Object Prefetch Filter

A Pattern for Improving the Performance of Object Retrieval of Object-Relational Mapping Tools

Tim Wellhausen

kontakt@tim-wellhausen.de
http://www.tim-wellhausen.de

June 10, 2007

Abstract: Object-relational mapping tools provide a bridge between object-oriented programming languages and relational database systems. Although the concepts of object-relational mapping are well understood, the performance of object retrieval still is a crucial point in using the tools. Prefetching objects is a means to improve the performance, in particular to avoid the so-called *n+1 selects* problem. This problem arises when a list of objects is retrieved from a database and for each object another database call needs to be executed to retrieve referenced objects. This paper presents a pattern that, if applied to the implementation of a mapping tool, gives application developers the choice to explicitly define for every query which portion of a network of objects to prefetch.

Introduction

Over the last years, the usage of object-relational mapping tools has become wide-spread. These tools diminish the impedance mismatch between object-oriented programming and relational databases by providing and supporting an object-oriented API to store objects into relational databases (for a list of available tools, at least for the Java platform, see [C2Wiki]).

Many publications, including individual patterns and pattern languages ([Fowler2003], [Keller-1997], [Keller1998]), have documented key elements of object-relational mapping. The fundamental problems are now well-understood and solved by the available tools. However, the mismatch still exists. As good as the available tools are, they cannot completely hide the conceptual difference between objects and relational algebra.

One of the major remaining issues is performance, in particular the performance of object retrieval when a network of persistent objects is navigated. Most recent mapping tools support transparent loading of associated objects on demand. That means, a developer doesn't need to explicitly fetch associated objects. Instead, this is done transparently by the mapping tool when the graph of objects is navigated.

Such a comfortable solution to object retrieval comes at a price, however. Used naively, this approach may lead to severe performance problems because each individual object retrieval operation causes a round-trip to the database. This behavior is well-known and documented as the *n+1 selects* problem (see, for example, [Bauer+2004]).

To comprehend the *n+1 selects* problem, suppose that there is an `Order` object that contains a set of `OrderItem` objects, each of which references a `Product` object (see Fig. 1 on the next page). When the `Order` object is retrieved, at first, none of its related order items are fetched. If the `Order` class provides a `getOrderItems()` method, however, all order items are transparently loaded when the method is called. Suppose now that all order items need to be checked for the availability of the ordered products. Every time an order item's `Product` object is requested, the database is called to fetch the product data. For `n` order items, the database is therefore accessed `1` (order items) + `n` (products) times in total.

A solution to the *n+1 selects* problem is object prefetching. Applying this technique, the number of round-trips to the database can be cut down significantly by fetching associated objects in advance, i.e. before they are needed for the first time.

Applied to the example of iterating over order items, the performance of checking the products could be hugely improved: When the set of order items is loaded, all associated products are loaded as well. Mapping tools do this either by using an `outer join` expression or by executing a second `select` statement.

There has been some research lately on how to best support object prefetching. Some approaches suggest to profile and dynamically change the prefetch behavior of an application ([Han+2003], [Ibrahim+2006]). As long as these approaches have not yet manifested in mainstream tools, however, developers still need to explicitly cope with prefetching.

This paper documents a pattern that supports the explicit prefetching of objects. It is based on the assumption that application developers know at programming-time what objects they need in order to implement a requirement. The OBJECT PREFETCH FILTER pattern suggests to explicitly define which objects are needed and to let the mapping tool prefetch the required network of objects with minimal costs when executing a query.

The pattern is difficult to apply if it is not supported by a mapping tool itself. Therefore, the OBJECT PREFETCH FILTER pattern primarily addresses the authors of mapping tools. Depending on the design of a mapping tool, it may be possible to implement the pattern on top of an existing tool. The OBJECT PREFETCH FILTER pattern therefore also addresses application developers.

