УДК 004.4'2+004.6

Ф. Н. Юданов

Новосибирский государственный университет
ул. Пирогова, 2, Новосибирск, 630090, Россия
E-mail: fedwiz@academ.org

ИСПОЛЬЗОВАНИЕ ТЕХНОЛОГИИ AUTOFETCH ДЛЯ ОПТИМИЗАЦИИ ORM ЗАПРОСОВ К БАЗЕ ДАННЫХ НА ПРИМЕРЕ УНИВЕРСИТЕТСКОЙ ИНФОРМАЦИОННОЙ СИСТЕМЫ (УИС)

Статья посвящена описанию исследовательской работы по оптимизации информационной системы масштаба предприятия на уровне работы с базой данных через ORM средства (ORM, Object Relational Mapping − общее название технологий, отображающих объектную модель на реляционную базу данных). Основным направлением исследований стало определение эффективности такого типа оптимизации, как проставление директив предвыборки (prefetch) в те или иные ORM запросы системы с целью минимизации числа обращений к БД и устранения так называемой «N + 1 select problem». В качестве основного инструмента при проведении указанной оптимизации была использована библиотека Autofetch. Исследования проводились на примере Университетской информационной системы (УИС), разрабатываемой в ЦНИТ НГУ и активно эксплуатирующейся в НГУ, а также в нескольких других организациях.

Ключевые слова: предвыборка, оптимизация, ORM средства, УИС, Autofetch.

Введение

Университетская информационная система (УИС) [Адаманский, 2005; Адаманский и др., 2006] − обширный проект, разрабатываемый в течение 4 лет в Центре новых информационных технологий Новосибирского государственного университета (ЦНИТ НГУ). Система осуществляет автоматизацию внутренних процессов учебного заведения, а также отвечает за ведение электронного документооборота [1].

Система построена на технологиях JavaEE™ с использованием Hibernate в качестве ORM-средства [2]. Как и большинство программных систем уровня предприятия, УИС обладает обширным и сложно структурированным слоем хранимых объектов, который насчитывает более 140 классов сущностей.

Несмотря на то, что в целом скорость работы системы позволяет пользователю взаимодействовать с ней в режиме диалога, без ощутимых задержек, существуют варианты использования (в основном, касающиеся генерации сложных отчетов), занимающие определенное время, в нескольких случаях − вплоть до десятков минут. Этот факт вызвал желание провести некоторую оптимизацию продукта на уровне работы с БД.

Основным направлением исследований стало определение эффективности такого типа оптимизации, как проставление директив предвыборки (prefetch) в те или иные ORM запросы системы с целью минимизации числа обращений к БД и устранения так называемой «N + 1 select problem». Привлекательность этого подхода в его универсальности и, по крайней мере, теоретическом отсутствии необходимости разбирать логику работы каждого конкретного ORM запроса и алгоритма, манипулирующего выбранными данными. При достаточно сильных ограничениях на применяемые ORM средства, проведение такой оптимизации даже может быть полностью автоматизировано. К сожалению, в УИС добиться полной автоматизации затруднительно, ввиду невозможности удовлетворения этим ограничениям (этот момент будет подробно разобран далее в статье).

Проводимые исследования основываются на применении библиотеки Autofetch, разработанной Ali H. Ibrahim в Университете Техаса в Остине [Ibrahim, Hook, 2006]. Эта библиотека реализует эвристический алгоритм определения целесообразности предвыборки связанных

[1] Университетская информационная система. Презентация УИС для учебных заведений [Электронный ресурс]. Режим доступа: http://softmotions.com/data/uis-edu/uis_pres.ppt

[2] Университетская информационная система. Технические подробности [Электронный ресурс]. Режим доступа: http://softmotions.com/data/uis-edu/uis_description2.pdf

сущностей, а также автоматическую модификацию Criteria-запросов Hibernate, проставляя в них соответствующие настройки. Библиотека распространяется свободно в исходных кодах.

Теперь, прежде чем приступить непосредственно к описанию процесса внедрения библиотеки в УИС, а также результатов, которые были получены на начальном этапе исследований, рассмотрим подробно суть решаемой проблемы и алгоритм, лежащий в основе Autofetch.

Проблема

Проблема «N + 1 select» повсеместно возникает при использовании ORM средств и является специфичной именно для данного подхода. Чтобы пояснить ее суть, рассмотрим простой пример. Пусть есть хранимые сущности, определенные классами *A* и *B*:

```
public class A {
   private Long id;
   private B b;

   public Long getId();
   public void setId(Long id);

   public B getB();
   public void setB(B b);
}

public class B {
   public String getName();
}
```

И над ними производится следующий набор операций:

```
Collection<A> aCol = (Collection<A>)
                     Query.create("SELECT a FROM A a").list();
for (A a: aCol) {
   System.out.println(a.getB().getName());
}
```

Рассмотрим, как происходит в этом случае обмен информацией с БД.

