be minimal.

Durability (❶) is achieved using logging ($\rightsquigarrow$4); but excessive logging has to be avoided as it would impact scalability (❸). Generality (❹) and convenience (❺) are achieved by allowing developers to present new or redefined classes that have been obtained in any way they wish, either as source Java or classes in bytecode form, e.g. obtained from a third party. They can also explicitly change the class hierarchy.

General transport of information from the pre-evolution to post-evolution state is supported by allowing developers to write *conversion methods* in Java. This requirement to execute arbitrary, developer-supplied code during evolution imposes three technical requirements on our implementation:

- A Java Virtual Machine (JVM) [28] must be available to execute this code (at present we use PJama$_1$, $\rightsquigarrow$3). It must support all of the normal PJama$_1$ semantics — particularly, the creation of **new** objects that then become persistent because they are reachable in the post-evolution state.
- Failures, e.g. uncaught exceptions, in this code must result in the whole evolution step being rolled back (requirement ❶).
- This code must have understandable semantics. We choose to guarantee that the pre-evolution data state remains visible and unchanged throughout the evolution transaction, so that algorithms that scan data structures do not encounter partially transformed data. We also allow access to the post-evolution state, selected via a class naming scheme.

2.4 Lessons from Version 1

The previously-reported work [15,17] emphasised the opportunity and need to verify an evolution before applying it. In order that developers can use their usual tools, we allow them to edit and recompile classes (or replace classfiles) in the usual way. The change involved in an evolution step is then defined by the difference between these new class definitions and the old definitions[2].

For any case where the default transformation between the old and new instance format is not satisfactory, a developer can provide a conversion method written in Java. During evolution, for each instance of the specified class in the old state, this method is called and supplied with that instance in its original state, and the new instance after default transformation[3]. While these methods are executing, they may traverse structures reachable from the old version of the instance and they may build arbitrary new structures reachable from the new version of the instance.

Before proceeding with any changes to the store, a verification step analyses all of these classes and methods (and explicit changes) to make certain that they are compatible. For example, it checks that there aren't still methods in the new definitions (or in the classes that are unchanged) that potentially use a discontinued member of a class. Only when it is clear that no predictable inconsistencies will be created between classes or between classes and data in the store, do we proceed with evolution. All of these changes we retain and reinforce.

The previous and current algorithm, identify the subset of changed classes whose instances need to be visited, and then they perform a scan of the store,

[2] In addition, the developer can specify explict class removals, class insertions and class renaming.

[3] There are actually variants, for example to allow the developer's method to choose which subclass to construct.

visiting those instances and supplying them to conversion methods if necessary. The previous algorithm simply faulted each old object into main memory, allocated a new object and performed the transformation. It kept all of the new objects in main-memory until the scan was complete, and then committed them to the store in a single checkpoint, fixing any references that were once to the old versions to refer to the new versions. This had two non-scalable features. The whole of the set of new versions and their dependent structures, and the fix-up table had to fit in main memory. The whole of the set of old versions and the whole of the set of new versions and the dependent structures of both sets had to fit in the persistent object store simultaneously.

3 Aspects of PJama$_1$

A full description, rationale and review of PJama is available [5], which also summarises progress since the original proposal [6,3][4]. Here we select a few aspects of the latest version, PJama$_1$.

PJama is an attempt to deliver the benefits of orthogonal persistence [7] to the Java programming language [21]. It attempts to capitalise on the established experience of persistence [9] and on the popularity and commercial support for Java.

PJama$_1$'s aims to provide: orthogonality, persistence independence, durability, scalability, class evolution, platform migration, endurance, openness, transactions and performance. Evolution inherits requirements from these general requirements, in particular: orthogonality, persistence independence and openness lead to generality (❹) and validity (❷). Durability, scalability and performance are directly inherited.

PJama$_1$ was constructed by combining the Sunlabs Virtual Machine for Research (previously known as ExactVM or EVM) [41] with Sphere. This architecture is illustrated in figure 1 and further described in [30,27]. Much of the evolution code is written in Java and uses reflection facilities over the POS.

4 Sphere's Support for Evolution

Sphere is definitively described in [37]. We refer to the application that is using Sphere as the *mutator*, which may be some C or C++ application, or an application in some other language and its virtual machine. The latter is the case when we consider PJama$_1$. The code, written in Java, that organises evolution, is an example of a mutator, from Sphere's point of view.

4.1 Overview of Sphere's Organisation

Sphere permits an object to be any sequence of fields, where a field can be a scalar of 1, 2, 4 or 8 bytes or a pointer of 4 bytes[5].

[4] An intermediate review appeared as [25,4]

[5] Sphere can be recompiled for 64-bit pointers, but cannot operate with a mixture of address sizes.

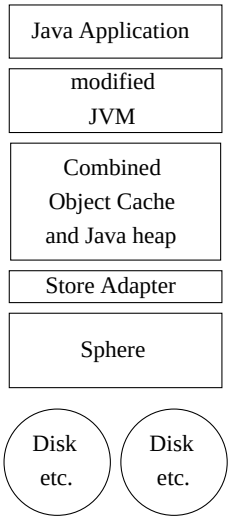

Fig. 1. PJama$_1$ Architecture — Sphere supports PJama

Objects are grouped into *partitions*. Each partition contains a data structure, called the *indirectory entry*, for each object that it holds. This provides one level of indirection to each object. The indirectory entries in a partition are grouped in a table called the *indirectory*.

Each partition is managed by a *regime*. This regime determines how space is administered. Objects are allocated in partitions that are appropriate for their size, number of pointers, etc.

