# Performance improvement for persistent systems by AOP

Yasuhiro Aoki    Sigeru Chiba

Tokyo Institute of Technology
aoki, chiba@csg.is.titech.ac.jp

## Abstract

Efficient data retrieval from databases is a significant issue of the design of persistent systems. We propose an aspect oriented persistent system named AspectualStore. AspectualStore opens up its data retrieval mechanism so that developers can customize it in an aspect for performance optimization. The aspect controls the generation of SQL queries and minimizes the number of round-trips. It can also apply the customization only when the dynamic behavior of data accesses matches given patterns. This customization based on dynamic contexts is useful in practice and it needs aspect-oriented programming. Traditional object-oriented programming is not appropriate although it can manage the customization based on static structures, such as customization for a particular class of data.

***Categories and Subject Descriptors*** D.2.11 [*Software Architectures*]: Domain-specific architectures

***General Terms*** Languages, Design

***Keywords*** Aspects, persistence, performance, databse

## 1. Introduction

In Java, application software accessing a relational database is often developed by using a persistent object system. Traditional persistent systems such as EJB2 [14] transparently resolve impedance mismatch between objects and relational database entities. Without such a persistent system, developers must do cumbersome work for resolving impedance mismatch. For example, developers have to directly describe SQL strings, manually retrieve data from database by JDBC [15], and program the map from a database record to an object.

A persistent system transparently resolves impedance mismatch. With a persistent system, developers have to do cumbersome work except retrieving a root persistent object. When a program accesses an object referred to by the root object, a persistent system automatically generates SQL strings and retrieves a record from a database and converts it into an object. For example, Figure 1 is a part of application program of a bibliography search system. Since it is written with a persistent system, no description for accessing a database is included in this code. The persistent system implicitly retrieves necessary data from the database and creates a persistent object when the program calls getAuthor() or getTitle() on a persistent object Paper.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*Workshop SPLAT '07* March 12-13, 2007 Vancouver, British Columbia, Canada.
Copyright © 2007 ACM 1-59593-656-1/07/03...$5.00

However, the traversal of a persistent object graph from its root object is often inefficient. A naive persistent system will send a query to a database whenever one reference between objects is resolved. To avoid this problem, we propose an aspect-oriented persistent system called *AspectualStore*. This system allows developers to customize it by an aspect so that the number of round-trips to a database will be minimized. It enables various prefetching, depending on dynamic contexts such as the path from the root object to the target object. To do that, AspectualStore opens up its internal data retrieval mechanism and thereby developers can extend AspectualStore to make various dynamic contexts accessible from an aspect. Although traditional persistent systems allow customization depending on static contexts, such as the class of accessed objects, AspectualStore enables more flexible customization.

```
01   void showPaperList(List<Paper> papers) {
02       Iterator it = papers.iterator();
03       while(it.hasNext()) {
04           Paper p = (Paper)it.next();
05           show(p);
06           ...
07       }
08   }

10   void show(Paper p) {
11       Author a = p.getAuthor();
12       String title = p.getTitle();
13       String name = a.getName();
14       /* show p.title and a.name */}
```

**Figure 1.** Java code using a persistent system

## 2. Performance Improvement

Although persistent systems provide transparent mapping between objects and relational databases, their execution performance is not satisfactory in practice without appropriate hints or customization given by developers. On the other hand, the hints that can be given to traditional persistent systems are limited and hence it has been difficult to achieve sufficient performance. For example, traditional systems allow customizing the prefetching scheme for a particular class of persistent objects. However, they do not allow customization depending on dynamic contexts of data accesses.

For example, see the showPaperList(List) method in Figure 1 again. This method shows the author's names and the titles of a collection of papers. If the show(Paper) method called in the showPaperList(List) method causes a database round-trip for obtaining the author's name from the author table (because the paper table includes only the author number), the while statement in showPaperList(List) would cause a large number of round-trips. This is very inefficient. For example, when a Paper object is obtained

from the paper table, the Author object referred to by the Paper object should be simultaneously prefetched from the author table. This prefetching can be efficiently executed by the join between the paper table and the author table. However, prefetching an Author object whenever a Paper object is retrieved will consume a huge memory space and make a serious performance impact. A large number of unnecessary Author objects may be prefetched.

