Объектное представление о реляционной модели

Введение

Цель данной статьи показать преимущества интеграции объектно-ориентированной технологии и реляционной модели данных. Слияние этих ведущих направлений открывает новые возможности как в процессе проектирования баз данных, так и на стадиях эксплуатации и модернизации.

Современные программные системы становятся сложнее, претендуя на решение глобальных задач, например таких, как создание единой системы управления предприятием. При этом автоматизация отдельных операций или отделов фактически исчерпала свой потенциал, а возможность безболезненного объединения нескольких подсистем в единое целое, как правило, вызывает сомнение. Реальная проблема заключается в том, что связи между подсистемами должны быть гибкими, изменяемыми и настраиваемыми, но используемый инструментарий, в виде современных реляционных систем управления базами данных (РСУБД), не обладает этими свойствами. Данное положение усугубляется требованиями бизнеса, который нуждается в высоком динамизме. Централизованное управление предприятиями постепенно сменяется схемами распределённого управления, где значительная часть вопросов, требующих принятия решений, переходят на уровень структурных подразделений. В результате те связи, которые работали десятилетиями, могут быть мгновенно разрушены, например, какое-то подразделение может отказаться от использования услуг централизованной бухгалтерии, посчитав, что это достаточно неэффективно и дорого.

Проектирование сложных систем в таких условиях требует новых подходов и иных инструментальных средств. Однако не следует сбрасывать со счетов те объёмы информации, которые сегодня хранятся в реляционных базах данных, и тот опыт работы, который имеют специалисты, проектирующие и обслуживающие эти базы данных. Следовательно, весьма желательно, чтобы новые инструментальные средства и технологии не создавались с нуля, а основывались на способах и средствах, существующих к настоящему моменту. С этой точки зрения, необходимость перехода к так называемым “постреляционным” системам хранения информации, может быть не обоснована.

Применение методологии объектно-ориентированного анализа и проектирования к реляционным базам данных - это очень большая и сложная тема. В рамках этой статьи будет рассмотрен небольшой пример, который поможет составить представление о том, как ООП позволяет сделать любую систему более гибкой и динамичной, исключив необходимость в постоянном переписывании структуры базы данных и приложений. Объектный подход открывает здесь прекрасные возможности. Принято считать, что главное достоинство объектно-ориентированного проектирования заключается в увеличении коэффициента повторно используемого кода. Безусловно, это важно, но не менее важно и то, что объектные системы несут в себе возможность модификации и развития. Применительно к базам данных, это положение позволяет начать проектирование будущей системы, не имея исчерпывающего представления о предметной области. Поскольку получение детальной информации о предметной области - процесс весьма трудоёмкий, то можно надеяться на сокращение сроков разработки систем, а, следовательно, и их стоимости.

1. Пример

Предположим, что имеется некоторая система управления молочным магазином или сетью магазинов. В качестве одной из её составных частей выступает

подсистема учёта товаров. Молочные продукты характеризуются следующим набором атрибутов:

1. наименование товара;
2. поставщик;
3. дата поставки;
4. количество товара в поставке;
5. единицы измерения;
6. цена поставщика;
7. цена продажи;
8. жирность;
9. конечная дата реализации.

По тем или иным причинам произошло изменение инфологической модели и теперь требуется регистрировать не только молочные продукты, но, допустим, и мебель. Предположим, что для мебели характерен следующий набор атрибутов:

1. наименование товара;
2. поставщик;
3. дата поставки;
4. количество товара в поставке;
5. единицы измерения;
6. цена поставщика;
7. цена продажи;
8. габариты;
9. тип покрытия (отделочный материал);
10. цвет.

Можно отметить, что первые семь атрибутов являются общими для каждого вида товара, в то время как остальные атрибуты у них уникальны.

Нам необходимо создать такую схему хранения информации, которая бы в соответствовала условиям примера.

1.1. Недостатки существующих реализаций реляционной модели

Попробуем найти решение проблемы в рамках современных РСУБД. Можно предложить три варианта решения:

- соединение всех атрибутов в единой сущности “ТОВАРЫ”;
- группировка общих атрибутов в одном отношении и разнесение уникальных атрибутов по различным вспомогательным отношениям;
- представление каждой сущности в виде отдельного отношения.

Примечание Ситуация хранения только общих атрибутов не рассматривается, поскольку противоречит условиям примера.

Рассмотрим достоинства и недостатки каждого решения.

1.1.1. Соединение всех атрибутов в едином отношении

Такое решение практиковалось в подавляющем большинстве случаев, которые мне приходилось наблюдать. Но можно ли назвать такое решение хорошим? Действительно, такие показатели, как “конечная дата реализации” и “жирность” являются обязательными для вида товара “Молочные продукты”, но они бессмысленны при описании вида товара “Мебель”. Аналогично можно сказать, что атрибуты “габариты”, “тип покрытия” и “цвет”, полезны только при описании “Мебели”, а при работе с “Молочными продуктами” их значения становятся неопределёнными. То есть, мы не можем объявить обязательными те атрибуты, которые являются характерными только для отдельных видов товаров, хотя это, безусловно, необходимо. Следовательно, наша система не сможет отследить ввод “Молочных продуктов” без указания срока реализации или жирности. Это может

привести к нежелательным последствиям. Попробуем наоборот, пусть атрибуты станут обязательными. Тогда потребуется ввод показателей “жирность” для мебели и “тип покрытия” для молока, но в реальности такой информации не существует. Возникает ситуация, когда пользователи в ответ на бессмысленный запрос могут ввести не менее бессмысленный ответ. В результате становится возможным, что на запрос выдать товар с наибольшим показателем жирности, мы получим в ответ: спальный гарнитур “Зима”.

Кроме того, отношение “ТОВАРЫ” будет обладать тенденцией к бесконечному росту при увеличении номенклатуры продаваемых товаров. Однако только незначительная часть атрибутов будет полезной в каждом конкретном случае. Рано или поздно отношение “ТОВАРЫ” превратится в монстра, который будет беспощаден к ресурсам сервера.

1.1.2. Группировка общих атрибутов в едином отношении

Такой способ решения позволяет избежать тех проблем, которые свойственны первому варианту. На самом деле, поскольку атрибуты свойственные каждой конкретной сущности разнесены по разным отношениям, то на них можно накладывать любые ограничения и проверки. С другой стороны, не происходит бесконтрольного увеличения отношения “ТОВАРЫ”, так как там хранятся только те атрибуты, которые являются общими для всех видов товаров. Но является ли это решение хорошим? Одно из незыблемых правил реляционной теории гласит, что поле (группа полей) одного отношения не могут быть другим отношением или ссылкой (прямой или косвенной) на другое отношение. Нам же придётся в отношение “ТОВАРЫ” ввести атрибут “вид товара” и по значению, сохранённому в этом поле, переходить либо к отношению “Молочные продукты”, либо к отношению “Мебель”. Такое решение проблемы приводит к нарушению принципов реляционной модели и лишает нас возможности полноценно использовать SQL, поскольку в нём не предусмотрена обработка подобных ситуаций.