В первом запросе мы получаем N объектов класса *A*. Далее для каждого из них мы вызываем метод, который для получения нужного экземпляра класса *B* внутри себя также делает запрос к БД (так называемый navigational query). Итого мы получаем те самые N + 1 запросов, в то время как в исходной программе содержится всего один.

Надо отметить, что в данном примере подразумевается использование «ленивой» (lazy) стратегии выборки. Это означает, что хранимый объект выбирается только тогда, когда он действительно используется программой. Эта стратегия не единственно возможная − существует альтернативный подход, называемый «активным» (eager), когда все связанные объекты выбираются немедленно после возникновения ссылок на них. В нашем примере это привело бы к тому, что N объектов класса *B* было бы получено до вхождения в цикл (при этом число запросов по-прежнему было бы равно N + 1).

Очевидно, в большинстве случаев ленивая стратегия выборки является более предпочтительной, поскольку так мы избавлены от риска получения информации, которая никогда не будет использована (к этому вопросу мы еще вернемся в данной статье).

Описанная же проблема становится проблемой, когда время, требуемое для обращения к базе, начинает ощутимо превышать время выполнения других операций. В значительном количестве случаев происходит именно так. А если число объектов, выбранных первым запросом (т. е. число N) становится очень большим, потери в производительности также получаются критическими.

Чтобы сократить число запросов, используются директивы предвыборки, заставляющие ORM получать связанные сущности в одном запросе с основной. В нашем случае (HQL − Hibernate Query Language) это бы выглядело так:

```
Collection<A> aCol = (Collection<A>) Query.create(
   "SELECT a FROM A a LEFT JOIN FETCH a.b").list();
```

Теперь данные обо всех связанных объектах *B* будут получены в одном SQL запросе с данными об *A*, а сами объекты инициализированы сразу после выполнения вышеприведенной инструкции, и поэтому метод *getB()* теперь не будет обращаться к БД. Таким образом, будет выполнен всего один запрос, а не N + 1. Следует, однако, понимать, что объем переданной из БД информации, как и суммарная сложность выборки, остается тем же (или примерно тем же).

Но здесь возникает новая трудность. Перепишем пример немного по-другому:

```
Collection<A> aCol = (Collection<A>) Query.create(
   "SELECT a FROM A a LEFT JOIN FETCH a.b").list();
for (A a: aCol) {
   if (1 == a.getId()) {
        System.out.println(a.getB().getName());
   }
}
```

В данном случае в программе используется не более одного объекта класса *B*, в то время как FETCH выбирает все N. Скорее всего, это означает, что вместо оптимизации мы имеем просадку производительности.

Таким образом, директивы FETCH требуют аккуратности в использовании. Определение же целесообразности применения предвыборки в том или ином запросе полноценного программного проекта, где число запросов исчисляется сотнями, а алгоритмы, работающие с объектами, намного сложнее приведенных в примере, становится серьезной задачей.

Следуют также упомянуть вторую распространенную проблему, возникающую при использовании ORM инструментов, которая является в какой-то степени противоположной первой. Речь о «проблеме декартова произведения» («Cartesian product problem»).

Дело в том, что предвыборка, как правило, реализуется через механизм OUTER JOIN, который работает таким образом, что в случае присоединения множественной связи выдает результат, объем которого пропорционален мощности связанного множества. При этом данные, которые передаются от БД, оказываются крайне избыточными. Эта избыточность обычно устраняется ORM библиотекой при обработке ответа от БД, однако обработка, а также собственно передача данных могут занять значительное время.

Вышесказанное означает, что, делая предвыборку множественной связи, мы получаем выигрыш в количестве запросов к БД ценой необходимости обработки некоторого объема избыточных данных. В зависимости от ситуации такая предвыборка может оказаться как удачным, так и провальным решением.

Autofetch

Алгоритм. Идея алгоритма Autofetch основана на понятии обхода (traversal) графа объектов (object graph) − графа составленного из всего множества хранимых в БД объектов и связей между ними.

Формально граф объектов можно представить как тройку *(O,F, E)*, где

1. *O* − множество всех хранимых объектов;

2. *F* − множество всех полей у классов хранимых объектов;

3. *E*: $(O \times F) \rightarrow P(O)$ − частичная функция, сопоставляющая паре (объект, поле) подмножество из *O* (*P(O)* – множество всех подмножеств из *O*). Значение *E(o,f)*, в случае, если оно определено, представляет собой множество объектов, которое является значением поля *f*

у объекта *o*. Мощность этого множества равна 1, если связь простая и может быть больше 1, если связь множественная.

