Способы отображения объектов в реляционных базах данных[*]

В.В. Рубанов

Аннотация. В статье рассматриваются различные способы обеспечения объектно-реляционного отображения. Проводится анализ соответствия объектной и реляционных моделей. Рассматриваются, в частности, вопросы представления в реляционных таблицах отдельных атрибутов объектов, массивов и коллекций, связей и объектных ссылок. Далее исследуются вопросы организации отображения отдельных классов и целых иерархий наследования. Также описываются тонкости объектно-реляционного отображения, связанные с обеспечением максимальной производительности. Приводятся конкретные выводы и рекомендации.

1. Введение

В данной статье рассматривается современное состояние дел в области средств хранения данных объектно-ориентированных программ в реляционных базах данных. Центральное место занимает исследование различных способов организации так называемого *объектно-реляционного отображения* (object-relational mapping). Обеспечение такого механизма является основной задачей при создании систем долговременного хранения объектов в реляционных базах данных. Трудности возникают из-за сильного несоответствия понятий объектной и реляционных моделей. Данная статья ставит своей целью описать специфику каждой из этих моделей и затем на основе этой информации сделать выводы о возможных путях решения исходной задачи путем комбинации классических и современных решений.

В первом разделе приводится собственно описание объектной модели, специфицированной ODMG (см. [10]), и описание понятий классической реляционной модели Кодда. Затем дается общий обзор соответствия элементов данных моделей. Второй раздел посвящен описанию конкретных решений, обеспечивающих отображение объектных понятий в реляционные, приводятся

[*] Статья основывается на результатах проектов, выполнявшихся в ИСП РАН по грантам РФФИ № 96-07-89591 и № 97-01-00142.

конкретные выводы и рекомендации, в том числе по вопросам производительности.

2. Реляционная и объектная модели. Соответствие между ними

2.1. Объектная модель ODMG

Данный подраздел содержит описание терминологии, необходимой для дальнейшего изложения. В нем определяется *объектная модель*, принятая ODMG в качестве стандарта (см. [10]). В этой модели специфицируются конструкции, которые должны поддерживаться любой ООСУБД:

- Базовыми примитивами являются *объект* и *литерал*. Каждый объект обладает уникальным *идентификатором* (OID). Литерал не имеет идентификатора.
- *Состояние* объекта определяется как совокупность значений его *свойств*. Свойства делятся на *атрибуты* и *связи*.
- *Поведение* объекта определяется набором *методов* этого объекта.
- *Тип* объединяет два понятия *интерфейса* и *класса*.
- *Интерфейс* специфицирует только абстрактное поведение своих экземпляров (без описания состояния).
- *Класс* специфицирует абстрактное поведение и абстрактное состояние своих экземпляров.

В БД хранятся объекты, доступ к которым, вообще говоря, может осуществляться из нескольких различных приложений. БД основывается на *схеме классов*, которая определяется на этапе разработки приложения и описывается на одном из специальных языков (*ODL - Object Definition Language, Java, UML*). В стандарте ODMG специфицируется язык ODL. БД содержит экземпляры типов (объекты), описанных в данной *схеме*.

Тип обладает внешней *спецификацией* и одной или несколькими *реализациями*. *Спецификация* определяет внешние характеристики типа, т.е. характеристики, которые видны пользователям этого типа, а именно:

- Атрибуты или переменные состояния (если они есть), значения которых можно получить для каждого экземпляра типа;
- Методы, присущие экземплярам типа;
- Исключения (exceptions), которые могут возникнуть в процессе работы методов типа.

Реализация содержит внутренние детали и аспекты механизмов работы экземпляров типа. *Реализация* включает в себя *представление* и набор *методов.*

Представление – это структура данных, получаемая из определения абстрактного типа на ODL после отображения в конкретный язык программирования (*ODL to program language mapping*). Каждому атрибуту абстрактного типа соответствует переменная в его представлении.

Методы – это тела операций (процедур), объявленных в спецификации данного типа, написанные с использованием классов, предоставляемых ОО средой.

Тип может обладать несколькими реализациями, хотя обычно только одна из них используется в конкретной программе.

Разделение спецификации и реализации типа, при которой спецификация описывается независимым от языка программирования образом, обеспечивает прозрачный доступ к объекту в неоднородном компьютерном окружении (соответствующий интерфейс на стороне клиента генерируется специальной утилитой – ODL/IDL компилятором).

2.1.1. Виды наследования

Зависимости между типами в ОМ представляются двумя видами наследования, которые отражают соответственно *наследование поведения* и *наследование состояния.*

2.1.2. Наследование поведения

Данному типу наследования соответствует понятие *ISA-связи* (*is–a–relationship*) – связи наследования между типами выражающей обобщение, когда супертип наследует поведение подтипа. Объектная модель поддерживает множественные *ISA*-связи наследования между интерфейсами. На языке *ODL ISA*-связи представляются двоеточием:

```
interface Person{…};
interface Employee {...};
interface Professor : Employee {...}, Person {…};
interface Associate_Professor : Professor {...};
```

2.1.3. Наследование состояния

ODMG определяет также связь наследования (***extends***), когда наследуется состояние объекта класса:

```
class Person {
      attribute string name;
      attribute Date birthDate;
```



```
};

class EmployeePerson extends Person : Employee {
      attribute Date hireDate;
      attribute Currency payRate;
      relationship Manager boss
            inverse Manager::subordinates;
};
```

Extends-связь выражает наследование поведения и состояния типа и не допускает множественного наследования.

2.1.4. Агрегированные типы

Объектная модель *ODMG* поддерживает различные типы объектов-агрегатов, которые состоят из элементов литеральных или объектных (возможно, агрегированных) типов. Среди прочих поддерживаются следующие агрегированные типы (t означает тип элементов агрегата):

```
Bag<t>
List<t>
```
Array<t>

Интерфейсы всех этих типов наследуются от типа Collection, который является абстрактным интерфейсом, объединяющим общие характеристики. Семантически Bag является неупорядоченным мультимножеством (с возможностью содержания одинаковых элементов). List это упорядоченное множество элементов. Array соответствует обычному понятию массива с возможностью динамического изменения размера. Соответствующие реализации этих типов в Java имеют такие же имена, но с префиксом D, то есть DBag, DList, и т.д.

2.1.5. Атрибуты

Как уже упоминалось выше, состояние объекта определяется набором значений его *свойств,* подразделяющихся на *атрибуты* и *связи*. Атрибуты составляют внутреннее состояние объекта. Они могут быть элементарных и агрегированных литеральных или объектных типов. При этом последние выделяются особо, так как включение одним объектом в свой состав других объектов является типичным для ОО программирования. Они называются *атрибутами-ссылками.* Реально объекты ссылаются друг на друга посредством своих идентификаторов (*объектной ссылки OID* или *идентификатора хранимого объекта POID*), поэтому значение атрибута-ссылки – это объектная ссылка OID некоторого объекта в приложении или POID в БД. При отображении хранимых объектов из БД в приложение

идентификаторы хранимых объектов соответствующим образом преобразуются в объектные ссылки.