To really improve performance, developers should be able to specify customization, such as a prefetching scheme, depending on dynamic contexts of application programs. For example, an Author object should be prefetched only when a list of Paper objects is retrieved as the argument to the showPaperList(List) method. However, describing such customization in an object-oriented language like Java would crosscut a program and decreases the readability of the program.

## 3. Using AOP for improving performance

We propose an aspect-oriented persistent system named *AspectualStore*. To improve the performance of applications of AspectualStore, the developers can customize how AspectualStore retrieves persistent objects according to dynamic contexts of the applications. For example, persistent objects can be prefetched when they are reached from the persistent root object along a particular path of object references. This customization is written in an aspect-oriented language and thus it never scatters over the application program. Since the performance concern is separated into an aspect, the developers do not have to edit the original program when they describe performance hints to fix the performance bottlenecks found by, for example, performance profiling. To support customization depending on dynamic contexts, AspectualStore opens up its internal structure. The developers can extend the internal structure by an aspect so that extra dynamic contexts necessary for performance improvement will be recorded and visible at runtime. Such extra contexts include which references have been navigated from a persistent object, which method is being executed, and which reference navigation causes a database round-trip.

### 3.1 Optimize the behavior of showPaperList(List)

Now we show how an aspect is used for optimizing the showPaperList(List) method in Figure 1. The performance problem of showPaperList() was that a large number of database round-trip may happen. To avoid this, we can customize the database accesses during the execution of showPaperList(List). Figure 2 is an aspect written in GluonJ [13, 10] for optimizing the behavior of showPaperList(List). This aspect means that the code block passed to @Before as a String argument is executed just before showPaperList method is called. In the code block, which is a before advice, $1 represents the first argument to showPaperList. The type of $1 is List<Paper> (see the line 01 in Figure 1). This code block describes that AspectualStore must prefetch the title and name fields of Author objects of each Paper object included in the argument to showPaperList.

It is also possible to write an aspect that automatically determines which fields of Author should be prefetched when the showPaperList method is called. Figure 3 is such an aspect. Since the showPaperList method runs the while loop for printing the data of each Paper object, the aspect records all the fields accessed from a Paper object during the first iteration. Then it prefetches those fields for the rest of the Paper objects by a single database query before the second iteration starts. For example, because getAuthor() is called in the show method in Figure 1, the reference from the Paper object to the Author object is recorded. Of course, we here assume that the navigated references during the first iteration will be also navigated during the rest of the iterations.

```
01  class PrefetchDefine  {
02    @Before("{ Persistable p = $1;
03               Loader l = $1.getBody().getLoader();
04               l.addFetch(\"author.title\");
05               l.addFetch(\"author.name\");
06               l.load(); }")
07    Pointcut p1 =
08        Pcd.call("showPaperList(..)");
09  }
```

**Figure 2.** An aspect for optimizing showPaperList(List)

```
01  @Glue class PrefetchDefine {
02    /* Make an object for recording the names of
03       the fields causing database accesses. /*
04    @Refine
05    static class ContextDefine extends Context{
06      Set loadedFields = new HashSet();
07      int depth = 0;
08      ...}

10    /* Record the names of the fields that caused
11   database accesses in showPaperList(). */
12    @Before("{Context c = $2;
13              while(c.depth > 0){...}
14              c.addField($1);}")
15    Pointcut pc =
16      Pcd.call("...PersistentEntity#load(..)").
17      and.cflow("showPaperList(..)");

20    /* Do prefetching. */
21    @Refine
22    static class IteratorDefine
23                extends PersistentListIterator {
24    public PersistentEntity next(Context cxt) {
25      if(ctx.hasLoadedFields()) {
26        Loader loader = getLoader(cxt);
27        Collection fs = cxt.getLoadedFiels();
28        loader.addFetches(fs);
29        // make an SQL string and execute prefetch
30        loader.load();}
31        return super.next(cxt); }}}}
```

**Figure 3.** Further optimization of showPaperList(List)

To perform such prefetching based on profiling, we must extend the implementation of AspectualStore itself because the original AspectualStore does not perform such profiling. The aspect in Figure 3 first adds extra fields to the Context class, which is an internal class of the implementation of AspectualStore (line 2-8). The added fields are used for recording the field accesses during the first iteration. The second part of the aspect is a before advice, which records a field access whenever the load method in PersistentEntity is called during the execution of showPaperList (line 10-17). The load method is an internal method of AspectualStore and it is called to access a database for obtaining a field value of a persistent object. The first argument ($1) to the load method is the accessed field and the second argument ($2) is the current Context object. The last part of the aspect is to modify the implementation of the next method in the PersistentListIterator class. This method is called at the line 4 in Figure 1. The modified next method performs prefetching when it is called for the second iteration. It calls the addFetches method to prefetch all the data that will be accessed during the rest of the iterations (line 28).

