МИНИСТЕРСТВО ОБЩЕГО И ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ РФ
МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ АВИАЦИОННЫЙ ИНСТИТУТ
(ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ)

На правах рукописи

Книжник Константин Александрович

Аспектный подход к созданию объектно-ориентированных систем управления базами данных

Специальность 05.13.11
“Математическое и программное обеспечение
вычислительных машин, комплексов, систем и сетей”

Диссертация на соискание ученой степени
кандидата физико-математических наук

Научный руководитель
Аврамов Игорь Вячеславович

Москва 1998

Официальные оппоненты:

____
____
____
____

Ведущая организация:
Вычислительный Центр Российской Академии Наук.

Защита состоится “ ” __________ 1998г. в ____ часов на заседании специализированного Совета К 053.18.09 в Московском государственном авиационном институте (техническом университете) им. Серго Орджоникидзе.

С диссертацией в виде научного доклада можно ознакомиться в библиотеке МАИ.

Диссертация в виде научного доклада разослана “ ” __________ 1998г.

Ученый секретарь Совета
кандидат физ.-мат. наук, доцент М.В. Ротанина

Оглавление

Глава 1

Введение

Объектно-ориентированные базы данных появились в середине 80 годов как результат осознания того, что реляционная модель плохо подходит для некоторых классов приложений. Все более возрастающая сложность запросов приложений показала необходимость расширения традиционной технологии реляционных баз данных, особенно для таких областей применения баз данных, как системы автоматизированного проектирования (CAD/CAM), системы поддержки принятия решения (CASE), системы автоматизации документооборота, географические и информационные системы. ООСУБД объединяют возможности СУБД с возможностями традиционных языков программирования, таких как C++ и Smalltalk. Было разработано достаточно большое количество ООСУБД, используемых преимущественно в специализированных приложениях. По мере развития ООСУБД, к ним добавились непроцедурный язык запросов и средства администрирования, сходные с предлагаемыми реляционными СУБД. В результате появилась возможность использования ООСУБД для более широкого круга приложений.

С другой стороны, поставщики реляционных баз данных в ответ на требования пользователей о предоставлении возможностей для создания собственных типов и определения полиморфных операций, стали разрабатывать объектные расширения реляционной модели. В результате работы в этом направлении появились гибридные, так называемые Объектно-Реляционные СУБД (ОРСУБД), пытающиеся скомбинировать возможности ООСУБД с традиционной реляционной моделью. Активно разрабатывается объектное расширение языка SQL (SQL3) позволяющего использовать наследование и полиморфизм при проектировании базы данных. Были созданы системы-адаптеры, предоставляющие объектные возможности поверх стандартных реляционных систем. Появление таких систем наряду с совершенствованием ООСУБД подчеркивает общую тенденцию к рассмотрению объектной концепции как ключевого механизма расширения возможностей баз данных.

Появление реляционных СУБД способствовало быстрому росту популярности технологии клиент-сервер. Клиент формирует запрос, отсылает его на сервер, где запрос выполняется и результаты обработки запроса возвращаются клиенту. В последнее время, по мере все большей и большей компьютеризации всех видов деятельности организаций и стремления к интеграции различных обособленных систем в общую среду, усилился интерес к распределенным базам данных, позволяющим организовать доступ к данным, хранящимся на различных серверах. Современные реляционные базы данных предоставляют средства для размещения как различных строк таблицы (горизонтальное разбиение), так и отдельных столбцов (вертикальное разбиение) в разных узлах сети. ООСУБД предоставляют гораздо большую гибкость для разработки распределенных приложений, так как объект является естественной единицей управления размещением данных и многие ООСУБД предоставляют средства для динамической миграции объектов в места частого использования. Поддержка распределенных данных оказывает существенное влияние на архитектуру СУБД в целом и на используемые для управления базой данных алгоритмы.

Основной особенностью ООСУБД является предоставление разработчику приложения всех средств современных объектно-ориентированных языков программирования для доступа к базе данных. При этом, с самого начала развития ООСУБД, наблюдалась тенденция использования хорошо известных языков программирования, таких как C++, Smalltalk, CLOS для взаимодействия с базой.

Для того, чтобы ООСУБД могли быть использованы для широкого класса приложений необходимо сделать их расширяемыми, т.е. предоставить разработчику возможность определять собственные методы управления доступом к объектам, отвечающие потребностям конкретного приложения. Для разных классов программных систем могут потребоваться совершенно противоположные подходы к организации совместного доступа к данным. Так, если каждое приложение работает с базой данных независимо от других приложений, то в этом случае наиболее адекватной моделью является классическая модель, основанная на понятии сериализуемых транзакций. При этом каждое приложение работает с базой данных как бы в монопольном режиме. Другой подход требуется, например, для системы автоматизации документооборота в офисе. В этом случае данные обрабатываются совместно несколькими приложениями и СУБД должна не изолировать изменения, вносимые одним приложением, от других приложений, а предоставлять средства для оповещения и рассылки изменений всем заинтересованным приложениям.