Заметим, что атрибуты являются *абстрактным* описанием состояния объекта и не обязаны соответствовать структурам данных при отображении в конкретный язык программирования. Они могут реализовываться и как набор соответствующих методов.

2.1.6. Связи

Тогда как *свойства-атрибуты* объекта характеризуют его внутреннее состояние, *свойства-связи* используются для задания семантических связей (отличных от связей наследования) с другими типами в спроектированной пользователем Объектной Модели Задачи (ОМЗ). Отметим, что *связи* – не то же самое, что атрибуты-ссылки. Атрибуты-ссылки характеризуют состояние объекта, которому принадлежат, путем задания ссылки на объекты, входящие в состав данного и являются *однонаправленными* (отражают семантику *объекта-агрегата*), тогда как *связи* характеризуют не состояние объекта, а участие его в том или ином отношении к другим объектам в смысле семантических связей задачи и всегда являются *двунаправленными.* Хотя синтаксическая реализация этих свойств в конкретных языках программирования и совпадает.

Связи могут существовать между экземплярами только двух типов (которые могут совпадать), т. е. *связи* являются бинарными. Свойства-связи, так же, как и атрибуты, объявляются в *ODL*-схеме БД. В объявление связей входит обязательное указание на обратное направление связи. Ниже приведен пример связи двух классов Professor и Course.

```
interface Professor {
       relationship   set <Course> teaches
                             inverse Course:: is_taught_by;
};
interface Course {
       relationship   Professor is_taught_by
                             inverse Professor:: teaches;
};
```

Как и атрибуты-ссылки, связи реализуются с помощью объектных ссылок (POID и OID) на связанный объект, массива ссылок, либо объектов-агрегатов таких ссылок. Связи по типу бывают *упорядоченные* и *неупорядоченные*, в зависимости от типа агрегата, представляющего связь. Например, Bag<Course> – неупорядоченная связь, тогда как List<Member> – упорядоченная. Заметим, что упорядоченные связи практически мало применяются для моделирования реальных прикладных задач. Связи также различаются по *мощности* (количеству элементов с обеих сторон) и бывают следующих типов: *1:1, 1:n, n:m.*

Механизм связей *ODMG* – *двунаправленный*. При этом СУБД обязана отслеживать корректность взаимных ссылок хранящихся в БД объектов согласно имеющейся схеме и обеспечивать целостность данных, в частности, удаление связей на объект со стороны других объектов при его удалении из БД (обеспечение целостности для атрибутов-ссылок не гарантируется).

Описанные в формате *ODL* связи – лишь абстрактная спецификация. Для организации нормальной работы и использования связей необходимо описать некоторые классы и методы, позволяющие устанавливать, видоизменять и удалять эти связи. Так *ODL* - описание связи (1:1):

```
relationship X inverse Z;
```

преобразуется в эквивалентное *IDL* – описание:

```
attribute X, Y;
void form_Y(in X target);
void drop_Y(in X target);
```

Аналогично, для связей мощности 1:n имеем:

```
relationship set<X> Y inverse Z;
```

На *IDL*:

```
attribute     set<X> Y;
void          form_Y (in X target)
              raises(lntegrityError);
void          drop_Y (in X target);
void          add_Y (in X target)
              raises(lntegrityError);
void          remove_Y (in X target);
```

2.1.7. Хранимые объекты (Persistent Objects)

Для приложения ООС является хранилищем перманентных (*хранимых) объектов*. При этом в конкретный момент времени такой объект может и не существовать в памяти программы. Но ООС знает уникальный идентификатор *(POID)* хранимого объекта и знает, как этот объект “синтезировать”, так что по запросу пользователя (т.е. по вызову соответствующих методов ООС) он может быть создан в памяти приложения и проинициализирован состоянием, хранимым в БД. Следует разделить понятия уникального *Объектного Идентификатора* в прикладной программе (*OID*) и *уникального Объектного Идентификатора Хранимого Объекта* в БД (*POID*). *OID* и *POID* представляют, вообще говоря, независимые пространства идентификаторов. При этом один и тот же хранимый объект, имеющий свой постоянный *POID* в БД, может обладать различными *OID* в прикладной программе при её запуске в разные моменты времени.

2.1.8. Экстент ОБД

Экстентом объектного типа называется совокупность всех хранимых объектов данного типа в рамках определенной БД. Если тип *B* наследуется от типа *A*, то экстент типа *B* является подмножеством экстента типа *A*. Экстенты имеют большое значение для работы целевого приложения. Они являются той базовой единицей, по которой производятся *OQL* запросы на получение хранимых в БД объектов.

2.1.9. Фундаментальные свойства объектно-ориентированной технологии

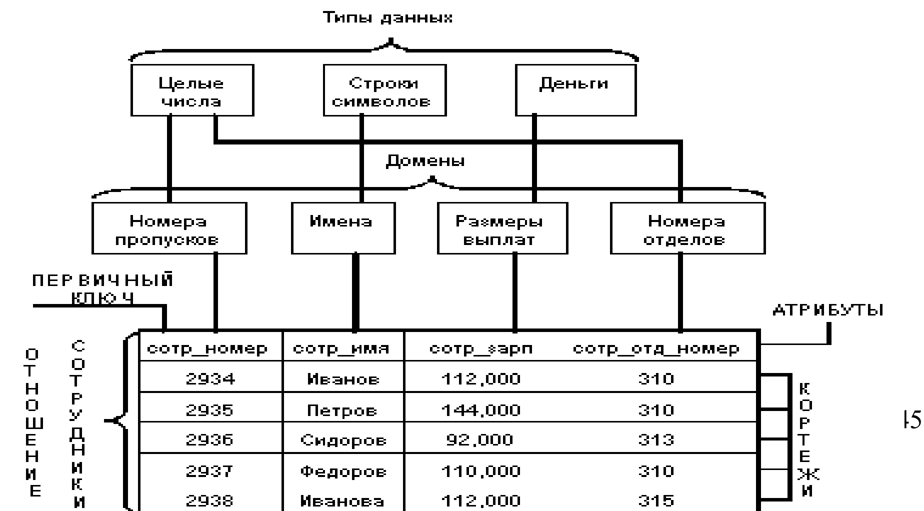
Заметим, что описанные понятия объектной модели отражают фундаментальные свойства объектно-ориентированной технологии:

- Инкапсуляция
- Наследование
- Полиморфизм

Данные свойства являются отличительными для ОО модели, и именно их отражение в реляционную модель вызывает наиболее принципиальные проблемы.

2.2. Реляционная модель

Рассмотрим теперь реляционную модель данных, предложенную изначально Коддом. Это необходимо для дальнейшего изложения. При рассмотрении используется информация и примеры из [2]. Основными понятиями реляционных баз данных являются *тип данных, домен, атрибут, кортеж, первичные и внешние ключи и отношение*. Для начала покажем смысл этих понятий на примере отношения СОТРУДНИКИ, содержащего информацию о сотрудниках некоторой организации:



2.2.1. Тип данных

Понятие *тип данных* в реляционной модели данных полностью адекватно понятию типа данных в языках программирования. Обычно в современных реляционных БД допускается хранение символьных, числовых данных, битовых строк, специализированных числовых данных (таких как "деньги"). А также специальных "темпоральных" данных (дата, время, временной интервал). Как уже замечалось, в объектно-реляционных системах активно развивается подход к расширению возможностей реляционных систем абстрактными типами данных. В нашем примере мы имеем дело с данными трех типов: строки символов, целые числа и "деньги".