1.1.3. Использование отдельных отношений

Теперь сущности: “Молочные продукты” и “Мебель” хранят свои атрибуты в отдельных отношениях. Это решение лучше первого варианта, поскольку не имеет его недостатков, и лучше второго, так как не приводит к нарушению правил использования реляционной модели. Но! Теперь утрачена такая сущность как “ТОВАРЫ”. Если у нас есть отношения, поддерживающие ссылочную целостность с отношением “ТОВАРЫ” (например, “Ведомость на списание”), то куда они будут ссылаться?

1.2. Применимость постреляционных моделей данных

Прежде всего, нельзя не отметить то, что переход на любую постреляционную модель (объектную или объектно-реляционную) влечёт за собой большие накладные расходы. Значительная часть расходов будет направлена на то, чтобы обеспечить перенос информации из существующей базы данных, модификацию программного обеспечения и обучение персонала. Кроме того, при использовании постреляционных баз данных возникают специфические трудности как при проектировании и создании базы данных, как и при её модификации и эксплуатации. Однако обсуждение этих вопросов выходит за рамки данной статьи.

1.3. Резюме

Приведённый пример - это лишь частный случай и с подобными проблемами можно столкнуться в любой предметной области: банковские вклады; номера в гостиницах; места в транспорте; специальности; должности; наконец, структурные подразделения на предприятии. Ни одно из решений, доступных в рамках существующих РСУБД, нельзя признать удовлетворительным. Означает ли это

ограниченность самой реляционной модели или это ограниченность её реализаций? Можно ли найти качественное решение для подобных задач или следует подумать об иной модели, например, постреляционной или объектной?

Решение, которое предлагается ниже, позволяет не только остаться в рамках реляционной модели, но и сохранить всё то, что сделано: и структуру базы данных, и программное обеспечение, а, следовательно, сохранить время и инвестиции в программное обеспечение.

2. Проекция понятий ООП на реляционную модель

Применимость технологии объектно-ориентированного проектирования зависит от возможности перенесения её базовых принципов и понятий на реляционную модель. К таким понятиям относятся: наследование, инкапсуляция и полиморфизм. Следует отметить, что *полной поддержки инкапсуляции в предлагаемом решении нет, поскольку нет сокрытия данных, хранимых в базе данных.* Собственно поэтому не стоит говорить, что данная методика превращает реляционную модель в объектную. Речь идёт об объектном представлении реляционной модели.

2.1. Проекция наследования

Можно отметить, что приведённый пример прекрасно иллюстрирует процесс наследования. Сущность "ТОВАРЫ" - это суперкласс, а сущности "Молочные продукты" и "Мебель" - его подклассы. Если продолжить аналогию, то заметим, что каждый подкласс должен обладать всеми атрибутами своего класса. Сосредоточив общие атрибуты сущностей в классе "ТОВАРЫ", мы тем самым определим их для всех его подклассов. Это именно то, что требовалось. Применительно к базам данных можно и нужно усилить данное положение: ***все атрибуты, индексы и ограничения, определённые на уровне класса, обязательны для всех подклассов, являющихся его наследниками***. Например, если на какой-то атрибут класса наложено ограничение уникальности, то он должен быть уникален на уровне всего класса, а не только внутри каждого подкласса. Это положение очень важно для поддержания целостности и непротиворечивости данных, поэтому, наверное, следует прокомментировать его подробнее. Допустим, у нас есть класс α и его подклассы β и γ. Предположим, что у класса α есть два атрибута *a* (integer) и *b* (date). Атрибут *a* определён как уникальный в классе α, и на атрибут *b* наложена проверка (CHECK (VALUE BETWEEN "1/1/1997" AND "1/1/1998")). Это означает, что если атрибут *a* принял значение *k* в одном из отношений β или γ, то значение *k* будет уникальным (единственным) для обоих подклассов β и γ. Аналогичные рассуждения можно привести и для атрибута *b*: ни в подклассе β, ни в подклассе γ значение данного атрибута не могут быть вне заданного диапазона.

Переопределение атрибутов, индексов и ограничений в подклассах недопустимо!

Использование наследования в существующих реляционных базах данных достаточно просто. Давайте вернёмся к нашему примеру. Первоначально магазин специализировался на продаже молочных продуктов, и мы имели в базе данных отношение "Молочные продукты", при изменении инфологической модели (появлении новой группы товаров "Мебель"), нам следует:

1. Определить общие для обоих типов товаров атрибуты и на их основе создать класс "ТОВАРЫ";
2. Объявить существующий класс "Молочные продукты" подклассом класса "ТОВАРЫ";
3. Создать новый класс "Мебель", как подкласс класса "ТОВАРЫ".

Такой путь выглядит более предпочтительно, чем освоение новых средств, наподобие "постреляционных СУБД", проектирования новой базы данных, определения схем переноса информации из существующей реляционной базы

данных в новую "постреляционную" и, наконец, переписывания всего программного обеспечения.

Классы могут быть реальными (ассоциированными с отношением) либо абстрактными (не имеющими ассоциированного отношения). Пока, для простоты изложения, будем считать, что каждый реальный класс ассоциируется с одним отношением, хотя, в общем случае, это ограничение лишает нас некоторых заманчивых возможностей.

2.2. Инкапсуляция

Современные реляционные СУБД не имеют механизма инкапсуляции: объединения кода и данных внутри класса. Однако они обладают всеми необходимыми компонентами для его создания. Безусловно, речь идёт о хранимых процедурах. Здесь есть два важных аспекта, первый, описание класса должно включать описание методов (свойств) и процедуры, отвечающей за реализацию объявленного метода в данном классе; второе, процедура должна выполняться в контексте того класса, метод которого с ней ассоциирован. Если выполнение первого требования достаточно тривиально и практически не зависит от используемой РСУБД, то второе условие реализуется с учётом специфики реализации 4GL выбранной РСУБД. Дело в том, что некоторые системы позволяют передавать и использовать в качестве параметров хранимых процедур имена таблиц или их полей, другие этого не допускают. В первом случае можно в качестве контекста передавать имя класса, в котором описан данный метод. Во втором случае потребуется, чтобы контекст (класс) был вложен непосредственно в хранимую процедуру. Различия в реализациях 4GL не должны влиять на приложения, которые работают с базой данных посредством описываемого объектного представления.