Each object is identified by a unique *persistent identifier* (PID). A PID consists of a pair ⟨`LP#, IEI#`⟩ where `LP#` is a *logical partition identifier* and `IEI#` is an *indirectory entry index* used to address the entry in the indirectory.

When necessary, an object holds a reference to a *descriptor*. Descriptors are auxiliary objects that describe the layout (e.g. location of reference fields) of objects of the same type (i.e. in the case of PJama, instances of the same class). A descriptor is lazily replicated in each partition that contains objects that need to be described by it and is then shared by these objects. A descriptor is eliminated automatically from a partition when the garbage collector removes the last object that it describes. Hence the presence of a descriptor implies instances in this partition and its absence implies that there are none of its instances here. A descriptor index in each partition supports a rapid test for the presence of a particular descriptor.

Partitions are introduced to permit incremental store management algorithms, such as evolution and disk garbage collection. The partition's regime selects particular versions of their increments to apply. A mapping between physical and logical partitions is used to permit simple atomic transitions, as in Challis' algorithm [11,12] or in shadow paging [1,13,35].

4.2 Recovery Mechanisms

The recovery mechanism is based on the ARIES write-ahead logging algorithms [33,32,31] so that we can exploit logical logging and other techniques for using relatively small amounts of data to support durability [22].

4.3 Sphere's Support for Evolution

The technology to support evolution is described in [23] and can be summarised as follows.

- The use of a partition and regime structure to selectively scan the store linearly and incrementally.
- The use of the *descriptor invariant* to rapidly discover whether any instances of a class exist within a partition and hence to rapidly identify the extent of the classes to be transformed, without having to maintain exact class extents during other computations, which are presumed to dominate processing.
- The use of the disk garbage collector's algorithms to manage identity and copying.
- The use of the logical to physical partition number mapping to achieve durable atomic transitions without excessive log traffic.
- The use of hidden *limbo* versions of objects ($\rightsquigarrow$5) to
  - avoid log traffic,
  - avoid PID allocation,
  - hold the old and new state simultaneously at a cost proportional to the actual changes, and
  - to reveal the new state atomically at the conclusion of evolution.

5 An Eager Evolution Algorithm

We will use the following terminology. We refer to objects that are undergoing evolution as *evolving objects*. During the execution of the conversion methods and for some period thereafter, these objects exist simultaneously in their old form, *old object*, and in their new form, *new object*. During part of this co-existance, they are both referred to by the same *PID*, to avoid the costs of allocating additional *PID*s, and in the post-evolution state the new object has the identity previously owned by the old object, which has now disappeared. While they share the same *PID*, the hidden new state is referred to as a *limbo object*.

The main steps in an evolution are the following (see [23]).

❶ *Specify the Set of Class Changes and Instance Transformations*: Developers generate, or obtain in the case of third-party code, a set of revised classes and ask the evolution tool to install them in a specified store. They must also define methods to take information from old instances of a class to new instances of that class, or some related class, whenever the default transformations will not suffice.

❷ *Analyse and Validate the Set of Changes*: The PJama$_1$ *build tool*, which includes a specialised version of the standard Java source compiler[6] is used to verify the consistency of the new set of classes. We define consistency as follows: the classes should be mutually *source compatible* [21]. This means that it must be possible to compile their source files together without problems. The tool maintains records of the classes used with and contained in the store, and compares those with the classes presented and reachable through the current `CLASSPATH`. It then selectively recompiles those classes for which the source has changed. If a class does not have source code (e.g. it belongs to a third-party library), this should be explicitly confirmed by the developer. After recompilation, the build tool compares the resulting classes with their original versions saved in the store. If a class has changed in a potentially incompatible way (e.g. a public method has been deleted), the tool forces recompilation of all of the classes that might be affected by this change. In the above case that would be all of the classes whose old versions called the deleted method, and which haven't yet been recompiled. If recompilation fails, the whole evolution is aborted. Therefore an inconsistent set of changes can never be propagated into the store.

Changes to the class hierarchy are performed implicitly, by changing the **extends** phrase of the classes definition. However, if the developer wishes to delete a class, say `D`, completely, they have to explicitly specify this operation.

An outcome of this analysis will be a set of class replacements, $\mathcal{R}$, of the form `C` $\mapsto$ `C'`, a set of new classes, $\mathcal{N}$, a set of classes to be deleted, $\mathcal{D}$, and a set of transformations, $\mathcal{T}$, (developer-supplied or default). For each member of $\mathcal{R}$ modified such that the format of its instances is changed, there must be a corresponding member of $\mathcal{T}$ or there must be no instances of the class currently in the store. This latter property is verified immediately, since Sphere performs that check very quickly. Similarly, for each class in $\mathcal{D}$, there must either be no instances of that class in the store, or a method must be specified in $\mathcal{T}$ to *migrate* all of its "orphan" instances to other classes.

We enter the next phase with a set, $\mathcal{CV}$, which is all of the classes whose instances must be visited during store traversal. These are the classes for which there exists a default or developer-defined transformation in $\mathcal{T}$. If $\mathcal{CV}$ is empty, skip steps ❸, ❹ and ❺.

❸ *Prepare for Evolution*: Carry out the *marking phase* of Sphere's off-line cyclic garbage collector for the whole store (SGGC, see [37]). This ensures that all of the garbage objects are marked so that they will not participate in evolution and hence be resuscitated by being made reachable.