Иными словами, граф объектов представляет собой структуру всех хранимых объектов, где дуги обозначают связи между объектами (рис. 1).

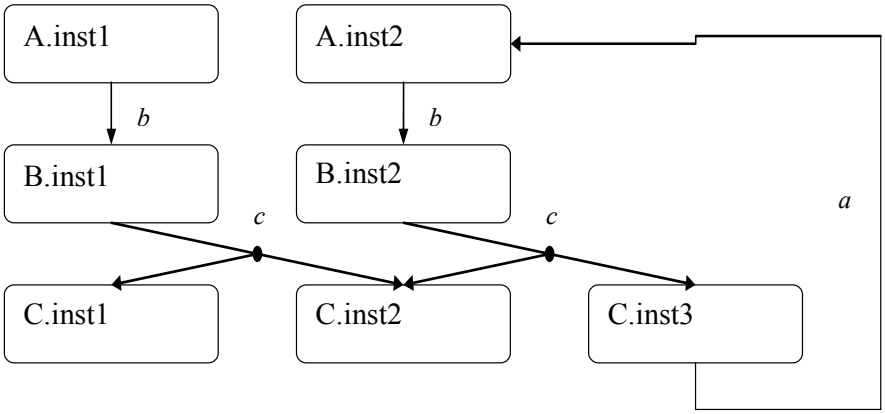

Рис. 1. Пример графа объектов. Связи через поля *A.b* и *C.a* − простые, *B.c* − множественные

Запросом (query) назовем функцию, выделяющую из графа объектов некоторый подграф, причем каждая компонента связности этого подграфа обязана содержать как минимум один объект некоторого заданного типа (корневой объект). Множество корневых объектов (root objects) − это тот набор сущностей, который мы изначально получаем, выполнив ORM запрос к БД, в то время как структура связанных сущностей образует весь остальной подграф.

Обходом графа объектов назовем лес (множество деревьев), включающийся в подграф запроса таким образом, что корнем каждого дерева является корневой объект.

По фрагменту программного кода, оперирующему результатами ORM запроса, можно очевидным образом построить обход графа объектов:

1) первичные результаты запроса назначаем корневыми объектами;

2) объект *o* вместе с дугой *(o, o*1*)* включается в обход, если этот объект во время исполнения фрагмента загружается из БД через связь с некоторым объектом *o*1 из обхода;

3) если существует несколько путей включения объекта *o* в обход, выбираем произвольный из кратчайших, остальные исключаем.

Таким образом, построенный обход дает нам информацию о том, какие хранимые объекты используются в программе, и какие связи задействованы при их выборке. Важно понимать, что один и тот же запрос может порождать совершенно разные обходы, в зависимости от контекста использования результатов запроса (рис. 2).

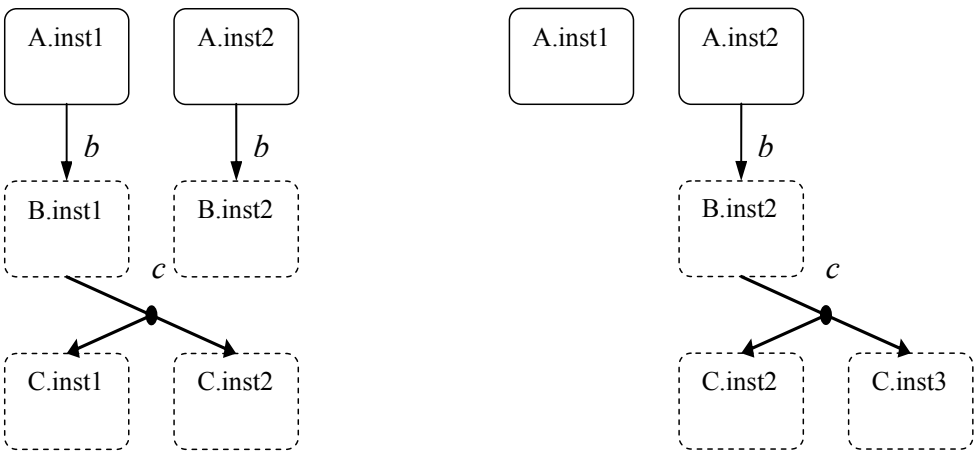

Рис. 2. Два различных обхода графа объектов из рис. 1. Корневыми являются объекты класса *A*

Профилем обходов (traversal profile) назовем дерево, каждая вершина которого помечена именем класса хранимых объектов, а дуга − тройкой *(f, used, potential)* из *(F × N × N)*.