Не следует недооценивать сути механизма инкапсуляции, ибо благодаря его наличию в системе можно создавать сложные схемы взаимодействия между сущностями, представленными в базе данных. К примеру, на его основе можно разрабатывать расчётные схемы, схемы документооборота, потоков работ и управляющих воздействий, а также многое, многое другое.

2.3. Полиморфизм

Благодаря полиморфизму один и тот же метод (свойство) может иметь различные реализации у подклассов. Полиморфизм реализуется достаточно тривиально: одному и тому же методу ставятся в соответствие различные хранимые процедуры. Абстрактный класс может содержать объявление метода без ассоциированной с ним хранимой процедуры, в отличие от этого, ***в реальном классе отсутствие хранимых процедур в любом из методов является недопустимым***. Роль конструкторов и деструкторов класса выполняют предложения CREATE и DROP, все остальные методы являются виртуальными. Значит, подкласс может переопределить суть любого метода, полученного от класса. ***Количество, тип и порядок передачи параметров метода подкласса должен соответствовать количеству, типу и порядку передачи параметров у метода класса***.

2.4. Обмен сообщениями

К сожалению не все современные РСУБД имеют механизмы обмена сообщениями. Поэтому, столь важный механизм может быть реализован только факультативно. Обмен сообщениями - это то средство, которое даёт возможность проектировать очень сложные и гибкие системы. Менеджер сообщений позволяет относительно легко распределить выполнение сложного задания по нескольким потокам и/или серверам баз данных. Если выбор сервера баз данных ещё только предстоит, то имеет смысл рассматривать наличие в нём данной функции как один из положительных критериев.

3. Реализация объектного представления

Реализацию объектного представления логично начать с проектирования среды хранения информации о классах. Конечно, можно создать собственную систему хранения, но, поскольку речь идёт о базах данных, то, наверное, более разумно воспользоваться средствами СУБД. Базу данных объектного представления можно рассматривать как часть базы данных некоторой предметной области, либо как самостоятельную систему хранения информации. Второй путь более привлекателен, поскольку позволяет использовать одно и тоже объектное представление над несколькими базами данных. Что позволяет, в частности, перейти от объектного представления на уровне сущностей (отношений) к объектному представлению на уровне инфологической модели (баз данных), но это тема отдельного разговора.

База данных объектного представления аналогична по своей структуре базе метаданных, которые используются для хранения информации об отношениях в реляционных СУБД. Действительно, это сходство проистекает от единства класса и отношения. Класс обладает практически всеми атрибутами, характерными для описания отношений: полями, индексами, триггерами, ограничениями и проверками. Дополнительным свойством классов является наличие у них методов. Следовательно, можно для описания классов использовать схемы, аналогичные схемам метаданных используемой СУБД, а, с другой стороны, *позволяет применять стандартные средства взаимодействия с базами данных, при незначительной их модификации*. Отсюда можно вывести ещё одно следствие: перетрансляция запросов из объектного представления в реляционное представление не требует сложных схем и значительных ресурсов.

Программное обеспечение, выполняющее трансляцию запросов из объектного представления в реляционное, будем называть сервером объектного представления (SOV - Server of Object View). Его задачами являются:

- получение запроса;
- определение необходимости перетрансляции и перетрансляция;
- направление запросов к СУБД;
- получение результата выполнения запроса;
- возможное преобразование результатов запроса.

Список перечисленных задач является минимальным, но не исчерпывающим. В реальных системах могут потребоваться дополнительные функции, например, те что реализованы в мониторах транзакций.

3.1. Метаданные классов

Схемы хранения объектного представления могут быть достаточно разнообразны, и их проектирование зависит от тех возможностей, которые они призваны обеспечить. Упрощенная схема объектного представления приведена на рис. 1. Она обеспечивает выполнение основных функций объектного представления: создание, изменение и уничтожение классов; предоставление необходимой информации для перетрансляции запросов и преобразования результатов выполнения запросов. Схема представляет собой концептуальную модель и реализована в виде ER-диаграммы (диаграммы Сущность-Связь).

Сервер объектного представления независим от сервера базы данных, как с точки зрения инструментальных средств, так и с точки зрения физического размещения. То есть сервер объектного представления может использовать свою СУБД, например, локальную, и свою базу данных. Реальное расположение базы данных может повлиять на производительность всей системы. Существует два приемлемых варианта расположения: на сервере баз данных или на сервере объектного представления, если, конечно, это различные сервера. В первом случае желательно иметь быстрый канал связи между сервером или серверами баз данных и сервером объектного представления. Второй случай предъявляет повышенные

требования к ресурсам сервера объектного представления, поскольку на нём должна производительно работать некоторая СУБД.

Ключевой сущностью в данной схеме является понятие классов. Связь Inherited (CLASSES – CLASSES) реализует схему наследования. Поскольку связь является необязательной, то, следовательно, допустимо любое количество классов, не имеющих суперкласса. Схема наследования не предусматривает введение понятия множественного наследования, то есть каждый класс имеет один и только один суперкласс. Для реализации множественного наследования необходимо реализовать бинарную связь многие ко многим (M-N связь). Но подобная практика не представляется мне разумной.