2.2.2. Домен

Понятие *домена* более специфично для баз данных, хотя и имеет некоторые аналогии с подтипами в некоторых языках программирования. В самом общем виде домен определяется заданием некоторого базового типа данных, к которому относятся элементы домена, и произвольного логического выражения, применяемого к элементу типа данных. Если вычисление этого логического выражения дает результат "истина", то элемент данных является элементом домена. Наиболее правильной интуитивной трактовкой понятия домена является понимание домена как допустимого потенциального множества значений данного типа. Например, домен "Имена" в нашем примере определен на базовом типе строк символов, но в число его значений могут входить только те строки, которые могут изображать имя (в частности, такие строки не могут начинаться с мягкого знака). Следует отметить также семантическую нагрузку понятия домена: данные считаются сравнимыми только в том случае, когда они относятся к одному домену. В нашем примере значения доменов "Номера пропусков" и "Номера групп" относятся к типу целых чисел, но не являются сравнимыми. Заметим, что в большинстве реляционных СУБД понятие домена не используется, хотя в Oracle начиная с V.7 оно уже поддерживается.

2.2.3. Схема отношения, схема базы данных

Схема отношения - это именованное множество пар {имя *атрибута*, имя домена (или типа, если понятие домена не поддерживается)}. Степень или "арность" схемы отношения - мощность этого множества. Степень отношения СОТРУДНИКИ равна четырем, то есть оно является 4-арным. Если все атрибуты одного отношения определены на разных доменах, осмысленно использовать для именования атрибутов имена соответствующих доменов (не забывая, конечно, о том, что это является всего лишь удобным способом именования и не устраняет различия между понятиями домена и атрибута). *Схема РСУБД* (в структурном смысле) - это набор именованных схем отношений.

2.2.4. Кортеж, отношение

Кортеж, соответствующий данной схеме отношения, - это множество пар {имя атрибута, значение}, которое содержит одно вхождение каждого имени атрибута, принадлежащего схеме отношения. "Значение" является допустимым значением домена данного атрибута (или типа данных, если понятие домена не поддерживается). Тем самым, степень или "арность" кортежа, то есть число элементов в нем, совпадает с "арностью" соответствующей схемы отношения. Попросту говоря, кортеж - это набор именованных значений заданного типа. *Отношение* - это множество кортежей, соответствующих одной схеме отношения. Иногда, чтобы не путаться, говорят "отношение-схема" и "отношение-экземпляр", иногда схему отношения называют заголовком отношения, а отношение как набор кортежей - телом отношения. На самом деле, понятие схемы отношения ближе всего к понятию структурного типа данных в языках программирования. Было бы вполне логично разрешать отдельно определять схему отношения, а затем одно или несколько отношений с данной схемой. Однако в реляционных базах данных это не принято. Имя схемы отношения в таких базах данных всегда совпадает с именем соответствующего отношения-экземпляра. В классических реляционных базах данных после определения схемы базы данных изменяются только отношения-экземпляры. В них могут появляться новые и удаляться или модифицироваться существующие кортежи. Однако во многих реализациях допускается и изменение схемы базы данных: определение новых и изменение существующих схем отношения. Это принято называть *эволюцией схемы базы данных*.

2.2.5. Ключи

В реляционной терминологии используются два вида ключей – *первичные* (primary key) и *внешние* (foreign key). Первичный ключ отношения это один или несколько элементов схемы данного отношения (атрибутов), совокупность значений которых уникальна для всех кортежей отношений. Иными словами это идентификатор кортежа. Внешний ключ задает ограничение на значения атрибутов в одном отношении в соответствии со значениями первичного ключа другого отношения. То есть, если для отношения описан внешний ключ для некоторого атрибута, то в этом отношении не может быть кортежей со значением данного атрибута, если в исходной таблице нет кортежей с соответствующим значением первичного ключа.

2.2.6. Общепринятая реляционная терминология

Описанные выше понятия соответствуют названиям из реляционной теории, где они определяются абсолютно формально и точно. Однако в литературе часто используется другая терминология, имеющая очень простую интуитивную интерпретацию. Обычным житейским представлением

отношения является *таблица*, *заголовком* которой является *схема отношения*, а *строками* - *кортежи* отношения-экземпляра; в этом случае имена *атрибутов* именуют *столбцы* этой таблицы. Поэтому иногда говорят "столбец таблицы", имея в виду "атрибут отношения". Этой терминологии придерживаются в большинстве коммерческих реляционных СУБД. *Реляционная база данных* на языке введенных определений это набор отношений, имена которых совпадают с именами схем отношений в схеме БД.

2.3. Соответствие основных понятий

Рассмотрим, как принципиально возможно привести описанные выше объектную и реляционную модели к соответствию. Здесь описываются только общие правила и трудности, более подробный анализ различных вариантов отображения элементов объектной модели на реляционную схему проводится в данной главе ниже в соответствующих пунктах.

Первым ключевым моментом является представление данных конкретного экземпляра *объекта* в виде кортежа(ей) реляционного отношения(ий). То есть *состояние объекта* запоминается в виде набора значений атрибутов в одной или нескольких реляционных таблицах. *Экстент* объектной модели это заполненная таблица в РСУБД. Такая таблица в дальнейшем будет часто называться *таблицей класса*. При этом каждому *атрибуту* базового типа в описании класса соответствует один атрибут реляционной таблице. Ситуация осложняется для *агрегатных типов*. В данном случае можно, например, использовать отдельную таблицу для хранения элементов данного агрегата.

Каждый объект идентифицируется своим *уникальным объектным идентификатором (OID)*. Реально для этой цели можно использовать столбец типа длинного целого числа. Данный столбец будет выполнять функции первичного ключа в соответствующей таблице класса. А *связи* между объектами выражаются в виде реляционных *внешних ключей,* так же как и простые односторонние ссылки на другие объекты.

Таким образом, с использованием приведенных выше приемы становится возможным организовать хранение объектных данных в реляционной базе данных. Однако существует принципиальное несоответствие парадигм. Рассмотрим, как отражаются в РСУБД фундаментальные свойства объектно-ориентированной технологии:

- *Инкапсуляция*. Реляционные БД не поддерживают средств ограничения доступа, кроме паролей и пользователей. Однако прикладная программа обычно выполняется на одном уровне привилегий и поэтому все скрытые данные, хранящиеся в БД, становятся общедоступными, а попытки обеспечить инкапсуляцию путем введения различных представлений (views) с определенными правами доступа ведут к существенному усложнению схемы БД и программного кода.