❹ *Perform all of the Instance Substitutions*: Traverse the store, one populated partition, `p`, at a time, visiting only partitions with relevant regimes, e.g. partitions containing only scalar arrays need not be visited. For each class, `C`, in $\mathcal{CV}$ lookup `C` in `p`'s descriptor table. If no descriptors are found, skip to the next partition. Conversely, if a descriptor for any `C` is found, scan the objects

[6] Specialised to enable it to deal with old and new definitions of classes simultaneously in conversion methods, and to access class information kept in the store.

in `p`. For each non-garbage object that has a descriptor that refers to a class in $\mathcal{CV}$, create a new instance in the format `C'` in a new partition, record the ⟨old-PID, new-PID⟩ pair[7], move data into the new object using the default transformation, and then apply any developer-supplied transformation.
When all of the evolving objects in `p` have been converted, the old and new worlds are co-located into one new partition using a slightly customised partition garbage collector. In the resulting partition, the old form of each instance is directly referenced by its PID but the *limbo* form lies in the following bytes, no longer directly referenced by a PID[8]. The change in logical to physical mapping is recorded in the log, as is the allocation and de-allocation of partitions, as usual for a garbage collection. If phase ❺ does not occur, e.g. due to a crash, the limbo objects are reclaimed by the next garbage collection. Hence the reachable store currently hasn't changed, it has only expanded to hold unreachable limbo objects and new instances reachable only from them.

❺ *Switch from Old World to New World*: At the end of the previous phase all of the new format data and classes are in the store but unreachable. Up to this point recovery from failure, e.g. due to an unhandled `Exception` thrown by a developer-supplied transformation, would have rolled back the store to its original state. We now switch to a state where recovery will roll forward to complete the evolution, as, once we have started exposing the new-world we must expose all of it and hide the replaced parts of the old-world.
A small internal data set is written to the log, so that in the event of system or application failure, there is enough information to roll forward, thus completing evolution. This set essentially records all partitions that contain limbo objects. Each partition in this set is visited and all evolving object/limbo object pairs are swapped i.e. the evolving object is made limbo and the evolved limbo object is made live. The now limbo old objects can then be reclaimed at the next partition garbage collection.
Once this scan has completed, the new world has been merged with the unchanged parts of the old world.

❻ *Complete the Manipulation of Persistent Classes*: The classes that have fallen into disuse are removed.

❼ *Commit Evolution*: Release the locks taken to inhibit other use of this store and write an end-of-evolution record in the log.

The evolution technology is complex, but all aspects of it are necessary. For example, the new versions of instances have to be allocated away from the instances

[7] To permit references to new objects to be fixed up to the corresponding old-object's *PID*.

[8] This has the advantage that no extra PIDs are needed in the new partition and that extra space is only needed for the normally small proportion of instances that have evolved in each partition. There is a slight problem as developer-supplied transformation code may try to revisit these limbo new objects while transforming some other instance, e.g. to form a forward chain. A solution to this, which complicates the Sphere interface, is being considered.

they replace so that they can both be referenced in the developer-supplied Java transformation code. They then have to be moved, so that *only two extra partitions* are needed to complete the evolution, and no extra PIDs are needed in fully populated partitions. The garbage collector expands or contracts partitions as appropriate, when it allocates the destination partition, but it cannot expand a full indirectory. Use of limbo objects to arrange that both old and new objects and their dependent data structures can co-exist in the store avoids writing images of evolving objects to the log. A prepass of the cyclic garbage collection mark phase ❸ is necessary attempts are made to evolve unreachable instances. Occasionally these caused the developer's conversion methods to fail because they contained obsolete data. A beneficial side-effect is that evolution also performs a complete disk garbage collection and the recovered space is available to use during the process.

6 Initial Measurements

We have measured the time that it takes for our system to evolve a store with a certain number of evolving objects. The number of factors on which this depends is large, so we concentrated on the following issues:

- Verify that the time grows linearly with the number of evolving objects.
- Explore the impact of non-evolving objects in the store on the performance, particularly when their number is much greater than that of those evolving.
- Explore how the complexity of the objects and of the conversion code affects the evolution time.
- Validate synthetic tests with some real-life applications.

6.1 Benchmark Organisation

Our benchmark consisted of three synthetic tests, which are summarised in the following table:

Test No.	Description	Objects	Change
1	Simple class	simple	simple
2	001 benchmark	complex	simple
3	001 benchmark	complex	complex

In all three tests we varied the number of evolving objects (denoted n) in the store between 20,000 and 200,000. The second varying parameter was the number of non-evolving objects per evolving object, denoted by g for *Gap*. This varied between 0 (all of the objects in the store are evolving) and 9 (9 non-evolving objects per one evolving object). The objects were physically placed in the store such that evolving and non-evolving objects were interleaved. This is illustrated in figure 2. From the evolution point of view, this is the worst possible store layout, since we have to scan and evolve all of the partitions.

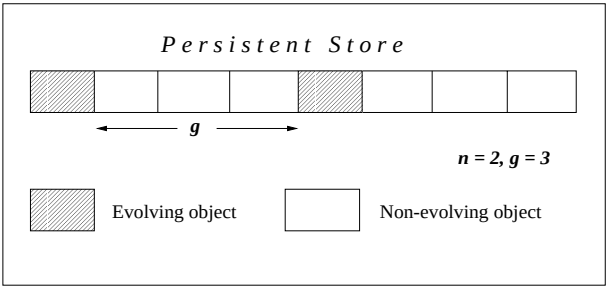

Fig. 2. Test store layout example

All of the tests were run with a constant Java heap size of 24MB (that is the default size for PJama_1), which, in the worst case, is an order of magnitude less than the space occupied by all the evolving objects. The sphere disk-cache size was set to the default value of 8MB.