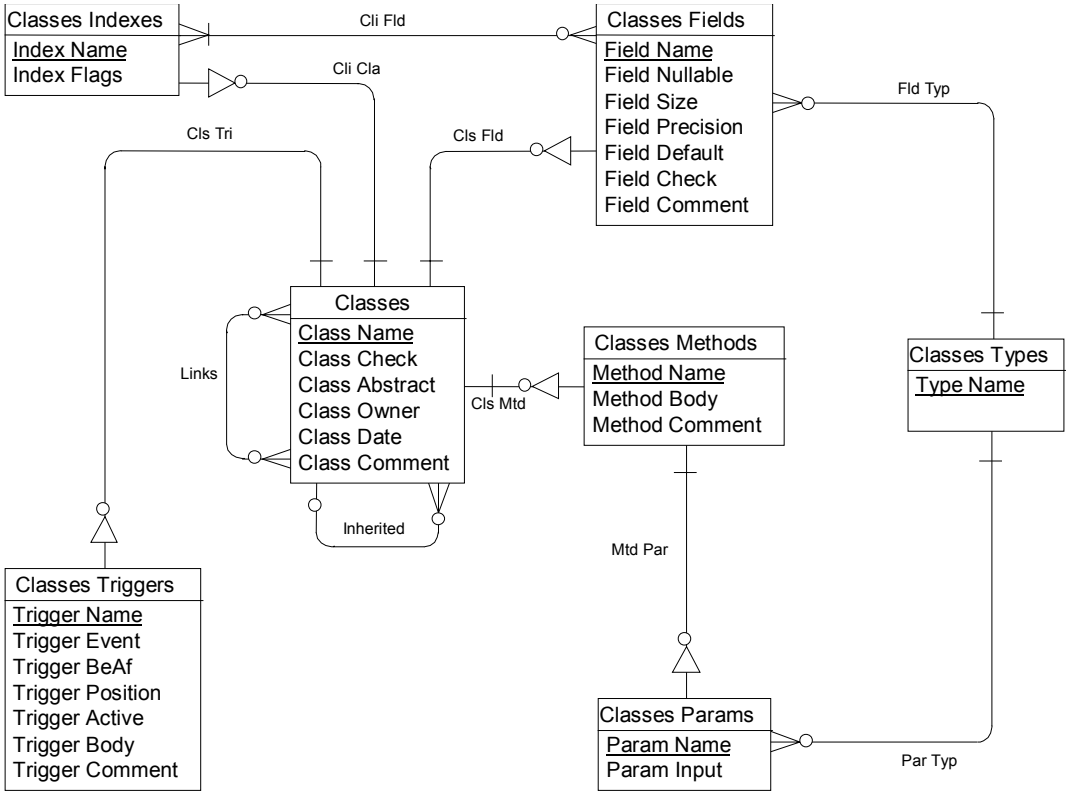

Рис. 1. Концепт уальная схема объект ного предст авления

Вторая связь - Links (CLASSES – CLASSES) позволяет устанавливать бинарные связи между классами. Из схемы видно, что данная связь представляет собой бинарную связь многие ко многим (M-N связь), то есть каждый класс может иметь произвольное количество связей с другими классами или с самим собой. В реляционной модели аналогом данной связи является связь, устанавливаемая по внешнему ключу (FOREIGN KEY). Строго говоря, данный тип связи возможен не только между классами, но и между классом и отношением. Однако можно сделать простое допущение, что каждое отношение представимо собственным классом. Это допущение устраняет неоднозначность.

3.1.1. Поля сущностей

3.1.1.1.Classes

Classes – это базовая сущность сохраняющее полное описание класса.

Class Name – название класса. Уникальный идентификатор (первичный ключ) класса в рамках системы.

Class Check – поле, содержащее все проверки, выполняемые на уровне класса. Например, если класс содержит поля ДАТА ПОСТУПЛЕНИЯ ТОВАРА и ДАТА СПИСАНИЯ ТОВАРА, то было бы целесообразно установить проверку ДАТА ПОСТУПЛЕНИЯ… < ДАТА СПИСАНИЯ…. Поскольку количество контрольных значений может быть произвольным, то можно выделить их в отдельную сущность. Но можно предположить, что сами проверки хранятся либо в виде триггеров, либо в некотором откомпилированном виде, и запускаются автоматически. В таком случае, допустимо хранить все контрольные проверки для каждого класса в одном поле. В любом случае, мне показалось неразумным "засорять" концептуальную схему излишними деталями.

Class Abstract – логическое поле, принимающее значение TRUE, если класс является абстрактным и, значение FALSE, если класс имеет ассоциированное с ним отношение.

Class Owner – поле, в котором сохраняется пользователь-разработчик, создавший данный класс.

Class Date – поле, в котором сохраняется дата создания класса.

Class Comment – поле комментария, где допустимо сохранять любые комментарии по поводу целей создания и правил использования класса. Хороший стиль программирования, в частности, рекомендует тщательно документировать все шаги, особенно, если их логику придётся понимать не только разработчику, но и его коллегам.

3.1.1.2.Classes Fields

Classes Fields – сущность, определяющая атрибуты класса. Данная сущность имеет бинарную связь с классом один ко многим (1-N), что означает, что каждый класс может иметь произвольное количество атрибутов, но каждый атрибут принадлежит только одному классу.

Field Name – поле, сохраняющее название атрибута данного класса. Значение данного поля должно быть уникальным в рамках класса.

Field Nullable – логическое поле, определяющее может ли данный атрибут принимать значение NULL. Если поле имеет значение TRUE, значит атрибут может быть NULL, в противном случае, он должен быть NOT NULL.

Field Size – поле, определяющее предельный размер атрибута. Данное поле используется в случае, если атрибут не имеет фиксированного размера, например, является строковым.

Field Precision – поле, в котором задаётся точность представления атрибута. Используется только для атрибутов представляющих вещественные числа.

Field Default – поле, в котором сохраняется значение атрибута по умолчанию.

Field Check – поле, в котором сохраняются проверки правильности значения атрибута.

Fields Comment – поле для комментария. Здесь желательно указывать назначение атрибута и правила его использования в прикладных программах и запросах.

3.1.1.3.Classes Indexes

Индексы класса имеют бинарные связи с собственно классами и полями классов. Каждый класс может иметь произвольное число индексов, но каждый индекс принадлежит только одному классу. Это определяет бинарную связь между классами и индексами, как один ко многим (1-N). Бинарная связь с полями класса является связью многие ко многим (N-M), поскольку каждое поле может быть включено в произвольное число индексов, и каждый индекс может содержать произвольное число полей. Строго говоря, классы, атрибуты и индексы образуют

тринарную связь, но её невозможно отобразить современными средствами проектирования на уровне концептуальной модели. Исходя из этого, и было допущено упрощение.

Index Name – поле, определяющее название индекса. Название индекса должно быть уникальным в пределах базы данных или класса, если используемая СУБД позволяет пересечение имён индексов, принадлежащих различным таблицам (отношениям).

Index Flags – данное поле представляет собой комбинацию флагов. Первый флаг отвечает за активность/пассивность индекса. Вторая группа флагов определяет направление сортировки значений по возрастанию или убыванию. Наконец, третья группа флагов определяет тип индекса: первичный (PRIMARY), уникальный (UNIQUE), внешний (FOREIGN), ординарный (ORDINARY).

3.1.1.4.Classes Triggers

Триггеры позволяют описывать реакции на то или иное событие относительно своего класса.

Trigger Name – поле, содержащее имя триггера.

Trigger Event – это поле определяет событие, на которое обязан отреагировать триггер (Insert, Update, Delete).

Trigger BeAf – логическое поле, которое определяет когда должен сработать триггер: до (before - FALSE) или после (after - TRUE) события.

Trigger Position – это поле определяет последовательность срабатывания триггеров. Оно необходимо в случае, если существует несколько триггеров на одном и том же событии.

Trigger Active – логическое поле, определяющее активность триггера. FALSE – означает, что триггер пассивен, TRUE – свидетельствует об активности триггера.

Trigger Body – поле, сохраняющее текст триггера.

Trigger Comment – поле необходимое для документирования назначения и правил использования триггера.