Профиль обходов строится на основе множества обходов, порожденных одним и тем же запросом:

1) корень дерева помечаем классом корневых объектов;

2) каждая вершина сопоставляется классу эквивалентности объектов из обходов. Объекты *o*1 и *o*2 из обходов *t*1 и *t*2 назовем эквивалентными, если цепочки дуг, ведущих к *o*1 и *o*2 от корневых объектов, совпадают в смысле совпадения имен связанных с дугами полей. Говоря неформально, объекты одного класса эквивалентности получены в программе одним и тем же способом.

Очевидно, что класс в смысле ООП у объектов из одного класса эквивалентности совпадает – этим классом помечаем вершину;

3) вершину *v*1 соединяем дугой с *v*2 и помечаем эту дугу именем поля *f*, если объекты из класса *v*2 были получены через связь (простую или множественную), заданную через поле *f* у объектов из *v*1. В силу построения множества вершин, поле *f* выбирается единственным образом;

4) величины *used* и *potential* проставляем по следующему алгоритму:

a) изначально для каждой вершины выставляем *used* = 0 *potential* = 0

b) для каждого обхода и каждого объекта из этого обхода у вершины, соответствующей классу эквивалентности этого объекта увеличиваем *used* = *used* + 1, а для всех вершин, соответствующих классам эквивалентности объектов, связанных с исходным, увеличиваем *potential* = *potential* + 1**.**

Итак, величина *used* для каждой связи характеризует, сколько раз объекты были действительно получены по этой связи, а *potential* – сколько раз были доступны ссылки на объекты через эту связь.

Очевидно, для каждой дуги профиля *used* ≤ *potential* (рис. 3).

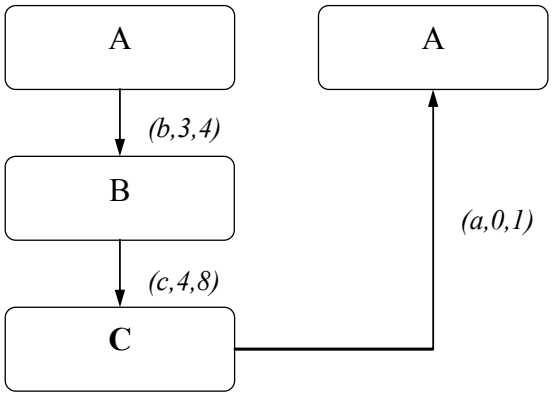

Рис. 3. Профиль обходов, построенный на основе двух обходов из рис. 2

Величина *used / potential* и является критерием целесообразности предвыборки сущности по данной связи. Так, значение 1 говорит о том, что объект используется в программном фрагменте каждый раз, когда возникает на него ссылка. Значение 0, наоборот, указывает, что объект не используется никогда и предвыборка по данной ссылке является бесполезной.

Численная оценка целесообразности предвыборки по связи *f(v)* строится на основе величины *used / potential***:**

1) *f(r)* = 1, где *r* – корень дерева;

2) *f(v) = (used / potential) * f(u)*, где *u* − родитель *v* в дереве, а величины *used* и *potential* берутся у дуги *(u,v)*.

Реализация. Библиотека Autofetch реализует описанный алгоритм на основе Hibernate 3.1. Библиотека работает за счет проксирования хранимых объектов, и запросов. Для каждого запроса строится профиль обхода и при каждом обходе модифицируются величины *used* и

potential. При очередном выполнении запроса на основе профиля автоматически расставляются директивы предвыборки.

Здесь следует ввести критерий идентичности одного запроса другому. Как уже упоминалось, профиль обходов существенно зависит от программного кода, использующего данные, поэтому строить подобный критерий на основе текста запроса или расположения запроса в программе было бы не лучшим вариантом.

В итоге, в Autofetch критерий уникальности запроса опирается на два элемента: собственно текст запроса и состояние стека вызовов на момент его выполнения (при этом некоторые вызовы разумно не учитывать, например вызовы собственно объектов библиотеки Autofetch).

Несмотря на относительную простоту идеи, лежащей в основе алгоритма Autofetch, его реализация с помощью Hibernate достаточно сложна, задевает глубинные механизмы функционирования Hibernate, включает в себя сложную систему проксирования объектов и перехвата операций над ними.

Также Autofetch переопределяет компонент HbmBinder, разбирающий и применяющий Hibernate-спецификации хранимых объектов.

Последний факт, значительно сужает возможности Hibernate, которые могут быть применимы вместе с Autofetch (далее этот вопрос будет рассмотрен подробнее на примере реально возникших сложностей).

Тем не менее, желание применить к готовой, отлаженной системе описанный выше алгоритм заставляет искать способы адаптировать имеющуюся реализацию к конкретным условиям, так как написание своей реализации алгоритма с нуля представляется намного более затратным решением.