- *Наследование*. Так как в РБД хранятся только данные объектов, то для корректного их воссоздания нужно хранить еще и информацию об их классах. Иерархия классов может быть представлена путем создания отдельных таблиц (*таблиц класса*) для каждого элемента этой иерархии. При этом для обеспечения корректного выполнения запросов данные объекта необходимо хранить и во всех таблицах родительских классов, и при обработке данного объекта надо следить за правильным использованием всех этих родительских таблиц.
- *Полиморфизм*. Поддержка данного механизма в РБД полностью отсутствует, и чтобы обеспечить корректную реконструкцию объектов, нужно следить за наличием достаточной дополнительной информации в таблицах класса. При этом большая нагрузка ложится на прикладного программиста по обеспечению правильного использования этой информации.

Итак, мы видим, что обеспечение базовых механизмов ОО модели в реляционной базе данных связано с рядом трудностей, связанных, прежде всего, с необходимостью решать задачу корректного отображения объектной иерархии на реляционную структуру, что при построении сложных информационных систем является очень нетривиальным делом. Далее проводится описание и анализ конкретных элементов решения данной задачи.

3. Приемы объектно-реляционного отображения

3.1. Идентификация объектов

Ключевым моментом в организации работы с объектами является их идентификация; при этом в рамках системы долговременного хранения различаются два вида идентификаторов: *Объектные Идентификаторы* в прикладной программе (*OID*) и *Уникальные Объектные Идентификаторы Хранимых Объектов* в БД (*POID*). Как уже отмечалось, каждый хранимый объект имеет свой постоянный *POID* в БД, который не совпадает с различными *OID* в прикладной программе при её запуске в разные моменты времени. С точки зрения рассматриваемой системы наибольший интерес представляет POID. Реализация этого элемента в приложении представляет собой отдельный класс, а в базе данных в виде набора столбцов таблицы класса. Целью POID является обеспечение возможности однозначно идентифицировать объект в приложении и его данные в реляционной таблице. При этом в реляционных таблицах POID играет роль не только ключа в таблице данного класса; он также используется в качестве ссылки на эти объекты со стороны других классов в соответствующих таблицах.

Существует несколько подходов к организации объектных идентификаторов. Главным выводом при анализе этих методик является то, что POID не должен быть связан с семантикой конкретного класса. Это ограничение связано с тем,

что содержание или структура бизнес-смысла может меняться, что влечет большую работу по переделке существующей схемы и содержащихся в ней данных. Кроме того, наличие унифицированной структуры POID для всех классов позволяет упростить код, обрабатывающий объектно-реляционное отображение.

В настоящее время общепризнанным подходом является организация POID в виде пары целых чисел – идентификатора класса объекта и идентификатора экземпляра самого объекта в рамках данного класса. То есть класс POID реализуется в следующем виде:

```
class POID {
      private int cid; // идентификатор класса
      private int poid; // идентификатор объекта внутри
      класса
      // методы доступа
      . . .
}
```

Такой подход позволяет вместе с решением основной задачи по уникальной идентификации объекта решать еще одну важную задачу – определять класс объекта. Среди схожих подходов можно упомянуть урезанные варианты данного подхода, а именно, когда уникальность идентификатора сохраняется только внутри класса (отсутствует поле cid), и когда идентификаторы уникальны внутри иерархии наследуемых классов (вместо cid имеется iid). Однако эти подходы усложняют работу программистов по отслеживанию класса объекта и, кроме того, при изменении дерева наследования приходится существенно переделывать уже существующие прикладные программы.

Следующим важнейшим вопросом организации объектной идентификации является генерация новых идентификаторов во время сохранения новых объектов в БД. В следующих пунктах описываются различные стратегии решения данной задачи. При этом главным аспектом является определение именно части poid, так как информация о cid является статической (по крайней мере, на время сеанса работы приложения), и проблем с ее получением не возникает.

3.1.1. Использование SQL функции MAX

Простейшим вариантом определения следующего значения объектного идентификатора является использование SQL функции MAX() на столбце poid соответствующей таблицы класса. То есть фактически при добавлении записи в таблицу класса вычисляется максимальное значение poid среди уже существующих записей и столбцу poid новой присваивается значение на единицу больше полученного. Проблема с использованием этого способа заключается в том, что возникает необходимость в выполнении большого количества SQL запросов, каждый из которых вызывает блокирование таблицы

и при наличии большого количества записей сам по себе занимает много времени на вычисление максимального значения идентификатора.

3.1.2. Использование служебной таблицы

Для устранения последнего недостатка предыдущего способа можно определить специальную служебную таблицу, имеющую два столбца – идентификатор класса и идентификатор объекта. В этой таблице должны храниться последние значения используемых идентификаторов. Сценарий присвоения нового идентификатора в данном случае выглядит следующим образом:

1. Таблица блокируется для доступа других пользователей
2. Выбирается значение идентификатора для соответствующего класса
3. Вместо него записывается значение на единицу большее
4. Блокировка снимается

Использование этого механизма обеспечивает независимость присвоения идентификатора от количества записей в таблице конкретного класса. Также унифицируется процедура получения этого идентификатора, нет необходимости в построении динамических SQL запросов с различными именами таблиц. Однако при интенсивной работе различных пользователей доступ к этой системной таблице может стать узким местом в производительности, из-за наличия блокировки во время выполнения транзакции присвоения нового poid.

3.1.3. Использование механизмов автоматических счетчиков РСУБД

Большинство существующих реляционных СУБД поддерживают так называемые автоматические счетчики для значений столбцов. Реализация этих механизмов может быть различной. Например, в MS Access для поля существует отдельный тип – *Counter*, а в Oracle для этой цели используется отдельный механизм *Sequence,* и значения в целевую таблицу прописываются с помощью триггеров. То есть для приложения отпадает необходимость в отслеживании уникальности poid, эта задача полностью ложится на сервер.

Однако данный подход имеет существенный недостаток. Механизм присвоения идентификаторов зависит от конкретного производителя РСУБД. Вследствие этого создаваемое приложение, во-первых, становится привязанным к конкретной РСУБД, а во-вторых, процесс присвоения идентификаторов находится вне контроля прикладного программиста, в частности, возникают проблемы с получением только что занесенного значения идентификатора, что необходимо для полноценной реализации сервиса генерации Poid (то есть объектов класса POID).

3.1.4. Использование хеш-функций

Принципиальным моментом описываемых далее методов является увеличение производительности за счет минимизации общения с сервером и переноса алгоритмов генерации poid на сторону клиента.

Наиболее естественным решением является использование специальных хэш-функций, которые используют уникальные аппаратные идентификаторы конкретной машины в комбинации с текущим временем. Такие технологии реализованы, например, в GUID компании Microsoft и в UUID компании Digital. Таким образом, уникальность значений достигается на стороне клиента, однако за это приходится расплачиваться потенциальной емкостью пространства идентификаторов. В этом случае для poid нельзя использовать простой int, нужно число гораздо большей емкости (в существующих реализациях используются 128-битные значения), так как между двумя последовательно создаваемыми идентификаторами имеется достаточно большой неиспользуемый промежуток. Кроме того, использование аппаратных данных не всегда доступно на различных платформах и создаваемые приложения нельзя сделать полностью переносимыми.