3.1.1.5.Classes Methods

Методы позволяют определить набор действий, допустимых для данного класса. Иными словами, методы образуют высокоуровневый интерфейс класса. По своей сути они являются хранимыми процедурами, ориентированными на работу с определённым классом.

Method Name – поле, содержащее имя метода.

Method Body – текстовое поле, содержащее описание метода на языке используемой СУБД.

Method Comment – комментарии к методу и правила его использования.

Каждый метод может содержать произвольное число параметров. Следует отметить, что некоторые современные СУБД позволяют использовать в теле хранимой процедуры переменные значения таблиц и полей, а другие СУБД не позволяют. В первом случае можно передавать имя класса (отношения) в качестве параметра, во втором случае – этого сделать нельзя. И тогда необходимо создавать каждый метод для каждого класса даже, если подкласс не переопределяет какой-либо метод своего суперкласса! В этом случае описание метода в суперклассе используется как шаблон для методов подклассов.

3.1.1.6.Classes Params

Параметры, передаваемые в методы, обладают именем, типом и направлением (входные или выходные).

Param Name – поле, содержащее имя параметра.

Param Input – логическое поле, определяющее является ли параметр входным (TRUE) или выходным (FALSE).

3.1.1.7.Classes Types

Classes Types – это справочное отношение содержащее список доступных для данной СУБД ординарных типов. К ним относятся, например, числовые типы, включая целые и вещественные, тип даты, строковые типы и т.д.

Classes Name – это имя типа данных, совпадающее с именем соответствующего типа данных в применяемой СУБД.

3.2. Перехват запросов

Сервер объектного представления должен получать запросы раньше, чем они попадают на обработку в ядро СУБД. Это необходимо для перетрансляции SQL запроса из объектной формы. Операция перетрансляции достаточно тривиальна и не требует существенных временных затрат или ресурсов. Объектная форма запросов должна быть максимально приближена к естественной форме запросов SQL, то есть очень желательно, чтобы изменения в SQL были минимальны, настолько, насколько это возможно. Это позволит безболезненно устанавливать сервер объектных запросов на уже действующих системах с большим набором прикладного программного обеспечения.

Само существо процесса перехвата сильно зависит от используемой СУБД и в большинстве случаев не представляет большого труда. Алгоритм работы сервера объектного представления предельно лаконичен:

- получить запрос;
- проверить относится запрос к классу (классам) или отношениям;
- если запрос направлен к классу (классам), то выполнить перетрансляцию;
- направить запрос к СУБД;
- получить результаты;
- если необходимо, то выполнить дополнительную обработку результатов запроса;
- вернуть результаты запроса пользователю.

3.2.1. Проверка запроса

Проверка запроса сводится к получению списка таблиц, участвующих в запросе. Для каждой таблицы делается попытка найти её имя в отношении CLASSES. Если хотя бы одна таблица представляет собой класс, то запрос направляется на стадию перетрансляции. В противном случае, он перенаправляется к СУБД в первозданном виде.

3.2.1.1.Запросы на создание класса

Запрос на создание класса следует отличать от запроса на создание обычной таблицы, с этой целью в операторе CREATE TABLE вводится дополнительно четыре опциональных параметра.

CREATE TABLE <table name> [AS CLASS [ABSTRACT][INHERITED FROM <class name>]] (
…
[METHOD <method name> (<list of parameters>) [,
METHOD <method name> (<list of parameters>)]]
)

Опция AS CLASS определяет, что <table name>, специфицирует имя класса.

Опция ABSTRACT определяет класс, как абстрактный. Здесь и далее под абстрактным классом понимается класс, который не имеет связанных с ним одной или более таблиц.

Опция INHERITED FROM определяет, что данный класс является подклассом <class name>.

Опция METHOD определяет, что хранимая процедура с именем <table name> || <method name> является методом <method name> данного класса. Оператор "||"

обозначает строковую операцию конкатенации. Вы вправе принять любое другое соглашение по именованию методов.

При создании подкласса, не являющегося абстрактным, необходимо получить имена и типы полей суперкласса, а также его индексы, триггеры, методы и ограничения. Это требуется для создания одной или более таблиц (экземпляров класса) ассоциированных с данным классом. Таблицы, ассоциированные с классом, должны иметь весь набор полей, полей, индексов, триггеров, методов и ограничений, определённых у всех суперклассов в совокупности.

Рассмотрим простейшую схему наследования. Пусть создаётся класс *C*, который является подклассом класса *B*, который, в свою очередь, является подклассом класса *A* (*A⇒B⇒C).* Пусть класс *A* имеет поля f_{1a} и f_{2a}, индекс i_{1a} и методы m_{1a}, m_{2a} и m_{3a}. Класс *B* имеет поля f_{1b}, f_{2b} и f_{3b}, и метод m_{1b}. Класс *C* имеет поле f_{1c}, индекс i_{1c}, методы m_{1c} и m_{2c}, а также ограничения co_{1c} и co_{2c}. Тогда таблица-экземпляр, созданная на основе описания класса *C* будет иметь:

поля: f_{1a}, f_{2a}, f_{1b}, f_{2b}, f_{3b}, f_{1c};
индексы: i_{a1}, i_{c1};
методы: m_{1a}, m_{2a}, m_{3a}, m_{1b}, m_{1c}, m_{2c};
ограничения: co_{1c}, co_{2c}.

В большинстве случаев порядок следования и определения полей, индексов, триггеров, методов и ограничений безразличен, но лучше придерживаться некоторого выбранного алгоритма, например, последовательность определения начинается с самого дальнего суперкласса (корня иерархии).

3.2.1.2.Запросы на изменение класса

Наверное, нет смысла говорить, что менять структуру классов – скверная практика. Но в момент разработки возникают ситуации, когда данная операция необходима. Именно поэтому должна быть реализована возможность изменения структуры классов. Насколько осторожно и аккуратно ею будут пользоваться разработчики, целиком зависит от их опыта и квалификации. Аналогично тому, как мы можем менять структуру таблиц с помощью оператора ALTER TABLE, точно также можно воспользоваться этим оператором для изменения структуры класса.

ALTER TABLE <table name> [AS CLASS [ABSTRACT][INHERITED FROM <class name>]] (ADD|DROP) …
[METHOD <method name> (<list of parameters>) [,
METHOD <method name> (<list of parameters>)]]

Опция AS CLASS позволяет превратить таблицу с именем <table name> в одноимённый класс. ABSTRACT определяет, что класс является абстрактным, если ранее класс существовал, но как реальный, то применение к нему данной опции приведёт к уничтожению связанных с ним одной или более таблиц. Опция INHERITED FROM <class name> делает возможным наследование подкласса от класса с именем <class name>. Это приводит к тому, что подкласс будет расширен полями, индексами, ограничениями, триггерами и методами суперклассов. В случае пересечения имён, например, если исходный класс и суперкласс имеют поля с одинаковым именем, операция должна быть отменена с выдачей соответствующего диагностического сообщения.