Внедрение Autofetch

AutofetchQuery. Одной из первоочередных задач при внедрении Autofetch в УИС было расширение функциональности библиотеки с целью получения возможности обработки запросов на языке HQL (язык ORM запросов, разработанный специально для Hibernate и синтаксически схожий с SQL, − именно таким способом представлено в УИС абсолютное большинство запросов к модели данных). Дело в том, что в том виде, в котором Autofetch предоставляется разработчиками, библиотека поддерживает лишь узкий класс запросов − так называемые Criteria запросы. Они отличаются тем, что управление ими производится целиком через программный интерфейс, что зачастую удобно, но сильно ограничивает разработчика в возможностях. Упомянутый программный интерфейс позволяет задавать такие параметры запроса, как тип выбираемой сущности, условия выборки, а также интересующие нас директивы предвыборки связанных объектов.

Именно возможность динамического проставления директив позволяет библиотеке регулировать и менять в произвольный момент параметры предвыборки того или иного запроса. К сожалению, интерфейс объекта Query, отвечающего за обработку HQL запросов не дает такой возможности. Все управление запросом осуществляется целиком на языке HQL, а внутренняя механика разбора и интерпретации текста запроса скрыта глубоко внутри реализации.

Поэтому, чтобы обеспечить динамическую саморегуляцию предвыборки в системе аналогично тому, как это происходит с Criteria-запросами, требуется освоить синтаксический анализатор языка HQL (можно взять готовый или написать свою специфичную реализацию).

Такой шаг был признан неоправданно дорогим в смысле расхода сил и времени, особенно с учетом того факта, что выгода от применения Autofetch в УИС на тот момент была неочевидна.

Поэтому решено было ограничиться следующей стратегией: полученная от Autofetch информация о директивах предвыборки записывается в файл (наиболее предпочтительным форматом был признан XML), который после некоторого времени работы УИС вместе Autofetch должен содержать в себе полный набор диагностических данных. Затем эти данные используются для ручной расстановки директив и затем, измерив время работы фрагментов системы до и после расстановки, мы производим оценку эффективности работы Autofetch.

Класс AutofetchQuery был написан по аналогии с уже имеющимся классом AutofecthCriteria путем проксирования Hibernate-объектов типа Query и перегрузки методов, вызывающих

обработку и выполнение запроса. В эти методы была добавлена процедура создания и обновления профиля обходов, а также журналирование диагностированных предвыборок.

Также был написан прокси-класс AutofetchSession, автоматически оборачивающий все запросы Hibernate в AutofetchQuery.

Описанная стратегия проксирования позволила при внедрении Autofetch в УИС ограничиться самыми минимальными изменениями, в то время как код, отвечающий за бизнес-логику системы, удалось оставить без изменений. Таким образом, присутствие Autofetch остается незамеченным для абсолютного большинства компонентов системы.

Hibernate Annotations. Первые попытки внедрения Autofetch в УИС привели к пониманию, что библиотека не только не рассчитана на работу с использованным в УИС методом описания хранимых сущностей − Hibernate Annotations, но, напротив, завязана на альтернативном методе − файлах hbm.xml.

Два указанных подхода в равной степени допускаются Hibernate. В обоих случаях можно выделить класс хранимого объекта и некоторые инструкции о его отображении на таблицы в БД. Эти инструкции имеют вид либо Java аннотаций, помещенных в исходный код класса в качестве метаданных, либо тегов XML, вынесенных в отдельный файл. Надо отметить, что два подхода отличаются весьма существенно, и неэквивалентны с точки зрения выразительных способностей.

Итак, возникла задача конвертирования модели данных УИС в другой формат. Ввиду сложности и значительных размеров модели данных, упомянутое конвертирование хотелось провести автоматически.

Выяснилось, что такая возможность есть, ее предоставляет библиотека Hibernate Tools, которая позволяет генерировать на основе аннотированных классов hbm.xml описания.

К сожалению, опыт показал, что возможности библиотеки не покрывают всего множества аннотаций, используемых в УИС. В частности, пришлось искать обход следующих ограничений.

1. Для поддержки аннотации @OneToOne, которая очень часто используется при описании связей между сущностями (связь «один к одному»), потребовалось написать шаблон на языке freemarker, позволяющий корректно обрабатывать большинство случаев применения @OneToOne (кстати, именно такой совет − реализовать недостающие функции самостоятельно, дают разработчики Hibernate Tools). Несколько случаев нетривиального использования данной аннотации было переписано вручную.