Running Example

To illustrate the *n+1 selects* problem and how the OBJECT PREFETCH FILTER pattern helps to solve this problem, a real-world running example is given. Consider an e-commerce application that takes orders for the products a company sells. The domain model for this application is shown in Figure 1. A costumer may place any number of orders. An order is made up of arbitrarily many order items, each of which refers to a product. Furthermore, every order has an invoice and a shipping address that both belong to the customer.

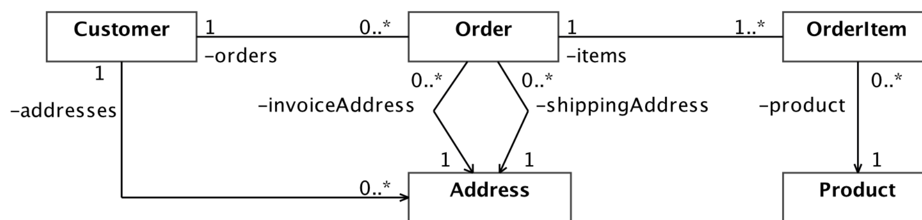

Fig. 1: Domain model of an e-commerce application

The e-commerce application needs to perform many tasks to keep the shop running. Among these tasks are:

- Present an overview list of all orders of a customer.
- Present a detailed list of all orders of a customer.
- Generate an invoice.

To generate an overview list of the customer's orders, the system has to retrieve both the customer object and all of its associated orders (see dashed line in Fig. 2). If the system has to generate a detailed list of the customer's orders, it additionally needs to fetch the order items of all orders and their related products (see additional dotted line in Fig. 2).

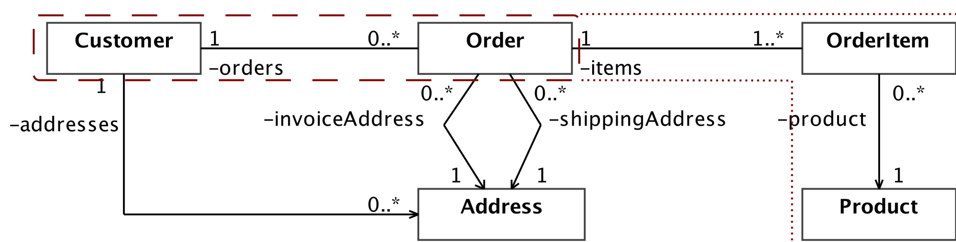

Fig. 2: Objects needed to generate a detailed list of a customer's orders

To generate an invoice for an order, the system needs to fetch the appropriate `Order` object and all associated objects that are shown in the diagram.

If the object-relational mapping tool in use does not support object prefetching, the performance of both the second and the third task would suffer because of the *n+1 selects* problem. If the mapping tool supports prefetching but does not provide a flexible means to define the objects to be prefetched on a per-query basis, it is not possible to optimize the object retrieval for all tasks: For some tasks, too few objects are prefetched, for other tasks, too many.

Context

Object-relational mapping tools support the development of object-oriented applications that store their persistent data in relational databases. Such a mapping tool is able to load and update persistent objects including all of their references and make the persistent data available at runtime as a network of interconnected objects.

The data access layer of an application integrates and uses such a mapping tool to load, change, and update persistent data. The application's domain model may contain explicit references between its entities, which may be 1:1, 1:n, and n:m relations. To implement the domain logic, the application needs to navigate along these references.

Problem

To retrieve persistent objects from a database, object-relational mapping tools provide some kind of query functionality. Using a query, an application gets objects and navigates among their object references to implement a use case or a service. If navigating the network of objects causes the mapping tool to fetch objects only at the time when they are requested, the application's performance is likely to drop because of the *n+1 selects* problem.

Therefore, to efficiently navigate among a network of persistent objects, the objects that are needed must be prefetched by the mapping tool, i.e. they must be loaded into memory before they are accessed. Without sophisticated heuristics, though, a mapping tool does not know by itself which objects an application needs to be prefetched.

How can an object-relation mapping tool support efficient prefetching?