In test 1 the old and the new versions of the evolving class were defined as follows:

```
// Old version          // New version
public class C {        public class C {
  int i, j;               int i, j;
                          int k;
...                     ...
}                       }
```

Default conversion (a simple method used by PJama if no custom conversion code is supplied) was applied to instances of `C`. According to the rules of default conversion, the values of fields `i` and `j` were automatically copied between the old and the new object, and the `k` field was initialized to 0.

Tests 2 and 3 were performed over stores populated with instances of the class from the adapted version of 001 benchmark, which we have taken from Chapter 19 of [42]. The initial Java version of this class looks as follows:

```
import java.util.LinkedList;

public class Part {
  int id;
  String type;
  int x,y;
  long build;
  LinkedList to;
  LinkedList from;

  // Methods to set/get fields, etc.
}
```

As defined in the 001 benchmark, an object of class `Part` contains a unique `id` and exactly three connections to other randomly selected parts. Instances referenced from the given `Part` instance `p` are stored in the `p`'s `to` list. In turn, all instances that reference `p` in their `to` lists are stored in the `p`'s `from` list, allowing a reverse traversal. The values of other fields are selected randomly from a given range.

In test 2 the only change in the new version of class `Part` was a different type of its `id` field — it was changed to `long`. Java types `int` and `long` are logically, but not physically compatible. This means that the values of the former type can be safely assigned to the fields of the latter, but the size of fields of these types are different (32 bits and 64 bits respectively). So object conversion is required in this case, but default conversion is enough to handle information transfer correctly.

In test 3 a more complex change was applied: the type of `to` and `from` fields was changed to `java.util.Vector`. The objects contained in the list can't be copied into another data structure automatically, so the following conversion class was written:

```
import java.util.LinkedList;
import java.util.Vector;

public class ConvertPart {
  public static void convertInstance(
                Part$$_old_ver_ partOld,
                Part partNew) {
    int toSize = partOld.to.size();
    partNew.to = new Vector(toSize);
    for (int i = 0; i < toSize; i++)
      partNew.to.add(partOld.to.get(i));

    int fromSize = partOld.from.size();
    partNew.from = new Vector(fromSize);
    for (int i = 0; i < fromSize; i++)
      partNew.from.add(partOld.from.get(i));
  }
}
```

As a result of conversion, for each `Part` instance two new objects are created, and six objects are discarded.

In each test run we were invoking our standard build tool that analyses and recompiles classes and then initiates instance conversion. It was only the instance conversion phase which we measured. In each test we measured the *Total Time*, defined as the time elapsed between the start and the end of conversion. We also measured the *Sphere Time*, defined as the time spent within the Sphere calls corresponding to step ❹, part 2 and step ❺ of the evolution algorithm. The difference between these two times was called the *Mutator Time*.

Every test run with the same values of n and g parameters was repeated ten times, and the average time value was calculated after discarding the worst case. All experiments were run on a lightly-loaded Sun Enterprise 450 server with four[9] 300MHz UltraSPARC-II CPUs [39], an UltraSCSI disk controller, and 2GB of main memory. The machine runs the Sun Solaris 7 operating system. The Sphere configuration included a single 1GB segment and a 150MB log. The store segment and the log resided on the same physical disk[10] (9.1GB Fujitsu MAB3091, 7,200rpm [29]).

6.2 The Experimental Results

In tests 1 and 2 we observe completely uniform behaviour, characterised by almost perfectly linear growth of the evolution time with both n and g. In test 1 the minimum and maximum total time values were 1.25 and 57.38 sec, whereas in test 2 they were 2.42 and 75.10 sec, respectively.

These figures show the total time taken during the evolution phase, with a further breakdown indicated at the extremes of both axes. Figure 4 shows the breakdown in more detail for a fixed $g = 0$, varying n. Figure 5 shows the same breakdown, this time for a fixed $n = 200,000$, varying g.

Graphs for all experiments at each value of n and g were generated, yielding the same typical set of results, namely that as the number of evolving objects increases, more of the total time is spent within the Sphere kernel, than within the Mutator.

Linear growth of the time with n means that the scalability requirement for evolution technology is satisfied at this scale. Despite the fixed Java heap size, the time grows proportionally with the number of evolving objects.

The growth of evolution time proportionally with the object gap (this parameter can also be interpreted as the total number of objects in the store), illustrates a trade-off we have made. When a partition is evolved, all of the objects contained in it are transferred into a new partition, thus a comparable amount of time is spent handling both evolving and non-evolving objects. The alternatives are to explicitly maintain exact extents or to segregate classes. Either would impact normal executions, we believe significantly[11].

On the other hand, the current implementation's results are quite acceptable: the time it takes to convert a store in which only 1/10th of objects are actually evolving is only about 4 times greater than the time it takes to evolve the store containing only evolving objects. We also performed experiments with the stores where evolving and non-evolving objects were laid out in the store in two solid

[9] The evolution code makes very little use of multi-threading.

[10] A pessimal arrangement to accentuate effects due to log writes.

[11] With an explicit extent, every object creation has to perform an insert and additional space is required. Such extents increase the complexity of garbage collection, which has to remove entries. Our regime scheme already provides as much segregation as the mutator chooses. We currently segregate into four regimes: large scalar arrays, large reference arrays, large instances and small instances. As segregation is increased, clustering matching access patterns is reduced.