CASE1 : Proceeding -> Paper

```
SELECT t0.title, t1.name from paper
  inner left join author t1 ...
  WHERE t0.prc_id = 5
```

proceeding table

| | | | |
|---|---|---|---|
| | | | |
| 5 | AOSD | ... | |
| ⋮ | | | |

id

journal table

| | | | |
|---|---|---|---|
| ⋮ | | | |
| 98 | GPCE | ... | |
| | | | |

id

paper table

| id | prc_id | | journal_id |
|---|---|---|---|
| 1 | 5 | ... | |
| 2 | | | 98 |
| 3 | 5 | | |
| 4 | 5 | | |
| 5 | | | 98 |
| 6 | | | 98 |

CASE2 : Journal -> Paper

```
SELECT t0.title, t1.name from paper
  inner left join author t1 ...
  WHERE t0.journal_id = 98
```

**Figure 4.** difference between SQL strings

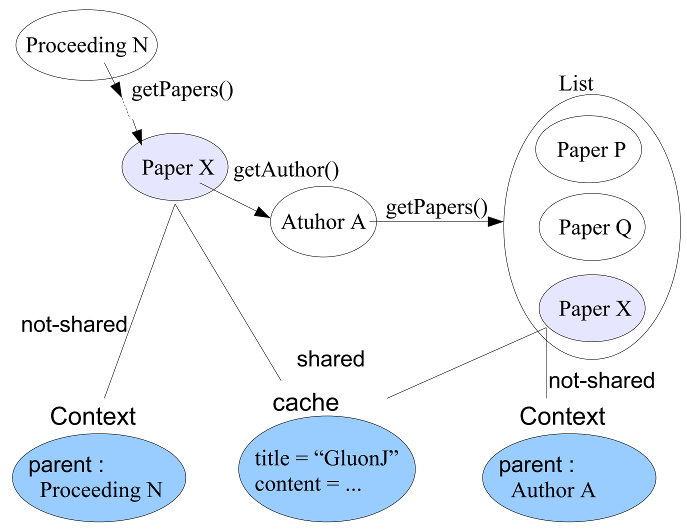


**Figure 5.** Context objects

### 3.2 Features of AspectualStore

AspectualStore provides a useful mechanism for developers to retrieve persistent objects without directly writing an SQL string. It also provides data structures for storing custom contexts of application programs and the mechanism of load-time weaving for efficient software development.

First, AspectualStore allows developers to optimize a database query on demand without directly specifying a SQL string. Most persistent systems statically generates SQL strings when the systems starts. The systems selects one of the generated SQL strings whenever they access a database. On the other hand, AspectualStore allows dynamically changing an SQL string. Developers can switch multiple SQL strings to access the same database table. AspectualStore can generate an SQL string on demand.

Developers do not have to directly describe an SQL string when they give a custom SQL string to AspectualStore. As we saw at the line 4 and 5 in Figure 2, the developers have only to call the `addFetch` method. Because AspectualStore maintains the current navigation path, it can generate an appropriate SQL string. Suppose that there are three database tables: a proceedings table, a journals table, and a papers table (Figure 4). When we first obtain a `Proceedings` object from the proceedings table and then obtain all the papers included in that proceedings, we must select the papers that have the proceedings identifier (`proc_id`) of the `Proceedings` object we first obtained. On the other hand, when we want to obtain the papers included in a `Journal` object obtained from the journals table, we must select the papers that have the journal identifier (`journal_id`) of that journal. Because the proceedings identifier and the journal identifier are different columns, we must use a different SQL string. These annoying implementation details are maintained by AspectualStore. Developers do not have to care about them.