Forces

To let the mapping tool prefetch objects, an application could explicitly add `outer join` expressions to a query so that the SQL statement that is executed on the database does not only load the data of the target object but also the data of its associated objects. In case of 1:1 associations, this approach works well. In case of 1:n and n:m associations, however, this approach leads to a duplication of the data retrieved from the database. In particular, if several 1:n or n:m associations need to be resolved, result sets grow huge because they contain cross products of data from the joined tables.

To avoid cross products, a network of objects should therefore be loaded not by executing a single SQL statement but by several individual statements. For this to achieve, an application could statically declare at a meta level dependencies between objects: Whenever an object of a specific type is loaded specific associated objects should also be loaded. The application may use this feature to both avoid cross products in the result sets and to reduce the number of queries.

However, as the example has shown, it is very difficult to optimize the prefetch behavior of a domain model statically because different use cases may have different prefetch needs. Sometimes, too few objects are prefetched, sometimes, too many.

As an alternative, the application could declare that whenever one object of a specific type is loaded, all other objects of the same type that appear in the current network of objects should be loaded as well. That means, a query at first only returns the query's target object(s). Whenever the application then gets an object that is part of a 1:n or n:m association, all other objects of that association end are also loaded.

While this approach indeed helps to avoid the *n+1 selects* problem, it only works inside the context of a session that is connected to the database. If the persistent domain objects need to be accessed outside this context, for example in a web presentation layer or in a rich client application, they need to be detached from the session. In that case, transparent loading of missing objects does not work any more. A web application's presentation layer or a rich client application rather need to receive a network of objects that is completely prefetched with respect to their specific requirements.

To fetch all objects that a web presentation layer or a rich client application needs, the application may explicitly traverse the respective portion of the object graph. By invoking the appropriate accessor methods one-by-one, the application makes the mapping tool fetch all needed objects. While this approach in fact ensures that all objects are loaded, the code that manually traverses the network of objects is tedious to write and error-prone to modify.

Solution

Extend the query mechanism of an object-relational mapping tool so that an application may explicitly specify on a per-query basis which objects need to be prefetched. Such an OBJECT PREFETCH FILTER gives the application the choice to dynamically define the portion of a network of objects that the mapping tool should prefetch when the query is executed.

Implement an OBJECT PREFETCH FILTER as a set of object traversal paths where each path is a concatenation of consecutive object associations, starting from the target object of the query. When the query is executed, let the mapping tool not only load the query's target object but also all objects along each of the given traversal paths. As effect, the application should be able to navigate from the target object along each of the traversal paths without causing any further database calls.

Implementation

The implementation of the pattern consists of two parts: a query mechanism that allows the definition of networks of objects to prefetch and an algorithm that actually prefetches objects. Because an implementation of the pattern heavily depends on the facilities of an actual mapping tool, both parts are illustrated by resolving the example.

Suppose, the example e-commerce application needs to present a detailed list of all orders of a customer. The application thus needs to load the `Customer` object, its `Order` objects, their `OrderItem` objects, and each order item's `Product` reference. To get and prefetch these objects, the application uses a dedicated query API, provided by the mapping tool.

A prefetch filter must be defined to include all of these objects. The appropriate pseudo code might look as follows:

```
Query query = session.createQuery("Customer");
query.setCriteria(...); // set criteria to load the customer by its id
query.setPrefetchFilter(
  new PrefetchFilter("orders",
                     "orders.items",
                     "orders.items.product"));
Order oder = (Order) query.retrieve();
```

Whether or not all nodes of the object network to prefetch have to be explicitly named is a matter of discussion. Instead of explicitly defining all traversal paths, only defining the path "`orders.items.product`" would already give the same information in this example.

The mapping tool needs to analyze the prefetch filter to create an object prefetch strategy. That means, the mapping tool has to decide how many SQL queries to execute to prefetch all objects as requested. For each association, the mapping tool has two options: include the associated objects using an `outer join` statement or execute a separate SQL query.