2. Все элементы, аннотированные @CollectionOfElements, были из модели удалены, так как данная аннотация (задающая множество объектов, не являющихся хранимыми) не поддерживается, и, более того, аналогов этой директивы в языке описаний hbm.xml просто нет.

3. Директивы CascadeType.DELETE_ORPHAN (указывающие на необходимость удаления связанного объекта после потери ссылки на него) были вырезаны, поскольку, с одной стороны, вызывали ошибку при конвертировании, а с другой – они не оказывают никакого влияния на процесс выборки данных.

4. Некорректно обрабатывались сущности, связанные отношением наследования, а именно не проставлялось значение поля discriminator, которое указывает на конкретный тип сущности, хранимой в данной строчке таблицы. Поэтому поле discriminator было заполнено вручную в уже сгенерированных xml-описаниях.

На этом этапе стала очевидной необходимость создания отдельной копии УИС для сбора диагностики, поскольку изменения, требуемые для подключения Autofetch к системе, приводили к некоторому (небольшому) ограничению по функциональности, а также могли стать причиной нестабильности в работе системы.

Это на данный момент затрудняет сбор диагностической информации на основе работы системы в режиме реальной эксплуатации. Обеспечение такой возможности является одной из первоочередных задач для дальнейшего развития исследований.

Модификация Autofetch. В библиотеку Autofetch также потребовалось внести некоторые изменения.

В основном они касались внедрения AutofetchQuery в механизмы Autofetch, а также включали обеспечение возможности переводить описания в виде аннотаций в hbm.xml.

Подробное описание изменений не представляет интереса с точки зрения методологии, поэтому в данной статье будет опущено.

Процесс оптимизации

Для тестирования эффективности применения методики Autofetch в УИС было выбрано несколько простых отчетов по студентам и преподавателям факультетов. Эти отчеты были сгенерированы несколько раз под контролем Autofetch (с подстановкой различных параметров, например, для разных факультетов).

На выходе мы получили набор запросов со ссылками на участки кода и указаниями, к каким связанным сущностям следует применить предвыборку.

Пример файла с диагностикой

```
<?xml version="1.0" encoding="UTF-8" ?>
    <queries>
        <query string="SELECT ste FROM StaffTableEntrySupportHbm AS ste
                    WHERE ste.timeWorker = true AND ste.person IS NOT NULL ">
            <stack>
                <![CDATA[java.lang.Thread.getStackTrace(Thread.java:1436),
                com.softmotions.univer.curricula.exec.reports.TeacherHoursReportExecutor.
                printTeacherHoursList(TeacherHoursReportExecutor.java:88),
                sun.reflect.NativeMethodAccessorImpl.
                invoke(NativeMethodAccessorImpl.java:39)
                mx4j.AbstractDynamicMBean.invoke(AbstractDynamicMBean.java:231)]
                ]]>
            </stack>
            <fetches>
                <fetch path="person" />
                </fetches>
        </query>
    </queries>
```

Далее, в код УИС были внесены соответствующие диагностике изменения и проведены измерения времени генерации отчетов до и после применения директив предвыборки.

Первые результаты показали, что время генерации большинства отчетов после проставления предвыборки ухудшилось. Это заставило заняться детальным изучением и отладкой процессов, происходящих во время применения HQL запроса и последующего обращения к полям хранимых объектов.

Обнаружилось, что выполнение HQL запроса часто порождает набор индуцированных запросов к БД, причем во многих случаях совершенно лишних с точки зрения логики программы. И если мы делаем предвыборку каких-либо сущностей, число таких запросов возрастает.

Причина описанного заключается в активной (eager) стратегии выборки полей хранимых объектов, которые в этом случае загружаются из БД непосредственно после того, как были выбраны родительские объекты, при этом значения таких полей могут быть вообще не используемы в вызывающем коде, что вызывает неоправданные потери в производительности.

Итак, мы пришли к пониманию важной вещи: применение Autofetch в проекте требует полного контроля над стратегией выборки объектов. Этот контроль автоматически обеспечивается, когда мы ограничиваемся Criteria-запросами, однако для общего случая требуется модификация модели данных, для устранения лишних запросов, порожденных активной стратегией выборки.

Здесь надо принимать во внимание два факта.

1. Если сам по себе Hibernate гарантирует, что по умолчанию связанные сущности выбираются лениво (по стратегии lazy), то для Hibernate Annotations в силу их наследования от EJB это не так. Для связей @ManyToOne и @OneToOne по умолчанию используется стратегия Eager.