Second, AspectualStore provides a `Context` object for every persistent object and its collection. The `Context` object is used for maintaining dynamic application contexts, which are necessary for various query optimization. The dynamic contexts include which references have been navigated from a persistent object, which method is being executed, which reference navigation causes database a round-trip, and so on. For example, the optimization for `showPaperList(List)` shown in Figure 3 used the dynamic contexts for selecting the fields that should be prefetched. It extended the `Context` object so that it would also record which fields were accessed during the first iteration of the while statement. Existing persistent systems and aspect-oriented languages do not provide direct accesses to such dynamic contexts related to data accesses [4]. However, providing a `Context` object is a significant feature of AspectualStore because AspectualStore is an AOP system for persistent objects.

AspectualStore creates a distinct `Context` object for each persistent object and its collection. Note that most persistent systems, including AspectualStore, may reuse persistent objects among sessions. Although the contents of a persistent object are shared among sessions, a `Context` object is not (Figure 5). Hence, developers can safely store session-specific data into a `Context` object. AspectualStore itself stores the navigation path of persistent objects into a `Context` object and uses it for generating an SQL string on demand.

Finally, AspectualStore provides the mechanism of load-time weaving for making software development efficient. All aspects are woven into AspectualStore when AspectualStore is loaded at the start-up time. Associating a `Context` class to a persistent class is also executed at load time. The weaving is done at the Java bytecode level. Our implementation of AspectualStore uses Javassist [12, 11] for the bytecode-level weaving. In case that AspectualStore is running on top of the Tomcat servlet server [2], a custom class loader provided by AspectualStore must be used for load-time weaving.

## 4. Discussion

### 4.1 Preserving the original behavior

Since AspectualStore opens up its internal structure, developers can freely customize the implementation of AspectualStore. This means that, in principle, developers can alter the semantics of the behavior of AspectualStore. However, the customization by an aspect should preserve the original semantics of AspectualStore and its application program.

For example, see Figure 6. This program retrieves some proceedings and prints only the AOP papers among the papers in that proceedings. Thus, the best optimization is to retrieve only AOP papers when it makes a `List<Paper>` object by the `getPapers` method (line 2). Retrieving non-AOP papers is redundant because we know

```
01  Proceeding prc = ...;
02  List<Paper> papers = prc.getPapers();
03  Iterator<Paper> it = papers.iterator();
04  while(it.next()) {
05    Paper p = it.next();
06    if (p.getGenre().equals("AOP") {
07      // show Paper details
08      ...
09    }
10  }
```

**Figure 6.** Print only AOP papers

that only the AOP papers are accessed in the while statement (line 4-10). However, an aspect implementing this best optimization is dangerous and fragile. We may modify the while statement in future so that other kinds of papers will be accessed.

To avoid this problem, if an aspect specifies that only AOP papers should be retrieved, AspectualStore issues two SQL queries. The first query retrieves only the primary keys of all the Paper objects. Then it issues the second SQL query and retrieves all the properties of only the Paper objects whose genre is AOP. Note that a Paper object is created for each primary key although the field values of non-AOP Paper objects are empty and retrieved later on demand. The size of papers (line 2) is equal to the number of the papers included in the proceedings. It is not equal to the number of the AOP papers.

Although issuing two SQL queries is less efficient than a manually optimized program using JDBC, the optimization by an AspectualStore aspect is safer. Because an aspect is often separately maintained, preserving the program semantics is important to avoid serious bugs in future evolution.

### 4.2 Related Work

There are many researches for improvement performance of applications on persistent systems. O/R mapping frame works such as Hibernate [6] and Cayenne [3] provide APIs to customize retrieval root object or set of root objects. At least in principle, efficient data retrieval is realized by adding all persistent objects which may be necessary in the application to the root object. For Example, Figure 7 is a java code using Hibernate for optimizing Figure 1. Developers can retrieve in a lump sum all objects which are used in application by writing a query string (line 2-6). But this way is not realistic. Because it is difficult for developers to determine exactly what related objects should be prefetched. Modularity and user interaction may interfere with these analyses. In contrast, developers using AspectualStore can instruct persistent objects or collections when they are reached from the persistent root object.