The mapping tool thus needs to balance two forces: The fewer SQL queries it executes, the more likely there are cross products that lead to huge result sets. The more SQL queries it executes, the more database round trips add to the overall response time. In general, 1:1 associations should always be joined in. If an object contains one 1:n or n:m association only, the associated objects could also be joined in. If multiple such associations exist, further queries are necessary.

A prefetch filter is not restricted regarding the depth of the association graph. The decision how to load associated objects therefore needs to be applied recursively. If the mapping tool decides to join in an association, it has to consider what to do with the associated objects' own references: join them in as well or executing further queries for them.

Additionally, a prefetch filter may contain circular dependencies. The mapping tool therefore needs to keep track of what objects have already been loaded before to avoid loading them multiple times or being stuck in a circular loop.

To load the `Customer` object of the example, a mapping tool may reason as follows: At first, the query's target object, the customer, is loaded; all of its orders are loaded as well by joining them in. This decision is appropriate because joining in one 1:n association does not yet lead to cross products. To avoid cross products, however, the order items are not joined in.

Having loaded the customer with all of its orders, the mapping tool collects the `Order` objects' identifiers. Then, it executes another query to load all `OrderItem` objects that belong to the `Order` objects loaded before, using the collected identifiers as foreign key references. To also fetch the order items' `Product` objects, these objects are joined in.

Before the query returns the `Customer` object to the application, it needs to connect all objects it has loaded so that the application receives a network of fully interconnected objects.

Consequences

An Object Prefetch Filter makes it possible to explicitly define a graph of associated objects to be prefetched when a query is executed. Because an individual prefetch filter may be applied on a per-query basis, each individual service implementation can be optimized so that only the necessary objects are retrieved in advance.

The mapping tool knows which objects to prefetch before executing a query. It is therefore able to optimize the query execution by balancing the number of database round-trips (using appropriate `join` expressions) and the amount of data to be transfered from the database (executing additional queries to avoid cross products).

Loading a fully initialized network of objects at once simplifies passing these objects out of a session context, for example to the web presentation layer. In that case, neither does the presentation layer need to perform further calls to receive missing objects nor does the service layer need to manually traverse the object network to explicitly fetch all associated objects.

On the downside, a prefetch filter must exactly specify in advance which associated objects to load and remain consistent with the needs of the business logic implementation. If a prefetch fil-

ter is wrong, queries may load too few or too many objects into memory, leading to further database calls or to wasted memory, respectively.

Even if the service implementation cannot be corrupted (as long as loading on demand is available), there is therefore a danger of loosing performance during maintenance work. When business logic is changed, great care has to be taken that the prefetch filters are changed accordingly.

Known Uses

The author is only aware of one publicly available mapping tool that implements the OBJECT PREFETCH FILTER pattern: *Cayenne* ([Cayenne]). The experience and knowledge of this pattern is mainly based on proprietary mapping tools that were developed and used inside companies.

One such proprietary tool, *Dots*, implements the OBJECT PREFETCH FILTER pattern as described. Using the Dots API, retrieving a Customer object with all of its orders, order items, and products looks as follows:

```
Query query = service.createQuery("Customer");
query.setCondition(CB.equal("id", id));
IFilterGraph filter = FilterGraphFactory.createFilterGraph();
filter.addIncludes("orders", "orders.items", "orders.items.product");
IQueryResult result = service.execute(query, filter);
Customer customer = (Customer) result.getFirstObject();
```

The open source mapping tool *Cayenne* also implements the OBJECT PREFETCH FILTER pattern. Using Cayenne's query API, loading an `Order` object with all of its associated objects being prefetched looks as follows:

```
SelectQuery query =
  new SelectQuery(Order.class,
                  Expression.fromString("id = "+id));
query.addPrefetch("customer");
query.addPrefetch("shippingAddress");
query.addPrefetch("deliveryAddress");
query.addPrefetch("items");
query.addPrefetch("items.product");
Order order = (Order) context.performQuery(query).get(0);
```