3.1.5. Метод сложного двухкомпонентного идентификатора

Главной идеей данного метода является разбиение poid на две логические части: серверную и клиентскую. Серверная часть идентификатора (HIGH value) поддерживается с помощью одного из изложенных методов. В качестве оптимального способа можно использовать специальную служебную таблицу. Как уже отмечалось, недостаток этого способа состоит в частом обращении к серверу для извлечения следующего значения, что может снижать производительность.

В предлагаемом методе обращение к серверу происходит очень редко, так как необходимость в извлечении серверного значения возникает только при переполнении клиентского счетчика, а фактически только при начальной загрузке приложения. Для генерации же очередного клиентского значения используется тривиальный последовательный счетчик с поддержкой соответствующей переменной в памяти приложения. Таким образом, работа метода выглядит следующим образом:

1. При загрузке приложения с сервера извлекается текущее серверное значение в соответствии с алгоритмом использования служебной таблице (см. выше). Данное значение записывается в соответствующей переменной в памяти приложения.
2. Для получения очередного идентификатора poid полученное серверное значение просто объединяется с получаемым локально клиентским значением (изначально оно равно 0). Эта процедура занимает минимум времени, но при этом обеспечивает гарантированную уникальность получаемого poid.

3. При переполнении клиентского счетчика (что происходит только при очень интенсивном создании новых объектов в текущем сеансе работы) с сервера извлекается новое значение серверной части и процедура сохраняется.

Среди рассмотренных здесь методов генерации объектных идентификаторов метод сложного двухкомпонентного идентификатора является наилучшим с точки зрения обеспечения производительности, при этом сохраняется простота реализации. Другие способы организации объектной идентификации можно посмотреть в [12] - [14].

3.2. Хранение отдельного атрибута

В данном пункте рассматривается задача хранения отдельного атрибута класса в реляционной таблице. Выше упоминалось, что базовой концепцией в решении данной проблемы является хранение значений атрибутов класса в столбцах реляционных таблиц. Ниже проводится обзор различных способов организации отображения атрибута на реляционный столбец, вернее, ситуации, которые описывает прикладной программист при задании объектно-реляционного отображения атрибутов. Описание некоторых приведенных здесь методик и их реализация хорошо представлены в продукте TopLink for Java компании ObjectPeople (см. [15]). Выбор конкретного способа индивидуален для каждого атрибута и зависит от семантики и характера используемых объектных и реляционных структур.

3.2.1. Доступ к значениям атрибутов

Чтобы сохранять и восстанавливать значения атрибутов конкретного экземпляра класса, необходимо обеспечить доступ к данным этих атрибутов для сервиса долговременного хранения объектов. Трудность заключается в инкапсуляции данных объекта, что запрещает доступ к частным (private) и защищенным (protected) атрибутам со стороны других непривилегированных классов приложения. Поэтому для виртуальных машин Java до версии 1.2 существовал единственный способ обеспечить доступ к нужным данным: посредством описания специальных методов доступа (accessor methods):

```
class Person {
        private String name;

        // методы доступа
        public String getName() {
                return name;
        }
        public void setName(String name) {
                this.name = name;
        }
};
```


В новой версии виртуальной машины Java появились специальные возможности по предоставлению при определенных условиях доступа к частным и защищенным атрибутам со стороны других классов, в нашем случае со стороны сервиса долговременного хранения. Данная технология носит названия *Reflection*. Подробнее о ней см. документацию по Java [16].

Таким образом, для всех приводимых ниже способов при описании отображения для каждого атрибута существует альтернатива выбора между двумя методами доступа (первый применим только для общедоступных (public) атрибутов в Java 1.1 и любых в Java 2):

- Прямой доступ к атрибутам
- Доступ с помощью специальных методов

Выбор нужного остается на усмотрение прикладного программиста и зависит от принятых в компании соглашений по написанию кода или от структуры уже существующих Java классов.

3.2.2. Прямое отображение атрибутов элементарных типов

Простейшим способом организации отображения атрибутов объекта на реляционные столбцы является отношение один к одному (*direct to field mapping*). То есть для хранения данных определенного атрибута класса используется определенный столбец реляционной таблицы, при этом не требуется никаких преобразований, и данные передаются непосредственными вызовами соответствующих методов JDBC. Основным моментом в этом случае является установление соответствия типов данных атрибутов приложения и типов данных реляционных столбцов. Здесь имеются в виду атрибуты элементарных типов, атрибуты-коллекции и атрибуты-ссылки на другие объекты рассматриваются в следующих пунктах. В рамках разрабатываемой нами системы ODESTOR соответствие базовых типов выглядит следующим образом:

Таблица соответствия типов данных ODL, Java и SQL разных производителей				
ODL	**Java**	**ANSI SQL**	**Oracle**	**Sybase**
Базовые типы				
Short	Short	SMALLINT	NUMBER(5)	SMALLINT
Unsigned short	Int	INTEGER	NUMBER(10)	INTEGER
Long	Int	INTEGER	NUMBER(10)	INTEGER
Unsigned long	Long	DECIMAL(19)	NUMBER(19)	BIGINT
Float	Float	REAL	NUMBER	FLOAT
Double	Double	DOUBLE PRECISION	NUMBER	DOUBLE
Char	Char	CHAR(1)	CHAR(1)	CHAR(1)
Boolean	Boolean	DECIMAL(1)	NUMBER(1)	BIT
Octet	Byte	SMALLINT	NUMBER(3)	TINYINT

Таблица соответствия типов данных ODL, Java и SQL разных производителей				
ODL	**Java**	**ANSI SQL**	**Oracle**	**Sybase**
String	String	VARCHAR	VARCHAR2(255)	VARCHAR(255)
Date	java.sql.Date	DATE	DATE	DATE
Time	java.sql.Time	TIME	DATE	TIME
Timestamp	java.sql.TimeStamp	TIMESTAMP	DATE	TIMESTAMP

3.2.3. Отображение с преобразованием типа

Прямое отображение атрибута в столбец реляционной таблицы может быть невозможно ввиду отличия типов данных (в терминах приведенной выше таблицы). В таком случае описывается *отображение с преобразованием типа.* То есть при указании правил отображения таких атрибутов указывается исходный и целевой тип в терминах типов Java. И тогда при работе с этими атрибутами происходит двухэтапное преобразование: к прямому отображению добавляется обычное преобразование типов Java.

3.2.4. Отображение с конечным набором значений

Данный тип отображения используется для преобразования типов, имеющих семантически фиксированное число значений. Дело в том, что в реляционных таблицах для экономии памяти во многих случаях используются специальные сокращения. Например, для обозначения пола человека в реляционной таблице может быть определен столбец типа char(1), в котором бывают только два значения: “M”(Male) и “F”(Female). В классе же при этом имеется атрибут типа String, в котором должны быть полные значения: “Male” и “Female”. В данном случае программист описывает специальную таблицу преобразования, по которой система долговременного хранения затем автоматически строит нужные методы трансформации.