Developers using AspectualStore improve performance of applications by aspect description. But there are many researches to improve performance of applications automatically. AspectualStore has extra work to describe aspects for optimization. Instead of this, AspectualStore supports more flexible optimization than automatic optimization. Most of automatic optimization approach prefetches objects according to only persistent objects behavior. In contrast optimization by aspect can consider about other factors such as method calls. In addition, a benefit of optimization by aspect is adding optimization to applications as needed. In the Automatic optimization, if developers are not satisfied the result of performance tests, there is no way to optimization. Optimization by aspect can realize more detail optimization.

context-controlled prefetch [8] prefetches data depending on access patterns of persistent objects. For example, when a field of a

```
01  // HQL description
02  String query = "FROM Proceeding p
03      LEFT JOIN FETCH p.papers
04      LEFT JOIN FETCH p.papers.author
05      WHERE ...";
06  Query q = session.createQuery(query);
07  // retrieve root objects
08  Proceeding p = q.load();
    ...
10  List papers = p.getPapers();
11  // show papers
12  showPaperList(papers);
```

**Figure 7.** Java code using Hibernate

persistent object which is an element of a collection is retrieved, [8] prefetches fields of all elements includes in that collection. But [8] has a dread of depth-first-search. AUTOFETCH [1] automatically prefetches by traversal profiling. By profiling traversals on query results, AUTOFETCH determines which objects added persistent root objects. PrefetchGuide[16] dynamically prefetches by access logs on persistent objects. But these approaches don't consider about applications which include user interactions.

There are many researches for prefetch. ObjectStore [5, 9] prefetches objects by the page. When an object is retrieved in a page, all objects in the same page is retrieved. This approach is useful only when many objects in same page is used. Curewitz et al. proposed using data compression algorithms to prefetch data [7]. These approaches are useful only when the identical objects are used repeatedly.

## 5. Conclusion

We proposed an aspect-oriented persistent system named *AspectualStore*. It is important for improving performance of applications on persistent systems to retrieve data from databases. To improve the performance of applications on AspectualStore, developers could customize how to retrieve persistent objects, depending on dynamic contexts of the applications. Since the customization of concern was separated in an aspect, developers could give performance hints separately without editing the original program when they found performance bottlenecks by performance profiling. To support customization depending on dynamic contexts, AspectualStore opened up its internal structure so that developers could extend it by an aspect and made necessary dynamic contexts available at runtime.

## References


[1] Ali Ibrahim, William R. Cook. Automatic prefetching by traversal profiling in object persistence architectures. In *European Conference on Object-Oriented Programming (ECOOP)*, Jun. 2006.

[2] Apache Software Foundation. Apache tomcat. http://tomcat.apache.org/.

[3] Apache Software Foundation. Cayenne. http://incubator.apache.org/cayenne/.

[4] Awais Rashid, Ruzanna Chichyan. Data-oriented aspect. In *Asian Workshop on Aspect-Oriented Software Development (AOAsia)*, Sept. 2006.

[5] Charles Lamb, Gordon Landis, Jack Orenstein, Dan Weinreb. The objectstore database system. *Communications of the ACM*, 34:50–63, 1991.

[6] JBoss, Inc. Hibernate. http://www.hibernate.org/.

[7] Kenneth M. Curewitz, P. Krishnan, Jeffrey Scott Vitter. Practical prefetching via data compression. In *Proceedings of International Conference on Management of Data*, pages 257–266, May. 1993.

[8] Philip A. Bernstein, Shankar Pal, David Shutt. Context-based prefetch for implementing objects on relations. In *Proceedings of the 25th International Conference on Very Large Data Bases*, 1999.

[9] Progress Software Corporation. Objectstore. `http://www.progress.com/realtime/products/objectstore/`.

[10] Shigeru Chiba. Gluonj. `http://www.csg.is.titech.ac.jp/projects/gluonj/`.

[11] Shigeru Chiba. Javassist home page. `http://www.csg.is.titech.ac.jp/~chiba/javassist/index.html`.

[12] Shigeru Chiba. Load-time structural reflection in java. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 313–336, Jun. 2000.

[13] Shigeru Chiba, Rei Ishikawa. Aspect-oriented programming beyond dependency injection (ecoop). In *European Conference on Object-Oriented Programming*, pages 121–143, July. 2005.

[14] Sun Microsystems, Inc. Ejb. `http://java.sun.com/products/ejb/`.

[15] Sun Microsystems, Inc. Jdbc. `http://java.sun.com/javase/technologies/database/`.

[16] Wook-Shin Han, Yang-Sae Moon, Kyu-Young Whang. Prefetchguide: capturing navigational access patterns for prefetching in client/server object-oriented/object-relational dbmss. *Information Sciences*, 152:47–61, 2003.