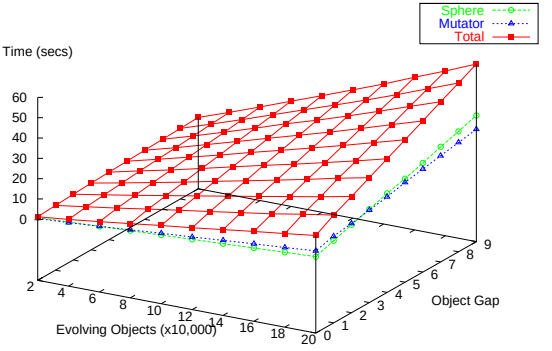

Fig. 3. Test 1 results

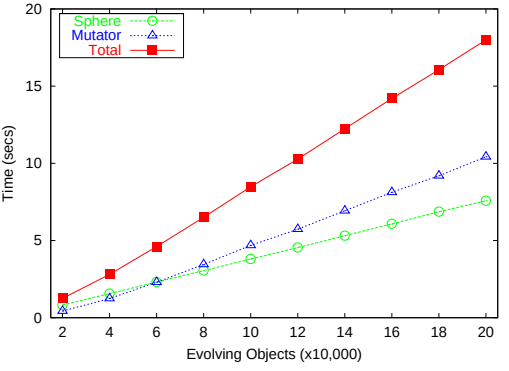

Fig. 4. Test 1 results – fixed $g = 0$

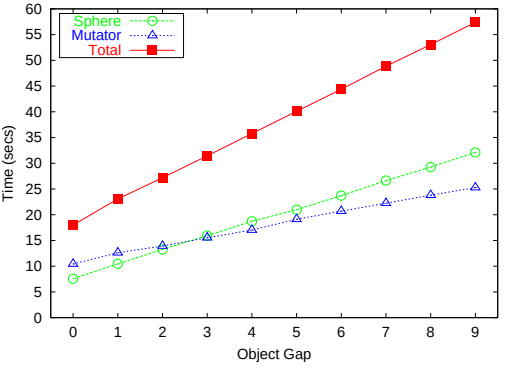

Fig. 5. Test 1 results – fixed $n = 200,000$

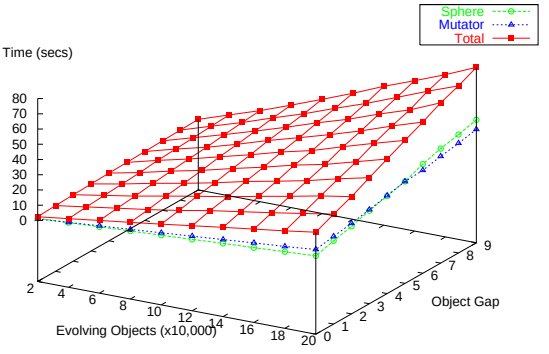

Fig. 6. Test 2 results

blocks, i.e. optimally clustered. On average, the slowdown for the store with $g = 9$ compared to the store with $g = 0$ was only about 5%.

In test 3 (figure 7) we observe the same linear behaviour of Sphere however the total time demonstrates a strange "quirk" in the part of the graph, where the number of objects is the greatest and they are packed densely. The cross-section of this graph at the constant value of $n = 200,000$ is presented in figure 8. The behaviour is clearly caused by some pathology in the upper software layer, i.e. JVM and Object Cache. At present we are investigating this problem.

To verify the linearity of our system's behaviour for much larger number of objects, we have performed the same evolution as in test 1, but with fixed $g = 5$ and with the number of objects varying between 100,000 and 2,000,000. The results are presented in figure 9. At the highest point in the graph, the store contains approximately 12,000,000 objects of which 2,000,000 interleaved objects are evolved.

In all of our tests we measured the amount of log traffic generated as part of evolution. Building limbo evolved objects generates no additional log traffic, as this step is performed as part of disk garbage collection, which itself has been optimised for very low log traffic (see [23,37]). Evolution only requires the generation of a log record for every evolving object at commit time i.e. when swapping the state of limbo objects to make them live (step ❺). Each log record is of a fixed size of 64 bytes (of which 40 bytes are system overhead), regardless of object size. In real terms 200,000 evolving objects generates approximately 16MB of log traffic[12].

[12] We anticipate that a further reduction in log traffic is possible by optimising the log records associated with swapping the limbo states. We believe we can cache the information, generating log records which represent a vector of swapped states, rather than the current one-per-object.