3.2.5. Преобразование в бинарный поток

Этот тип целесообразно использовать для сохранения атрибутов, содержащих данные сложной структуры и/или имеющих большой размер. Но при этом это внутреннее содержание не должно использоваться на детальном уровне в запросах к объектно-реляционной прослойке, то есть данный способ подходит для хранения «листовых» атрибутов, не участвующих в OQL запросах. Физически данное преобразование осуществляется с использованием механизма преобразования данных объекта в бинарный поток (serialization, подробнее см. выше в разделе описания подходов к хранению данных

объектов). Этот бинарный поток сохраняется в столбце типа VARBINARY или LONGVARBINARY.

3.2.6. Отображение со сложной трансформацией

Сложная трансформация данных используется в наиболее общих случаях, когда пользователь сам хочет описать собственные преобразующие методы. При этом происходит преобразование одного или нескольких атрибутов класса в несколько столбцов таблицы. В частном случае несколько столбцов реляционной таблицы используются для хранения данных одного атрибута класса. Например, в старых базах данных дата и время часто хранятся в отдельных столбцах, и необходимо использовать сложную трансформацию для отображения этих столбцов в атрибут типа java.util.Date. То есть можно использовать атрибуты не только базовых типов.

Технически при описании данного типа отображения прикладной программист задает два набора методов класса. Один из них отвечает за чтение данных из предоставляемой в качестве параметра реляционной строки (JDBC ResultSet); каждому атрибуту соответствует отдельный метод. Другой набор методов служит для извлечения данных из атрибутов класса, и число этих методов соответствует числу реляционных столбцов, участвующих в отображении.

3.3. Хранение коллекций и массивов

В предыдущем пункте рассматривались аспекты работы с атрибутами базовых типов (за исключением общего случая *отображения со сложной трансформацией* и *отображения с преобразованием в бинарный поток*), но описанные способы не подходят для хранения атрибутов, содержащих коллекции этих базовых типов, то есть массивов и специальных коллекционных типов, описанных в объектной модели. Коллекции *объектов* рассматриваются отдельно в пункте *Хранение связей между объектами*. Здесь же анализируется организация хранения массивов примитивных типов.

3.3.1. Преобразование в бинарный поток

Для массивов, если в запросах к БД не используется навигация по конкретным элементам, вполне целесообразно использовать уже рассмотренный выше способ преобразования в двоичный поток с помощью технологии сериализации в Java. Этот способ в особеннсости рекомендуется для массивов byte[].

3.3.2. Использование таблиц для каждого типа

Использование преобразования в двоичный поток невозможно, если внутренние данные массива используются для запросов к базе данных. Кроме того, этот способ неудобен для коллекций элементарных типов, в связи с

необходимостью преобразования в объект, поддерживающий интерфейс Serializable. Решением для хранения внутренних элементов массива может служить образование отдельных таблиц для массивов *каждого* из элементарных типов. То есть массивы элементарных типов рассматриваются как отдельные типы данных, и для каждого из них производится соответствующее построение нужной реляционной таблицы следующего вида:

CID	POID	AID	SEQ	VALUE

Здесь CID и POID идентифицируют исходный объект, к которому относятся данные, AID обозначает идентификатор атрибута-массива в исходном классе (статическая информация), SEQ используется для идентификации порядка элементов в упорядоченных массивах. Столбец VALUE имеет тип, соответствующий базовому типу Java, для которого предназначается данная таблица. Недостаток данного способа состоит в перегруженности таблицы для систем с большим количеством данных, что отрицательно сказывается на производительности. Но зато в базе данных имеется малое количество таблиц, не загромождается схема РСУБД.

3.3.3. Использование таблиц для каждого атрибута

Для устранения узкого места в таблицах коллекционных типов можно использовать отдельные таблицы для каждого атрибута массива. При этом экономится место на исключении столбца AID, и таблица для хранения данных атрибута-массива выглядит следующим образом:

CID	POID	SEQ	VALUE

Однако использование данного способа при наличии сложной схемы базы данных может существенно загромождать структуру БД, усложнять семантику обработки рассматриваемых атрибутов. Реально прикладной программист должен иметь возможность выбора типа отображения конкретного атрибута-массива при задании общего объектно-реляционного отображения. Для больших по объему данных атрибутов рекомендуется использовать отдельные таблицы, а все мелкие атрибуты хранить в соответствующей таблице типа.

3.4. Хранение ссылок и связей между объектами

До сих пор шла речь в основном о хранении атрибутов, содержащих данные элементарных базовых типов. При отображении объектных ссылок и связей (в терминах описанной объектной модели) возникают определенные нюансы. Во-первых, сама парадигма объектной базы данных обеспечивает навигацию по внутренней структуре объектов, поэтому простейший рассмотренный тип преобразования в двоичный поток для хранения связей и объектных ссылок не

подходит. Во-вторых, для связей нужно обеспечить *двустороннюю* способность к навигации.

Преобразование в двоичный поток может использоваться только тогда, когда программист уверен, что объектная ссылка является листовой и не участвует в OQL запросах. Данная ситуация, например, возникает, когда атрибут описывается как *структура (struct),* то есть формально в Java он является объектной ссылкой, но семантически данная структура не является объектом, у нее нет *объектного идентификатора,* и на нее ссылается только *один* объект. Данный вид отношения называется *агрегацией*. При этом, кроме способа преобразования в двоичный поток, такой атрибут можно хранить с помощью ***метода отображения с агрегацией***. При таком способе внутренние данные атрибута-структуры разворачиваются и хранятся в таблице исходного класса таким образом, как если бы они были обычными атрибутами этого класса.

Как уже отмечалось, ключевым моментом в отображении объектов на реляционные таблицы является представление ссылки на любой объект в виде пары целых чисел (CID, POID). Таким образом, атрибут, являющийся односторонней ссылкой, представляется в РСУБД парой столбцов, в которые записываются соответствующие идентификаторы нужного объекта-цели. Это так называемый *метод внедренного ключа*.

Аналогично решается проблема отображения связей ***один-к-одному и один-ко-многим***. В этом случае атрибуты-ссылки помещаются в обоих классах, участвующих в связи. В случае связи один-ко-многим и наличии упорядочивания элементов связи, кроме столбцов-идентификаторов в классе, отвечающем за множественную часть связи, вводится дополнительный атрибут SEQ, в котором хранится информация о порядке элементов. В результате система долговременного хранения в состоянии однозначно определить все стороны связи. Для связей один-к-одному также рекомендуется *способ объединения в одну таблицу*, когда данные двух классов хранятся в одной таблице, это обеспечивает очень высокую производительность.

Ситуация осложняется в случае связей ***много-ко-многим***. Здесь простым включением столбцов-идентификаторов в таблицу класса обойтись нельзя. Выходом из положения является создание для каждой связи отдельной *таблицы связи*. В данной таблице содержится в общем случае 6 столбцов:

CID1	POID1	SEQ1	CID2	POID2	SEQ2

Четыре из этих столбцов (CID1, POID1, CID2, POID2) отвечают за идентификацию объектов, участвующих в связи, а два (SEQ1, SEQ2) служат для определения порядка элементов в случае наличия упорядоченности на одной или двух сторонах связи. Заметим, что для общности можно использовать способ отдельных таблиц также и для связей типа *один-к-одному и один-ко-многим.* При этом в случае *один-к-одному* каждая пара столбцов

(CID, POID) является уникальной, а в случае *один-ко-многим* уникальной будет пара столбцов, соответствующих единичной стороне связи.