To give a counter example, *Hibernate* ([Hibernate]), a very wide-spread mapping tool, currently does not support the pattern as described. However, it provides two prefetch strategies. First, when creating a query, an application may define which objects should be prefetched by joining them in. Using Hibernate's criteria API, the example of creating an overview list of all orders of a customer looks as follows:

```
Criteria criteria =
  session.createCriteria(Customer.class).
  add(Expression.eq("id", id)).
  setFetchMode("orders", FetchMode.JOIN);
Customer customer = (Customer) criteria.uniqueResult();
```

Although it is possible to prefetch objects to which there are 1:n or n:m relations, the Hibernate team advices against this because of cross products.

Besides dynamic prefetching, Hibernate also supports statically defined prefetching. Using appropriate mapping meta data, it can be specified that all objects that are involved in an association between specific types should be fetched from the database in blocks. The following meta data specification shows an example of this feature:

```
<class name="customer" table="customer">
  <set name="orders" fetch="subselect">
    <key column="customer_id"/>
    <one-to-many class="Order"/>
  </set>
</class>
```

The declaration `fetch="subselect"` makes Hibernate fetch all `Order` objects from all customers retrieved by a query when the `Order` objects of one customer are accessed for the first time. This behavior therefore also avoids the *n+1 selects* problem but not on a per-query basis.

Conclusion

The OBJECT PREFETCH FILTER pattern provides a solution for solving the *n+1 selects* problem in the context of implementing business logic that retrieves and updates persistent data. The paper explained how an object relational mapping tool could implement the pattern: it needs to provide an API to let an application define graphs of objects to prefetch and needs to implement an algorithm that efficiently prefetches objects as requested. By setting filters, an application is thus able to explicitly define which objects it needs in which case, therefore optimizing the execution of services on a per-query basis.

Acknowledgements

Many thanks go to my EuroPLoP shepherd Uwe Zdun who gave insightful feedback that helped me a lot to improve the paper.

References

[Bauer+2004] C. Bauer, G. King. *Hibernate in Action*. Manning, 2004

[C2Wiki] Portland Pattern Repository's Wiki: *Object Relational Tool Comparison*, 2007, http://c2.com/cgi/wiki?ObjectRelationalToolComparison

[Cayenne] Apache Cayenne, http://cayenne.apache.org

[Fowler2003] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003

[Han+2003] W.-S. Han, Y.-S. Moon, and K.-Y. Whang. PrefetchGuide: capturing navigational access patterns for prefetching in client/server object-oriented/object-relational DBMSs, In *Information Sciences: an International Journal*, 2003

[Hibernate] Hibernate, http://www.hibernate.org

[Keller1997] W. Keller. Mapping Objects to Tables – A Pattern Language. In *Proceedings of the 2nd European Conference on Pattern Languages of Programming*, 1997 http://www.objectarchitects.de/ObjectArchitects/papers/Published/ZippedPapers/mappings04.pdf

[Keller1998] W. Keller. Object/Relational Access Layers – A Roadmap, Missing Links and More Patterns. In *Proceedings of the 3rd European Conference on Pattern Languages of Programming*, 1997 http://www.objectarchitects.de/ObjectArchitects/papers/Published/ZippedPapers/or06_proceedings.pdf

[Ibrahim+2006] A. Ibrahim, W. Cook. Automatic Prefetching by Traversal Profiling in Object Persistence Architectures. In *Proceeding of the 20th European Conference on Object-Oriented Programming*, 2006

[Wellhausen2004] T. Wellhausen. Query Engine – A Pattern for Performing Dynamic Searches in Information Systems. In *Proceedings of the 9th European Conference on Pattern Languages of Programming*, 2004
http://www.tim-wellhausen.de/papers/QueryEngine.pdf

[Wellhausen2005] T. Wellhausen. User Interface Design for Searching – A Pattern Language. In *Proceedings of the 10th European Conference on Pattern Languages of Programming*, 2005
http://www.tim-wellhausen.de/papers/UIForSearching.pdf