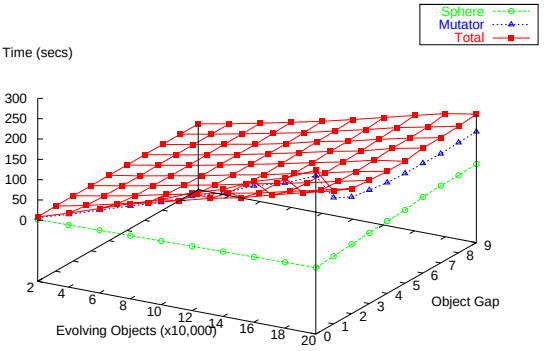

Fig. 7. Test 3 results

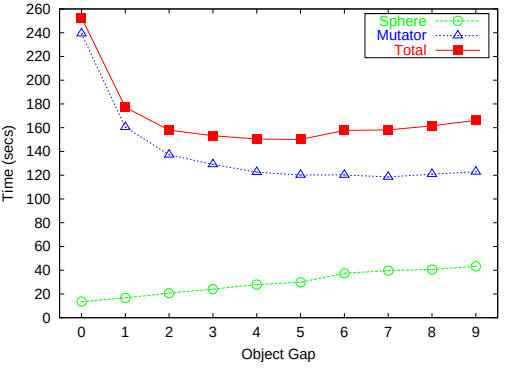

Fig. 8. Test 3 graph cross-section at $n = 200000$

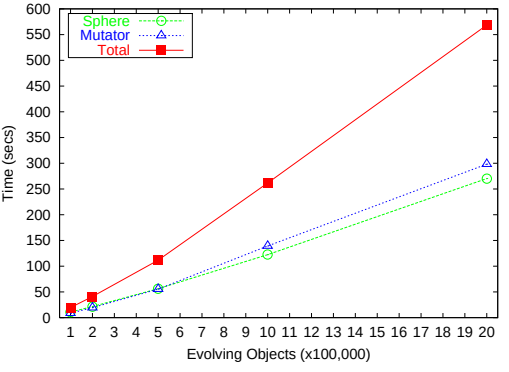

Fig. 9. Test 1 with large number of objects, fixed $g = 5$

We observed the time per evolving object. For each test we have calculated average total and average Sphere time per object (in test 3 not taking into account the pathologically behaving part of the graph). The results are summarised in the following table:

Test	Test 1	Test 2	Test 3
Object size (words)	2 – 3	7 – 8	7 – 7
Average total time per object (ms)	0.174	0.182	0.624
Average Sphere time per object (ms)	0.102	0.130	0.143

Comparing the time for test 1 and test 2, we observe relatively small change of time (30% for Sphere and almost 0% for total time), whereas the object size has grown about three times. We can conclude that at least for small objects and simple conversions the number of evolving objects matters much more than their size. Consequently, effects such as significant difference in conversion time for same size stores are possible. However, this might be different for larger objects. We plan to measure this in the near future.

6.3 A Real-Life Application

To validate these synthetic tests with a real-life application, we performed several experiments with GAP (Geographical Application of Persistence), an application which is being developed at the University of Glasgow by a succession of student projects. User map edits and and cartographic generalisation were added this year [24]. GAP is a relatively large system consisting of more than 100 classes, of which about 25 are persistent. The size of the main persistent store containing the map of the UK as poly-lines composed into geographic features, is nearly 700MB. So far the application's facilities focus mainly on displaying maps at various scales (with user-controlled presentation) and finding requested objects. In our evolution experiments we were changing persistent instances of "geographical line" class, subclasses of which represent such geographic features as roads and rivers. The form of vector data storage for such a feature can be in two forms shown on figure 10.

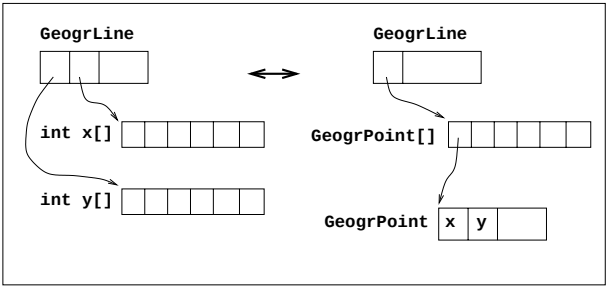

Fig. 10. Representations of Geographical Lines

During GAP development, both of these representations were tried, so we decided that conversion between them is a good example of a "real-world" schema change. We converted successfully about 900,000 line objects (complete UK data store), which took about 30 minutes. We also performed several experiments with smaller numbers of objects and observed practically linear time growth.

7 Related Work

To our surprise, we have managed to find very few works that deal with scalability and recoverability of evolution and its performance. There is one work where the O_2 system is benchmarked, which was published in 1994 [19]. Its extended variant was then included into [42]. In that work the authors concentrate on measuring and comparing the performance of immediate and deferred updates to variable size object databases. Since in PJama$_1$ we have currently implemented only immediate (eager) conversion facilities, this work is of no direct relevance to us. It is also not clear, whether the requirements of scalability and safety were considered in O_2. Furthermore, it is hard to compare the performance results, since in this work the authors didn't specify the changes they were making, and the hardware they used would be considered almost obsolete these days.

The RD45 project at CERN has focussed on the problems of providing persistent storage for vast quantities of data generated in physical experiments since 1995. At present the main system selected for use in this project is Objectivity/DB. Its evolution facilities are quite sophisticated. For example, it supports three kinds of object conversion: eager (immediate in their terminology), lazy (deferred) and what is called on-demand, which is eager conversion applied at the convenient time to the selected parts of the database. It also supports both default conversion and programmer-defined functions. A CERN internal report [18] contains some performance measurements for evolutionary object conversion with this system. Unfortunately, they mainly cover lazy conversion performance. And again, the hardware configuration used in the experiments (90MHz Pentium PC with 64MB memory and 1GB hard disk), does not allow us to compare absolute performance values. They also showed that the time was linear in the number of objects evolving, and not particularly sensitive to object size.

As for the scalability and recoverability requirements, it looks as if Objectivity/DB satisfies the latter but does not satisfy the former, at least when eager conversion is used. According to the report, during evolution this system gradually loads all the evolving objects into RAM and keeps them there until the end of the transaction.

8 Conclusions

In this paper we have identified the need for scheme evolution in persistent object systems and discriminated between two categories of evolution. We focus on

providing *development* evolution using an eager algorithm that exploits the potential for incremental algorithms which we designed into Sphere. This algorithm makes only moderate demands on the logging system, yet provides tolerance to crashes and a separation of co-existing before and after states. It utilises algorithms already developed for disk garbage collection and allows application developers to supply methods written in Java to migrate information from the old instances to their new versions. It is incremental, proceeding a partition at a time, and only needs two additional partitions, plus space for completely new data structures made reachable by the developers' conversion code. Yet the transition from the old to new state is atomic. Although illustrated for classes defined in Java, it is general purpose and could be used for any other collection of persistent classes and their instances.