Общим правилом для отражения связей межу объектами в реляционной базе данных является описание реляционных внешних ключей (foreign key) для обеспечения целостности данных. Поэтому поля (CID, POID) в каждой из таблиц связи и атрибуты-связи в таблицах класса объявляются внешними ключами по отношению к соответствующим таблицам класса-цели.

3.5. Отображение одного класса

Итак, в предыдущих пунктах описаны способы организации хранения отдельных атрибутов различных типов. Теперь рассмотрим, как в РСУБД обеспечивается хранение объектов класса. В соответствии с общими принципами объектно-реляционного отображения понятию класса в объектной модели соответствует понятие таблицы (отношения) в реляционной модели.

Однако здесь существуют устоявшиеся стратегии отражения класса на реляционные таблицы. Простейшим способом организации этого отражения является классическое ***табличное отображение (TABLE mapping)***. В этом случае *все* (в том числе и все унаследованные) атрибуты одного класса хранятся в отдельной таблице, и в этой таблице нет лишних атрибутов, не относящихся к рассматриваемому классу.

Следующим типом отображения класса является ***метод подмножества (SUBSET mapping)***. Здесь для хранения данных определенного класса в реляционной таблице используется только часть полей (но при этом в таблице по-прежнему содержатся поля для хранения всех атрибутов данного класса). То есть в таблице имеется избыточное число полей, и для конкретного класса используется их подмножество. Такая ситуация часто возникает при работе с унаследованными базами данных.

И, наконец, последний случай -- это ***метод надмножества (SUPERSET mapping)***. В отдельной таблице СУБД хранится только часть необходимой для данного класса информации, то есть данные класса хранятся в нескольких таблицах. Для скрытия этой внутренней структуры таблиц часто используются реляционные представления (view).

Подробнее использование приведенной классификации рассматривается ниже, также см. [11], [17] - [25].

3.6. Отображение дерева наследования

Основным моментом в построении схемы объектно-реляционного отображения является выбор стратегии отражения в РСУБД иерархии наследования классов. Проблема состоит в решении, в каких таблицах хранить атрибуты каждого класса иерархии. В различных реализациях существующих продуктов используется три варианта указанной стратегии.

3.6.1. Отображение с фильтрацией (filtered mapping)

Метод заключается в поддержке одной глобальной таблицы (*таблица иерархии*) для каждой иерархии классов. Таблица строится следующим образом. Сначала в таблицу попадают все атрибуты корневого класса, а затем для каждого нового наследника в таблицу добавляются специфические для этого наследника атрибуты. Так продолжается до классов-листьев. В результате в таблице иерархии содержатся все возможные атрибуты для хранения экземпляра любого класса, принадлежащего этой иерархии. Это частный случай *метода подмножества*.

Данный вид отображения обеспечивает максимальную производительность работы приложений и простоту реализации сервиса долговременного хранения объектов. При изменении роли объекта в иерархии нет необходимости в изменении структуры таблиц, полиморфизм обеспечивается автоматически, запросы по извлечению данных также очень простые. К недостаткам относится необходимость изменять таблицу при добавлении новых атрибутов в *любом* из классов потомков, такая же проблема при добавлении новых потомков. Кроме того, за счет того, что каждый из объектов в таблице иерархии использует только часть столбцов для хранения своих данных, происходит излишняя трата дискового пространства.

3.6.2. Горизонтальное отображение (horizontal mapping)

Для устранения проблемы неиспользуемого пространства можно разбить глобальную таблицу иерархии на несколько таблиц. При этом в каждой такой таблице будут находиться все атрибуты конкретного класса, в том числе и унаследованные. Данная ситуация соответствует методу *табличного отображения*.

Использование метода горизонтального отображения обеспечивает высокую эффективность и простоту запросов по манипулированию данными. Однако ему присущи определенные недостатки. Во-первых, при модификации класса, у которого есть потомки, необходимо модифицировать и все таблицы этих потомков. Во-вторых, изменение места объекта в иерархии классов влечет за собой необходимость в перемещении данных из одной таблицы в другую и присваивании нового объектного идентификатора. В-третьих, поддержка полиморфизма затруднительна ввиду необходимости обработки нескольких таблиц для извлечения нужных данных.

3.6.3. Вертикальное отображение (vertical mapping)

Данный вид отображения отражает иерархию наследования наиболее естественным образом. Как и при горизонтальном отображении для каждого класса определяется отдельная таблица, но в этой таблице хранятся только атрибуты, *специфические* для данного класса. Между таблицами в иерархии

определяются внешние ключи, по которым в дальнейшем происходит соединение таблиц (join) для манипулирования содержащимися в них данными. Рассматриваемый способ является частным случаем *метода надмножества*.

Вертикальное отображение очень хорошо поддерживает механизм полиморфизма. При извлечении данных определенного типа в таблице хранятся данные не только данного класса, но и всех потомков. Модификация классов, входящих в иерархию, теперь происходит с наименьшими усилиями – нужно изменить только одну таблицу. К недостаткам способа относится сложные механизмы манипулирования данными объектов. Для чтения всех требуемых данных необходимо обращаться к нескольким таблицам вплоть до корневой. Проблему можно частично решить введением реляционных представлений для имитации таблиц класса, как при использовании горизонтального отображения.

3.7. Замечания по производительности и оптимизации

Одним из основных требований практически к любому приложению является высокая производительность. Поэтому вопросы оптимизации работы сервиса долговременного хранения является важнейшим аспектом при построении таких систем. В данном пункте приводятся основные выводы и фундаментальные принципы обеспечения высокой скорости объектно-реляционных преобразований. Способы повышения производительности можно разделить на два принципиальных раздела. Первый это повышение эффективности работы реляционного сервера, а второй это оптимизация запросов к серверу со стороны клиента, в частности, минимизация сетевого трафика и числа обращений к серверу (подробнее см. [26]).

3.7.1. Оптимизация сервера

Одним из самых важных вопросов является правильный выбор одного из вариантов объектно-реляционного отображения в каждом конкретном случае, понимая, что всегда существует баланс между производительностью и гибкостью способа. В этом случае можно дать следующие рекомендации. При отображении дерева наследования наиболее производительными являются варианты *отображения с фильтрацией* и *горизонтального отображения*. Метод *вертикального отображения*, будучи наиболее очевидным и соответствующим реальной объектной схеме, может значительно снижать скорость работы. По возможности нужно скрывать нормализованные реляционные таблицы с помощью использования реляционных представлений. При отображении связей нужно избегать отдельных *таблиц связи* и применять их только для связей типа много-ко-многим, когда они необходимы. Для связей один-к-одному настоятельно рекомендуется способ *объединения в одну*

таблицу. Для связей один-ко-многим следует применять *метод внедренных ключей*.