Традиционно изменение таблиц возможно только за счёт добавления или удаления полей и ограничений (CONSTRAINT). Но, работая с классами, можно аналогичным образом добавлять или удалять методы.

Если класс, в котором производятся изменения, имеет подклассы, то все изменения в базовом классе должны быть спроецированы на его подклассы. Так, например, удаление поля в суперклассе должно повлечь удаление соответствующих полей в каждом его подклассе. Аналогичные требования можно выдвинуть и относительно индексов, ограничений, триггеров и методов.

3.2.1.3.Запросы на удаление класса

Запрос на удаление класса синтаксически полностью аналогичен запросу на удаление таблицы.

DROP TABLE <class name>;

Здесь <class name> является именем удаляемого класса.

При удалении класса следует предусмотреть проверку ссылочной целостности. Если класс имеет бинарные связи с другими классами или таблицами, то его удаление невозможно. В такой ситуации операция отменяется, а пользовательское приложение должно быть оповещено о возникшей проблеме.

Удаление суперкласса автоматически приводит к удалению всех его подклассов.

3.2.1.4.Запросы на изменение данных

К запросам на изменение пользовательских данных относятся запросы вставки (INSERT), изменения (UPDATE) и удаления (DELETE). Обработка этих запросов имеет как сходные, так различные черты. Главное различие в обработке запроса на вставку и запросами на изменение и удаление заключается в том, что запрос на вставку всегда направлен к одному и только одному экземпляру класса (таблице). В то время, как запросы на изменение и удаление захватывают не только тот класс, к которому они обращены, но и множество его подклассов. Это свойство не всегда очевидно. Можно проиллюстрировать это простым примером. Нельзя создать (отобразить) абстрактную плоскую фигуру, но не составляет труда нарисовать окружность заданного радиуса и цвета. В то время как можно потребовать изменить цвет любую фигуру или потребовать убрать с экрана фигуры, цвет которых равен заданному в условии цвету. Первое действие по созданию (отображению) аналогично добавлению, вторая и третья операции аналогичны изменению и удалению записей в некоторой таблице.

В связи с изложенным выше, следует отметить, что запрос на вставку в абстрактный класс, лишён смысла. Нельзя добавить в базу данных абстрактный товар, но можно занести в неё кефир или спальный гарнитур. При попытках записи в абстрактный класс приложение пользователя должно быть уведомлено об ошибке. Но поскольку вставка в класс ничем не отличается от вставки в таблицу, то нет нужды в каких-либо изменениях синтаксиса оператора вставки.

Здесь имеет смысл отметить что, тем не менее, запросы на вставку должны перехватываться сервером объектного представления и обрабатываться. Это связано с наличием связей между классами. Поскольку существующие СУБД поддерживают связи только между таблицами, а класс может быть представлен множеством таблиц, то, следовательно, сервер объектного представления должен выполнить эту работу сам. В качестве иллюстрации, можно привести простой пример. Пусть товары в магазин поставляют множество различных поставщиков, которые можно разделить на группы: поставщики-производители продукции, оптовые поставщики и, наконец, частные лица. Все группы поставщиков наряду с общими атрибутами обладают и уникальными, характерными только для данной группы. Тогда целесообразно представить каждую группу поставщиков, как подкласс абстрактного класса “ПОСТАВЩИКИ”. Теперь у нас появляется необходимость связать два абстрактных класса “ТОВАРЫ” и “ПОСТАВЩИКИ” бинарной связью многие ко многим (M-N) (каждый поставщик может поставлять множество разнообразных товаров и один вид товара может поставляться множеством различных поставщиков). Теперь при занесении в базу данных информации о новой поставке товара нам необходимо связать его с поставщиком. Однако поставщик товара может принадлежать любой группе (классу). Поэтому невозможно установить связь на уровне таблиц, но связь “ПОСТАВЩИКИ” – “ТОВАРЫ” абсолютно справедлива на уровне классов. С другой стороны, поскольку каждый товар должен иметь своего поставщика, то справедливо

утверждение, что связь между поставщиком и товаром определяется на абстрактном уровне классов "ПОСТАВЩИКИ" – "ТОВАРЫ", что и требовалось доказать.

Установление связей между классами значительно упрощает решение многих задач, которые в современных РСУБД решаются весьма плохо. Но задача поддержания ссылочной целостности между классами целиком возлагается на сервер объектных запросов, поскольку СУБД этого делать не умеют. Данная задача не столь трудна и имеет тривиальное решение.

Запросы на изменение и удаление, направленные к конкретному классу воздействуют и на все его подклассы. Здесь опять не возникает необходимости в изменении синтаксиса SQL запросов данного вида. При удалении всегда, а при изменении в только случае, если изменяются поля, входящие в уникальные (первичный) индексы, необходимо выполнять проверку на ссылочную целостность. Вся операция по удалению и изменению должна проводиться в рамках одной транзакции примерно в следующей последовательности:

- начать транзакцию;
- запустить триггеры класса (before);
- проверить ссылочную целостность;
- провести операцию;
- запустить триггеры класса (after);
- закончить транзакцию.

В случае если хотя бы один из шагов не может быть выполнен, транзакция должна быть возвращена в исходное состояние без каких-либо изменений в базе данных (rollback). При успешном завершении всех операций транзакция должна быть завершена оператором commit work.

3.2.1.5.Запросы на выборку данных

Выборка данных из класса сводится к выборке данных из всех таблиц, ассоциированных, как с данным классом, так и со всеми его подклассами. Такое решение позволяет пользователям рассматривать любой класс, как одно цельное отношение. Детали того, что данное отношение является логическим и разбросано по нескольким физическим отношениям (таблицам), скрыты от пользователя. В результате достигается возможность абстрагирования от несущественных деталей. Когда нужно видеть такую сущность, как товар, то бессмысленно уточнять о каком виде товара идёт речь. Пользователь имеет возможность манипулировать только теми атрибутами, которые являются общими для любого товара. С другой стороны, сущность "ТОВАРЫ" определяет первичный ключ и, следовательно, не возникает проблем в случае, если необходимо узнать более подробную информацию о конкретном товаре. Связи между классами дают возможность составлять запросы на весьма высоком уровне абстракции. Например, мы будем в состоянии ответить на вопрос о том, какие партии товара поставлял поставщик, имеющий заданный адрес (банковские реквизиты, и т.п.), в течение определённого периода времени. При этом подразумевается, что и сущность "ПОСТАВЩИКИ" и сущность "ТОВАРЫ" представлены классами, то есть имеют несколько отличающихся в деталях видов.