The steps in the algorithm are:

❶ Identify the changes as the difference between old and new class definitions.
❷ Verify that the changes are complete, mutually consistent and will leave the store in a consistent state.
❸ Prepare the store by carrying out the marking phase of a cycle-collecting disk garbage collector.
❹ Scan the store, one partition at a time, calling conversion code for reformatted instances. Create new versions and their newly-reachable data structures in a new partition.
❺ Merge new and old versions of instances in a partition, using a modified garbage collector. Repeat from ❹ until all partitions that may contain instances have been processed.
❻ Fix up pointers in new objects to new versions of evolving objects.
❼ Atomically flip from exposing the old versions to exposing the new versions.

We have developed this algorithm to be scalable after experience with versions that we have reported previously [15,17], which required the complete evolved state to be resident in main memory. Proceeding one partition at a time limits the total space requirement, but still delivers linear performance and an atomic switch from the pre-evolution to post-evolution state. Our initial measurements, reported in this paper, indicate that we have met the target of linear performance with modest space and log requirements.

Further analysis, particularly with real work-loads, will be needed to confirm this. We have uncovered one anomaly which occurs when conversion methods try to revisit new objects that were created by earlier conversion method calls in a different partition. This requires a solution.

We are aware of several factors that will accelerate these algorithms, particularly the use of larger transfer units [37] and compression of the few log records that we write.

A feature of the evolution algorithms that we are developing is that they carry out extensive checks before proceeding as a "best effort" at catching developers' mistakes and inconsistencies, rather than letting them become persistent structural defects in a persistent object store. This line of development is also closely integrated with our work on making evolution more convenient by integrating

it with the build and compilation processes. Whilst we have mentioned it here, that work is still under development, and we expect to report it in a future paper.

We have noted that established software requires incremental background evolution, *deployed evolution* rather than the eager interruption of normal operations inherent in the work reported in this paper. We plan to investigate lazy and class versioning algorithms that meet such deployed evolution in our future work.

References

1. M.M. Astrahan, M.W. Blasgen, D.D. Chamberlin, K.P. Eswaran, J.N. Gray, P.P. Griffiths, W.F. King, R.A. Lorie, P.R. McJones, J.W. Mehl, G.R. Putzolu, I.L. Traiger, B.W. Wade, and V. Watson. System R: A relational approach to database management. *ACM Transactions on Database Systems*, 1(2):97–137, June 1976.
2. M.P. Atkinson, V. Benzaken, and D. Maier, editors. *Persistent Object Systems (Proc. of the 6th Int. W'shop on Persistent Object Systems)*, Workshops in Computing, Tarascon, Provence, France, September 1994. Springer-Verlag.
3. M.P. Atkinson, L. Daynès, M.J. Jordan, T. Printezis, and S. Spence. An Orthogonally Persistent Java™. *ACM SIGMOD Record*, 25(4), December 1996.
4. M.P. Atkinson and M.J. Jordan. Issues raised by three years of developing PJama. In C. Beeri and O.P. Buneman, editors, *Database Theory — ICDT'99*, number 1540 in Lecture Notes in Computer Science, pages 1–30. Springer-Verlag, 1999.
5. M.P. Atkinson and M.J. Jordan. A Review of the Rationale and Architectures of PJama: a Durable, Flexible, Evolvable and Scalable Orthogonally Persistent Programming Platform. Technical Report TR-2000-90, Sun Microsystems Laboratories Inc, 901, San Antonio Road, Palo Alto, CA 94303, USA, 2000.
6. M.P. Atkinson, M.J. Jordan, L. Daynès, and S. Spence. Design issues for persistent Java: A type-safe, object-oriented, orthogonally persistent system. In Connor and Nettles [14], pages 33–47.
7. M.P. Atkinson and R. Morrison. Orthogonal Persistent Object Systems. *VLDB Journal*, 4(3):309–401, 1995.
8. M.P. Atkinson, M.E. Orlowska, P. Valduriez, S. Zdonik, and M. Brodie, editors. *Proc. of the 25th Int. Conf. on Very Large Data Bases*. Morgan Kaufmann, Edinburgh, Scotland, UK, September 1999.
9. M.P. Atkinson and R. Welland, editors. *Fully Integrated Data Environments*. Springer-Verlag, 1999.
10. F. Cattaneo, A. Coen-Porsini, L. Lavazza, and R. Zicari. Overview and Progress Report of the ESSE Project: Supporting Object-Oriented Database Schema Analysis and Evolution. In B. Magnusson, B. Meyer, and J-F. Perrot, editors, *Proc. 10th Intl. Conf. on Technology of Object-Oriented Languages and Systems (TOOLS 10)*, pages 63–74. Prentice Hall, 1993.
11. M.F. Challis. The JACKDAW database package. In *Proc. of the SEAS Spring Technical Meeting (St Andrews, Scotland)*, 1974.
12. M.F. Challis. Database consistency and integrity in a multi-user environment. In B. Shneiderman, editor, *Databases: Improving Usability and Responsiveness*, pages 245–270. Academic Press, 1978.