2. Существуют ситуации, когда ленивая выборка невозможна в принципе, а явные указания стратегии выборки игнорируются. В УИС такой случай обнаружился в тех местах, где

аннотация @OneToOne была использована с атрибутом mappedBy, который инвертирует связь, что позволяет сделать ее односторонней в том смысле, что лишь одна из сущностей хранит в себе информацию о связи. Это бывает удобно, поскольку в этом случае выставление связанной сущности требуется лишь с одной стороны (проще говоря, достаточно вызова одного метода, вместо двух), но такая связь через Hibernate не может быть выбрана лениво никогда.

Поэтому первым серьезным шагом в оптимизации хранимого слоя УИС стали:

1. Проставление директив FetchType.LAZY везде, где это значение не подразумевается по умолчанию.

2. Рефакторинг связей с mappedBy. К счастью, это можно сделать достаточно безболезненно для проекта, путем добавления средствами прямого доступа к БД (которые в том числе предоставляет и Hibernate) дополнительной колонки с id связанной сущности в таблицу, хранящую данные сущности, из которой мы убираем mappedBy.

Надо отметить, что связи с mappedBy присутствовали практически во всех наиболее часто используемых сущностях в УИС, поэтому эффект от приведенных выше оптимизаций оказался значительным (время генерации тестируемых отчетов улучшилось от 1,5 до 4 раз).

После этого были повторены измерения с помощью Autofetch. Новые результаты демонстрировали поведение полностью соответствующее предсказанному в смысле уменьшения числа запросов, однако реальный выигрыш по времени был не столь значителен, а в некоторых случаях и исчезающе мал.

Скорее всего, это можно объяснить тем, что алгоритмы генерации большинства отчетов независимо от предвыборки порождают число запросов к БД линейное от количества изначально выбранных сущностей. Последнее уже не связано с проблемой «N + 1 selects», поскольку сами запросы не являются следствием обращениям к связям изначально выбранных сущностей.

Можно сделать вывод, что не всегда организация предвыборки дает ощутимые преимущества. Надо отметить, что в статье разработчика Autofetch [Ibrahim, Hook, 2006] приводится информация о тестировании библиотеки на различных наборах данных и запросов. При этом указываются как случаи, которые дают небольшую просадку производительности ~ 8 % (величиной этой просадки авторы оценивают накладные расходы на применение Autofetch), так и случаи, когда Autofetch на 98 % уменьшает число запросов к БД и в той же пропорции улучшает время работы примера. Понятно, что такие примеры целиком искусственны и на реальной функционирующей системе к таким показателям невозможно приблизится даже отдаленно.

Тем не менее эксперименты показали, что выигрыш от применения Autofetch в УИС может быть получен. Предварительно можно сказать, что он достигает наибольшей величины на сложных и длительных операциях с данными и составляет 12−25 %:

Отчет по факультетам (2007−2008)

	Без оптимизации	После оптимизации стратегии выборки	После подстановки директив Autofetch
Время генерации (мс)	280 172	85 516	64 484
Выигрыш относительно предыдущего	–	69,5 % (в 3,3 раза)	24,6 % (в 1,3 раза)

Отчет по кафедрам (2007−2008, ММФ)

	Без оптимизации	После оптимизации стратегии выборки	После подстановки директив Autofetch
Время генерации (мс)	80 735	22 593	19 812
Выигрыш относительно предыдущего	–	72,0 % (в 3,6 раза)	12,3 % (в 1,14 раза)

Список преподавателей (2007−2008, все кафедры)*

	Без оптимизации	После оптимизации стратегии выборки	После подстановки директив Autofetch
Время генерации (мс)	68 156	69 516	67 688
Выигрыш относительно предыдущего	–	−1,9 %	2,6 %

* Никакая из оптимизаций не работает, так как все время расходуется на одном HQL запросе.

Другие методы оптимизации

Как уже отмечалось ранее, оптимизация с помощью библиотеки Autofetch привлекательна тем, что не требует изучения контекста выполнения каждого запроса. Для многих вариантов использования системы такая «поверхностная» оптимизация может оказаться достаточной. Однако часто существуют способы добиться намного более значительного выигрыша в производительности.

Один из таких способов − настройка стратегии выборки данных, уже был подробно разобран в данной статье по той причине, что без него применение Autofetch в большинстве случаев бессмысленно.

Также нужно отметить следующие способы [Bauer, King, 2006]:

1. Выборка проекции. Эта оптимизация может быть крайне полезна, если время инстанцирования выбираемых объектов становится значительным по сравнению с общим временем работы программного фрагмента. В этом случае имеет смысл совсем отказаться от создания хранимых объектов, а из БД выбирать только те поля, которые используются программой (они и составляют проекцию объекта). Минус данного метода заключается в снижении читаемости и поддерживаемости кода, поскольку приходится заменять удобные и интуитивно понятные обращения к методам объекта, на операции с разрозненными данными. Но итоговый выигрыш может компенсировать этот момент.