Запросы на выборку данных записываются на SQL с помощью оператора SELECT. Этот оператор в своей объектной форме ничем не отличается от традиционного оператора и может иметь сложную структуру. Схема трансляции запроса, направленного к классу, в запрос, направленный к физическим отношениям (таблицам) логически достаточно прост и может иметь различные реализации. Реализация схемы трансляции сильно зависит от используемой СУБД. Дело в том, что каждая коммерческая СУБД имеет как свои ограничения, так и сильные/слабые стороны при выполнении сложных запросов. Для достижения эффективной реализации эти особенности конкретной СУБД необходимо учитывать.

Самое простое решение, доступное почти для всех коммерческих СУБД – это порождение временных таблиц. Однако такое решение может снизить скорость

выполнения запроса на некоторых СУБД. Для них возможно более эффективным решением будет динамическая генерация VIEW, курсоров или даже хранимых селективных процедур. Наконец, некоторые СУБД имеют очень эффективные оптимизаторы запросов и могут легко "переваривать" запросы практически любой сложности. Для таких СУБД вполне применимо решение "в лоб" с использованием соединения запросов к нескольким отношениям посредством UNION. В исключительных случаях, можно превратить запрос к классу в последовательность запросов к физическим отношениям и "склеивание" результатов проводить непосредственно на сервере объектного представления. Но следует учитывать, что наличие в запросе агрегатных функций или предложений ORDER BY или GROUP BY, может привести к весьма сложной реализации самого сервера объектного представления.

Суть механизма трансляции сводится к разбору предложений FROM, как основного запроса, так и вложенных подзапросов. Классы, перечисленные в данных предложениях, должны быть заменены на совокупность таблиц, ассоциированных, как с самими классами, так и с их подклассами.

Пример трансляции запроса с помощью соединения (UNION) можно проиллюстрировать следующим образом:

пусть есть некоторый запрос о товаре:

SELECT * FROM GOODS WHERE SUPPLIER = 'Рога и копыта';

Его трансляция из объектной формы в физическую форму примет следующий вид:

SELECT <fields of class> FROM FOODS WHERE SUPPLIER = 'Рога и копыта'
UNION
SELECT <fields of class> FROM FURNITURES WHERE SUPPLIER = 'Рога и копыта';

Здесь <fields of class> есть перечень полей, определённых для класса "ТОВАРЫ" (GOODS), но не для отношений "ПРОДУКТЫ" (FOODS) или "МЕБЕЛЬ" (FURNITURES). Поскольку набор атрибутов каждого из отношений может быть больше, чем у класса "ТОВАРЫ", то, следовательно, выбираться из отношений будет только некоторое подмножество принадлежащих им атрибутов. В этом случае необходимо заменить символ "*" на список атрибутов, которые определены на уровне класса "ТОВАРЫ".

Аналогичным образом в более сложных запросах можно сохранять результаты вложенных подзапросов во временных таблицах или продуцировать VIEW, курсоры или селективные хранимые процедуры.

4. Заключение

4.1. Преимущества использования серверов объектного представления

Классифицированное представление информации и возможность работы на различных уровнях абстракции позволяет пользователям, проектировщикам и разработчикам значительно проще решать свои задачи. Для пользователей удобство заключается в том, что они могут получать корректные результаты даже не подозревая об изменениях в структуре хранения информации. То есть разработчики могут добавить в систему хранения новые сущности, не прибегая к утомительной процедуре оповещения пользователей. Действительно, давайте представим, что нам потребовалось ввести новый подкласс "АВТОМОБИЛИ" класса "ТОВАРЫ". Для пользователей, которые работали с сущностями "ПРОДУКТЫ" или "МЕБЕЛЬ" не произойдёт никаких изменений. Нет необходимости в переписывании программного обеспечения или переучивании самих пользователей. Их работа не претерпит никаких изменений. Для тех пользователей, которые оперировали абстрактным классом "ТОВАРЫ", тоже ничего не изменится, хотя и появился новый подкласс "АВТОМОБИЛИ". Схема трансляции объектной формы запроса на физические

отношения сделает для них механизм подключения новых подклассов совершенно прозрачным. Они могут и не знать о том, что теперь их запросы включают в себя обращения к новому подклассу. Таким образом, достигается не только экономия времени разработчиков, но и исключаются многие моменты, связанные с переходом на новые информационные схемы.

С точки зрения разработчиков и административного персонала удобство объектной схемы тоже не вызывает сомнения. Прежде всего, исчезает проблема соответствия схемы хранения информации (схемы БД) и программного обеспечения. Количество причин рассогласования сокращается весьма значительно, если не исчезает полностью. Упрощается и разработка нового программного обеспечения. Сегодня ведущие RAD системы позволяют создавать шаблоны диалогов (форм). Если связать понятие шаблона с понятием класса, то тогда разработка программного обеспечения для нового подкласса, начинается не с чистого листа. Разработчикам остаётся фактически определить только работу с теми атрибутами и методами, которые определены у нового подкласса. При разработке сложных систем такой подход позволяет не только экономить время, но и упрощает создание форм в едином стиле.

Реальная проблема, с которой сталкиваются все проектировщики, заключается в том, что отсутствует исчерпывающее представление обо всех сущностях предметной области, их связях и ограничениях. Технология объектного проектирования, в такой ситуации, может быть панацеей. Любая объектная система обладает способностью к развитию. Следовательно, нет необходимости создавать всю систему целиком. На первом этапе определяются ключевые понятия и сущности, уточнять которые можно впоследствии бесконечно долго. Если вернуться к нашему примеру, то вначале достаточно определить такие сущности, как “ПОСТАВЩИКИ” и “ТОВАРЫ”. Определить взаимосвязи и ограничения. После этого достаточно ввести всего один реальный подкласс класса “ПОСТАВЩИКИ” и один реальный подкласс класса “ТОВАРЫ”. Этого достаточно, что бы отработать большинство функций взаимодействия, существующих между этими сущностями. В дальнейшем можно совершенно безболезненно добавлять как новые виды поставщиков, так и новые виды товаров. Система будет гарантировано работоспособна.

В больших проектах очень важно достичь высокого уровня параллелизма. Использование технологии объектного проектирования позволяет решить и эту проблему. После определения основных (абстрактных) сущностей, их структуры и правил функционирования, можно приступать к детализации, разрабатывая реальные отношения и определяя их взаимодействие. Эта фаза выполняется параллельно для каждого реального отношения, поскольку основные интерфейсы и характеристики специфицированы на абстрактном уровне.