При проектировании схемы базы данных необходимо также выполнить классические настройки сервера. Основным аспектом здесь является определение индексов и внешних ключей. Индексы следует определять для столбцов, которые часто используются в условиях запросов, и для столбцов внешних ключей. Для достижения максимального результата можно использовать особенности конкретного реляционного сервера, в частности, хранимые процедуры, асинхронные и пакетные запросы. Также нужно размещать таблицы, к которым осуществляется одновременный доступ, на физически разных пластинах жесткого диска или вообще на разных дисках.

3.7.2. Оптимизация клиента

При оптимизации клиентской части сервиса долговременного хранения главной задачей является минимизация числа обращений к серверу. Достигается это с помощью механизма кэширования. То есть при чтении данных объекта они сохраняются на стороне клиента в специальной системной области сервиса долговременного хранения и при повторных обращениях данные берутся уже из этой области. Кроме того, существенным моментом в организации кэширования является создание в памяти приложения виртуального образа часто используемых объектов. При этом кардинальное ускорение происходит не только при чтении отдельного объекта, но и при выполнении навигации по объектным связям.

Авторы [26] провели исследование производительности различных ОО приложений, использующих в качестве хранилища РСУБД. По результатам этих тестов утверждается, что хорошо спроектированное ОО приложение, использующее объектно-реляционное отображение, обладая всеми преимуществами использования ООБД, не только не уступает в производительности приложению, написанному на основе API конкретной РСУБД, но и превосходит его за счет использования клиентского кэширования объектных данных. Поэтому вопросы оптимизации объектно-реляционных сервисов являются действительно очень важными.

4. Заключение

Результаты проведенных в данной работе исследований были успешно применены при работе над проектом реализации системы ODESTOR. Данная система относится к классу объектно-реляционных систем, предоставляющих интерфейс чисто объектной БД над реляционной СУБД. Такие системы по-прежнему очень актуальны по причине наличия большого числа унаследованных реляционных баз данных. Разными коллективами производились исследования производительности систем этого класса. В итоге основным выводом стало, что главным аспектом обеспечения высокой

производительности является именно грамотная и настроенная под конкретные нужды архитектура объектно-реляционного отображения. При этом хорошо спроектированные системы ОР отображения обеспечивают даже бóльшую производительность, чем классические решения на основе использования библиотек низкого уровня. Этот вывод строится на двух аспектах, во-первых, хорошо построенная система ОР отображения проигрывает лишь при непосредственных операциях обмена данными, а во-вторых, за счет использования централизованного управления кэшированием объектных данных на уровне приложения во многих случаях получается порядковый рост производительности. В итоге в реальных условиях система ОР отображения показывает в среднем бóльшую производительность. В связи с этим проведенные в данной работе исследования оптимальной схемы ОР отображения приобретают дополнительную актуальность.

Литература

1. “Object-Oriented Design”. G. Booch. 1992, Behjamin/Cummings Publishing Company.
2. “Основы современных баз данных”. Кузнецов С.Д. Информационно-аналитические материалы Центра Информационных Технологий. www.citforum.ru.
3. “Тенденции в мире систем управления базами данных”. Кузнецов С.Д. Информационно-аналитические материалы Центра Информационных Технологий. www.citforum.ru.
4. “Object Database vs. Object-Relational Databases”. Steve McClure. IDC Bulletin. www.cai.com/products/jasmine/analyst/idc/14821E.htm.
5. “Extending Relational Databases”. Martin Rennhackkamp. DBMS Magazine, December 1997
6. “Why Use an ODBMS?”. Poet Software White Paper. www.poet.com/products/oss/white_papers/rel_vs_obj/rel_vs_obj.html
7. “Объектно-ориентированные базы данных: среда разработки программ плюс хранилище объектов”. Андреев А.М., Березкин Д.В., Кантонистов Ю.А. www.inteltec.ru/publish/russian/articles/objtech/oodbms_o.htm
8. “Манифест систем объектно-ориентированных баз данных”. М. Аткинсон, Ф. Бансилон, Д. ДеВитт, К. Диттрих, Д. Майер, С. Здоник. Системы Управления Базами Данных, # 4/95, стр. 142-155.
9. .“The Object Database Standard: ODMG 2.0”. R.G.G. Cattell, Douglas K. Barry, 1997. www.odmg.org
10. .“The Object Data Standard: ODMG 3.0”. R.G.G. Cattell, Douglas K. Barry, 2000. www.odmg.org
11. “Mapping Objects To Relational Databases”. Scott W. Ambler. AmbySoft White Paper. http://www.ambysoft.com/mappingObjects.pdf
12. “Object Identity”. Khoshafian S. , Copeland G. OOPSLA Conference, 1986
13. “A rigorous model of object reference, identity and existence”. Kent W. JOOP June, 1991.
14. “Storage management for persistent complex objects”. Khoshafian, Franklin, Carey. Information Systems, vol.15, no.3, p. 303-320, 1990.
15. “TopLink for Java Documentation”. Object People. www.objectpeople.com/toplink/java/java.htm
16. “Java 2 Platform, Standard Edition Documentation”. java.sun.com/docs/index.html
17. “Object Relational Mapping Strategies”. Visual Business Sight Framework documentation. Mapping tool guide. www.objectmatter.com
18. “Foundations of Object Relational Mapping”. Mark L. Fussell. www.chimu.com
19. “Object-relational Access Layers – a Roadmap, Missing Links and More Patterns”. Wolfgang Keller. EPLoP’98.

20. “Patterns for Object/Relational Access Layers”. Wolfgang Keller. www.objectarchitects.de/ObjectArchitects/orpatterns/index.htm
21. “Integrating Objects with RDBMs”. GemStone White Paper. www.gemstone.com/products/s/papers_integrate.html
22. “Crossing Chasms: A Pattern Language for Object-RDBMS Integration (The Static Patterns)”. Kyle Brown and Bruce G. Whitenack. www.ksccary.com/Articles/ObjectRDBMSPattern/ObjectRDBMSPattern.htm
23. “A Pattern Language for Relational Databases and Smalltalk”. Kyle Brown and Bruce Whitenack. www.ksccary.com/Articles/PatternLangForRelationalDB/PatternLangForRelationalDB.htm
24. “Baker’s dozen”. Tim Ottinger. www.oma.com/ottinger/BakersDozen.html
25. “Cetus Links on Objects and Components: Object Relational”. WWW links collection. http://www.cetus-links.org/oo_db_systems_3.html
26. “Architecting Object Applications for High Performance with Relational Databases”. S. Agarwal, C. Keene, A.M. Keller. Persistence Software White Paper.
27. “Проблемы организации объектно-ориентированного доступа к реляционным базам данных”. К.В. Антипин, В.В. Рубанов. Труды Института Системного Программирования (сдано в печать), вып. 2. 2000 г.
28. “Объектно-ориентированное окружение, обеспечивающее доступ к реляционным СУБД”. В.П. Иванников, С.С. Гайсарян, К.В. Антипин, В.В. Рубанов. Труды Института Системного Программирования (сдано в печать), вып. 2. 2000 г.