Так, для алгоритма генерации списка преподавателей в УИС (3-я таблица из предыдущего пункта) удалось уменьшить время работы почти в 14 раз.

2. Рефакторинг модели данных. В случае если в системе обнаруживается сущность, которую всегда выгодно использовать в проекции, встает вопрос о целесообразности существования такого класса объектов вообще. Как правило, его следует разбить на несколько подклассов. В отличие от предыдущей оптимизации, этот метод требует глобальных изменений в проекте, однако, методологически он правильней и позволяет сохранить прозрачность программного кода.

Не исключено, что в дальнейшем возникнет потребность в применении этого метода и в УИС.

3. Применение кэша второго уровня (second-level cache) и кэша запросов (query cache). Hibernate предоставляет средства для тонкой настройки стратегии кэширования как отдельных объектов, так и их совокупностей как результатов запроса.

Например, в случае, если некоторый запрос выполняется многократно с одними и теми же параметрами, избыточность может быть легко устранена вызовом метода Query.setCacheble(true). Полезность такой оптимизации в каждом случае определяется путем анализа журнала обращений к БД.

4. Наконец, ручная оптимизация алгоритма программы или HQL запроса, иногда сводящаяся к переписыванию отдельных фрагментов заново. Наиболее общий и трудоемкий подход, но в некоторых случаях серьезный эффект может дать только он.

Эти способы не имеют прямого отношения к рассмотренной задаче оптимизации путем расстановки предвыборок, однако все они могут быть использованы в сочетании с этой оптимизацией и, по всей видимости, будут (или уже были) применены в процессе работы над УИС.

Заключение

В данной статье были изложены начальные этапы исследовательской работы по оптимизации УИС на уровне представления данных. В основном они включали в себя процесс внедрения библиотеки Autofetch в проект с разбором возникших трудностей и анализ эффективности применения библиотеки и оптимизации предвыборками вообще. Изложенная информация может быть полезна в качестве некоторого опыта практического применения данной методики, который позволяет оценить целесообразность такого применения при оптимизации системы масштаба предприятия, построенной на JavaEE™.

Что касается собственно работы над УИС, она будет продолжена в следующем виде.

1. Достижение путем модификации Hibernate Tools возможности автоматически (без вмешательства программиста) конвертировать модель данных УИС в нужный формат.
2. Тестирование УИС с подключенным Autofetch, с целью гарантировать стабильность работы системы в таком режиме.
3. Сбор диагностической информации о предвыборках на основе длительной работы УИС в режиме реальной эксплуатации.
4. Расстановка директив предвыборки в соответствии с полученной диагностикой.
5. Выявление вариантов использования, для которых необходима более глубокая оптимизация и проведение этой оптимизации с помощью методик, описанных в разделе «Другие методы оптимизации».

Данные, полученные на текущий момент и приведенные в статье, позволяют рассчитывать на то, что с помощью изложенных действий удастся достичь желаемых результатов и добиться оптимального использования ORM в УИС.

Список литературы

Адаманский А. В. Архитектура контейнера программных компонент Jaxion // Вестн. Новосиб. гос. ун-та. Серия: Информационные технологии. 2005. Т. 2, вып. 1. С. 88–91.

Адаманский А. В., Денисов А. Л., Кочеев А. А. Опыт автоматизации вуза. Система УИС // Вестн. Новосиб. гос. ун-та. Серия: Информационные технологии. 2006. Т. 4, вып. 1. С. 2–6.

Ibrahim A. H., Hook W. R. Automatic Prefetching by Traversal Profiling in Object Persistence Architectures // ECOOP 2006. 2006. P. 50–74.

Bauer Ch., King G. Java Persistence with Hibernate. N.Y.: Manning Publications Co. 2006.

Материал поступил в редколлегию 27.05.2008

F. N. Yudanov

ORM QUERIES OPTIMIZATION USING THE AUTOFETCH TECHNOLOGY BY THE EXAMPLE OF UNIVERSITY INFORMATIONAL SYSTEM (UIS)

The current article describes the research work to optimize enterprise level informational system. The optimizations were focused on persistence layer based on ORM tool (Object Relational Mapping − a programming technique for converting data between incompatible type systems in relational databases and object-oriented programming languages). The main line of investigation was to determine the efficacy of such optimization type as data prefetching in ORM queries to minimize the number of interactions with database and to solve so-called «N + 1 select problem». The most of optimizations were made using the Autofetch library. All researches were based on University Informational System (UIS) − the development of CNIT (Center of New Informational Technologies) NSU. The system is currently in exploitation in NSU and a few other organizations.

Keywords: prefetch, optimization, ORM tools, UIS, Autofetch.