4.2. Дальнейшее развитие объектного представления

Идеи, изложенные выше, можно развивать в нескольких направлениях. Например, можно ввести понятие классов-переключателей (switch-classes). Суть этих классов понятна из следующего примера: допустим, создаётся база данных для результатов социологических опросов. Предположим, что с точки зрения исследователей очень важно иметь раздельное представление о результатах различных групп респондентов (например, по возрастным группам или по социальному положению или по половому признаку). Безусловно, можно разместить все результаты в едином отношении, введя соответствующее поле или поля, отвечающие за необходимую группировку. Но можно поступить иначе. Например, абстрактному отношению “РЕСПОНДЕНТЫ” поставить в соответствие некоторый домен значений. В этом случае результаты каждой группы будут размещены в отдельной физической таблице, что эффективнее. Работа класса-переключателя заключается в том, что он определяет из предложения WHERE к какому физическому отношению перенаправить запрос. Здесь опять пользователь использует прозрачный

для себя механизм и может получать, как значения из всей совокупности таблиц данного класса, так и из конкретной таблицы. Данный пример предназначен для иллюстрации механизма работы классов-переключателей, но из него неочевидны преимущества данной схемы хранения информации. Чтобы оценить достоинства необходимо вырваться из привычной иерархии: база данных ⇒ отношение ⇒ атрибут.

Давайте рассмотрим более сложный пример. Пусть нам предстоит разработать систему класса ПРЕДПРИЯТИЕ. Любое предприятие состоит из структурных подразделений. В свою очередь структурные подразделения включают в себя три сущности: люди (сотрудники), оборудование и материалы. Следовательно, можно запроектировать абстрактное структурное подразделение! А значит и создать абстрактное предприятие. Но сущность "СОТРУДНИКИ", как впрочем, и две другие сущности, представляются несколькими отношениями. Таким образом, появляется реальная потребность перенести объектное представление с уровня физических отношений на уровень баз данных. Но здесь и возникает основная проблема использования существующих коммерческих СУБД, так как они недостаточно эффективны при одновременном обслуживании большого числа баз данных. Даже те из них, которые декларируют мультибазовый режим работы, при таких нагрузках требуют значительных ресурсов от сервера баз данных и всё равно недопустимо теряют производительность. А между тем, использование объектных технологий позволило бы проще и эффективнее решать многие задачи. "Ахиллесова пята" предприятий – оптимальная структура и управление. Столь популярное слово как "реенжиниринг" (reengineering) сегодня не нашло своего воплощения в виде готовых программных продуктов, и это не смотря на всю привлекательность решения данной задачи. На чём базируется идея реенжиниринга – на высокой гибкости современного производства. Выживает тот, кто имеет более гибкую структуру производства и более эффективное управление. Изменения в структуре производства перестали быть единовременной акцией и должны происходить непрерывно, дабы соответствовать требованиям рынка. В современных условиях – это очень большая проблема для специалистов, отвечающих за информационное обеспечение. Однако решение есть, и оно лежит на поверхности. Предположим, что мы уже сумели создать с помощью некоторой гипотетической СУБД сеть баз данных структурных подразделений. Теперь опишем функции, которые должно выполнять это абстрактное предприятие. Очевидно, что для любого предприятия справедливо будет утверждение, что имеется один или несколько входов, на которые из внешнего мира подаётся нечто (назовём это сырьём). На другой стороне, существует один или несколько выходов, куда поступает другое нечто (назовём это продукцией). Обслуживание входов и выходов (поиск сырья и поставщиков, складирование, маркетинг, сбыт товара, поиск рынков и т.д.) – это некоторые функции. Далее, возможно, но не обязательно, сырьё в процессе производства меняет свою структуру, превращаясь в продукцию. Это ещё один набор функций, включая собственно соблюдение технологии и контроль. Все функции прекрасно представимы в виде графов, и они могут быть описаны в виде набора бизнес-правил. Теперь нам осталось предоставить менеджерам возможность накладывать полученный граф функций на сеть структурных подразделений. Это и будет основной инструмент, позволяющий наглядно и просто проводить процесс реенжиниринга. Он позволит аналитикам менять функции и структуру подразделений, определять оптимальные пути прохождения управляющих воздействий и обратных реакций, просчитывать эффективность каждой операции и выносить обоснованные заключения.

Осталось ответить на вопрос о том, возможно ли создание СУБД, отвечающей заданным требованиям, в обозримом будущем? Да, создание такой СУБД вполне возможно. Собственно все идеи, высказанные выше, рождались в обратной последовательности. Сначала была разработана спецификация на объектную СУБД. Она не имеет ничего общего с теми СУБД, которые сегодня получили статус

объектно-ориентированных, поэтому, чтобы избежать путаницы, теперь её называют Эго-системой. Она допускает практически неограниченное число уровней представления информации и легко адаптируется под данные классы задач. Однако её практическое воплощение не получило поддержки. Далее последовала трансляция найденных решений на существующие РСУБД, часть из которых изложена в этой статье. Описание Эго-системы тема отдельной очень большой работы. Эго-система привлекает тем, что начинается с понятия личности пользователя (отсюда в названии Эго) и может простираться до глобальных систем, создавая единую информационную среду.

4.3. Важное и последнее замечание

Мне бы очень хотелось определить своё отношение к существующим идеологиям развития СУБД. Иерархические и сетевые модели хранения информации в своё время сыграли положительную роль в становлении информационных систем. Но появление великолепных работ Кодда, описывающих реляционную модель, стало тем поворотным моментом, когда возможность свободной работы с информацией стала доступной не только разработчикам систем, но и пользователям. Появление объектных СУБД в конце 80-х годов стало своего рода фальш-стартом. Они не привнесли ничего нового в способы хранения информации. Сегодня, когда объектный подход к созданию программного обеспечения получил широкое распространение (что, к сожалению, не всегда сопровождается пониманием этой технологии), ООСУБД вызывают большой интерес. Мне кажется, что бездумное следование моде, может сильно затормозить развитие программного обеспечения в ближайшие годы.

В данной работе мне хотелось показать, что объектная технология не должна противопоставляться реляционной модели. Наоборот, достоинство той и другой парадигмы раскрываются в полной мере тогда, и только тогда, когда они работают вместе. Ранее подчёркивалось, что введение элементов объектной технологии позволяет сохранить без изменения значительную часть программного обеспечения, структуру базы данных и при этом повысить её гибкость. Именно этим и примечательна предлагаемая техника.

Эго-система спроектирована исключительно на объектной технологии, но она использует всю мощь реляционной модели. Это позволило не только получить великолепную компактность как кода самой системы, так и хранимой информации, но и высокую производительность, гибкость и простоту использования.