13. D.D. Chamberlin, M.M. Astrahan, M.W. Blasgen, J.N. Gray, W.F. King, B G. Lindsay, R. Lorie, J.W. Mehl, T.G. Price, F. Putzolo, P.G. Selinger, M. Schkolnick, D.R. Slutz, I.L. Traiger, B.W. Wade, and R.A. Yost. A history and evaluation of system R. *Communications of the ACM*, 24(10):632, October 1981. Reprinted in M. Stonebraker, Readings in Database Systems, Morgan Kaufmann, San Mateo, CA, 1988.
14. R. Connor and S. Nettles, editors. *Persistent Object Systems: Principles and Practice*. Morgan Kaufmann, 1996.
15. M. Dmitriev. The First Experience of Class Evolution Support in PJama. In Morrison et al. [34], pages 279–296.
16. M. Dmitriev. Class and Data Evolution Support in the PJama Persistent Platform. Technical Report TR-2000-57, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland, 2000.
17. M. Dmitriev and M.P. Atkinson. Evolutionary Data conversion in the PJama Persistent Language. In *Proc. of the 1st ECOOP W'shop on Object-Oriented Databases*, 1999.
18. European Organization for Nuclear Research (CERN). Using an Object Database and Mass Storage System for Physics Analysis. http://wwwinfo.cern.ch/asd/rd45/reports/m3_96/milestone_3.htm [May 9, 2000].
19. F. Ferrandina, T. Meyer, and R. Zicari. Schema Evolution in Object Databases: Measuring the Performance of Immediate and Deferred Updates. In *Proc. of the 20th Int. Conf. on Very Large Data Bases, Santiago, Chile*, 1994.
20. GemStone Systems Inc. The GemStone/J iCommerce Platform. http://www.gemstone.com/products/j/main.html [May 9, 2000].
21. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, December 1996.
22. C.G. Hamilton. Recovery Management for Sphere: Recovering a Persistent Object Store. Technical Report TR-1999-51, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland, December 1999.
23. C.G. Hamilton, M.P. Atkinson, and M. Dmitriev. Providing Evolution Support for PJama_1 within Sphere. Technical Report TR-1999-50, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland, December 1999.
24. R.P. Japp. Adding Support for Cartographic Generalisation to a Persistent GIS. BSc Dissertation, University of Glasgow, Department of Computing Science, 2000.
25. M.J. Jordan and M.P. Atkinson. Orthogonal Persistence for Java — A Mid-term Report. In Morrison et al. [34], pages 335–352.
26. M.J. Jordan and M.P. Atkinson. Orthogonal Persistence for the Java Platform — Specification. Technical report, Sun Microsystems Laboratories Inc, 901, San Antonio Road, Palo Alto, CA 94303, USA, 2000.
27. B. Lewis and B. Mathiske. Efficient Barriers for Persistent Object Caching in a High-Performance Java Virtual Machine. In *Proc. of the OOPSLA'99 w'shop "Simplicity, Performance and Portability in Virtual Machine Design"*, 1999.
28. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
29. Fujitsu Ltd. 3.5-inch Magnetic Disk Drives MAB3045/MAB3091. http://www.fujitsu.co.jp/hypertext/hdd/drive/overseas/mab30xx/mab30xx.html [January 5, 2000].
30. B. Mathiske, B. Lewis, and N. Gafter. Architecture of the PEVM: A High-Performance Orthogonally Persistent Java Virtual Machine, March 2000. These proceedings.

31. C. Mohan. Repeating History beyond ARIES. In Atkinson et al. [8], pages 1–17.
32. C. Mohan, D. Haderle, B. Lindsay, H. Pirashesh, and P. Schwarz. ARIES : A Transaction Recovery Method supporting Fine-granularity Locking and Partial Rollbacks using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.
33. C. Mohan, B. Lindsay, and R. Obermarck. Transaction Management in the R* Distributed Database Management System. *ACM Transactions on Database Systems*, 11(4):378–396, December 1986.
34. R. Morrison, M.J. Jordan, and M.P. Atkinson, editors. *Advances in Persistent Object Systems — Proc. of the 8th Int. W'shop on Persistent Object Systems (POS8) and the 3rd Int. W'shop on Persistence and Java (PJW3).* Morgan Kaufmann, August 1998.
35. D.S. Munro, R.C.H. Connor, R. Morrison, S. Scheuerl, and D. Stemple. Concurrent shadow paging in the flask architecture. In Atkinson et al. [2], pages 16–42.
36. E. Odberg. Category classes: Flexible classification and evolution in object-oriented databases. *Lecture Notes in Computer Science*, 811:406–419, 1994.
37. T. Printezis. *Management of Long-Running, High-Performance Persistent Object Stores.* PhD thesis, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland, 2000.
38. D.I.K. Sjøberg. *Thesaurus-Based Methodologies and Tools for Maintaining Persistent Application Systems.* PhD thesis, Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland, 1993.
39. Sun Microsystems Inc. Workgroup Servers, Sun Enterprise™ 450. http://www.sun.com/servers/workgroup/450/ [January 5, 2000].
40. P. Wegner and S.B. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In S. Gjessing and K. Nygaard, editors, *ECOOP '88, European Conf. on Object-Oriented Programming, Oslo, Norway*, volume 322 of *LNCS*, pages 55–77. Springer-Verlag, August 1988.
41. D. White and A. Garthwaite. The GC interface in the EVM. Technical Report TR-98-67, Sun Microsystems Laboratories Inc, 901, San Antonio Road, Palo Alto, CA 94303, USA, 1998.
42. C. Zaniolo, S. Ceri, C. Faloutsos, R.T. Snodgrass, V.S. Subrahmanian, and R.Zicari. *Advanced Database Systems.* Morgan Kaufmann, 1997.
43. S.B. Zdonik. Version management in an object-oriented database. In *Proc. of the IFIP Int. W'shop on Advanced Programming Environments*, pages 405–422, Trondheim, Norway, June 1987.
44. R. Zicari. A Framework for Schema Updates in an Object-Oriented Database System. In F. Bancilhon, C. Delobel, and P. Kanellakis, editors, *Building an Object-Oriented Database System: The story of O_2*. Morgan Kaufmann, 1992.