Объектно-ориентированные системы ставят перед своими разработчиками множество различных вопросов. К сожалению, для ООСУБД не существует такой хорошо разработанной теоретической базы, как для реляционной модели. Задачи в которых используются ООСУБД настолько различны по своим требованиям, что практически невозможно разработать универсальную систему, которая предоставляет эффективные решения для всех возможных запросов. Основным преимуществом и одновременно основным недостатком РСУБД является их универсальность, т.е. независимость модели СУБД от модели, используемой приложением. Поэтому программист, разрабатывающий приложение для работы с реляционной базой, часто вынужден работать сразу с несколькими моделями данных, использовать разнородные понятия и тратить много сил на преобразование данных из одного представления в другое. Выбор объектно-ориентированного языка в качестве языка задания и манипуляции данными в СУБД позволяет использовать одну и ту же модель и понятия для проектирования и реализации как базы данных, так и приложения с ней работающего. Однако при этом требуется, чтобы модель, предоставляемая ООСУБД, была достаточно общей и гибкой для удовлетворения нужд различных классов программ.

Основной целью ООСУБД является предоставление программисту прозрачного доступа к базе данных. Прозрачность в данном случае означает, что код, работающий с базой данных, ничем (или почти ничем) не отличается от кода, работающего с обычными данными в оперативной памяти. Таким образом, при работе с базой данных можно воспользоваться стандартными библиотеками и использовать единый код для работы с сохраняемыми и временными объектами. Такой подход в наибольшей степени позволяет реализовать парадигму повторного использования кода, сокращает размер приложения и делает его более понятным.

Однако, работа с базой данных затрагивает аспекты, которые не отражены в универсальных языках программирования: синхронизация доступа, атомарность внесения изменений, восстановление и откатка. Эти аспекты можно реализовать путем создания библиотеки специальных функций, но в этом случае нарушается прозрачность доступа к базе. В данной работе предложен альтернативных подход, основанный на использовании метаобъектного протокола (МОП), позволяющего вынести код, отвечающий за реализацию аспектов доступа к базе данных, на уровень метаклассов (т.е. классов описателей классов). К сожалению, далеко не все языки программирования обладают метауровнем, т.е. средствами для переопределения семантики конструкций языка. Поэтому представляет особый интерес исследование возможности реализации МОП в языках, не поддерживающих метаобъектный протокол.

Целью работы является разработка принципов проектирования универсальных объектно-ориентированных СУБД, позволяющих эффективно организовать хранилища объектов для различных классов приложений. Для этого в работе исследуются возможности и особенности существующих ООСУБД и на базе этого анализа строится метамодель для управления доступом к объектам в СУБД. Реализация ООСУБД, построенной на основе этой метамодели, велась в соответствии со следующими требованиями:

1. Возможность реализации в рамках данной метамодели различных стратегий управления доступа к объектам, удовлетворяющих требованиям различных классов приложений.

2. Наличие базовых метаклассов, реализующих наиболее распространенные стратегии управления объектами и позволяющих легко определять собственные стратегии, используя наследование.

3. Наличие прозрачных интерфейсов с базой данных для различных объектно-ориентированных языков, и в первую очередь для C++. При этом предполагается использование стандартных средств разработки для данного языка (т.е. в язык не вносятся расширения, требующие использования специальной среды программирования). Особое внимание должно уделяться обеспечению всестороннего статического и динамического контроля при работе с объектами базы данных. В интерфейс к СУБД должны входить классы для поддержки платформно-независимой модели параллелизма, позволяющей наиболее эффективно использовать возможности современных многопроцессорных систем.

4. Использование эффективных алгоритмов для объектного сервера, обеспечивающих высокую степень масштабируемости и надежности. Особое внимание при проектировании сервера должно уделяться распараллеливанию процесса выполнения объектных запросов различных клиентов.

5. Поддержка распределенной работы. Система должна обеспечивать работу практически произвольного числа клиентов, расположенных на различных узлах сети, с несколькими серверами объектов. При этом для функционирования как клиентов, так и серверов могут одновременно использоваться различные платформы. Прозрачность доступа к объектам, расположенных на различных серверах, должна дополняться возможностью явного помещения объекта на заданный сервер.

6. Возможность непрерывной работы в течении длительного периода времени без остановки системы, а также наличие средств восстановления при аппаратных и программных сбоях различных уровней. В частности необходимо наличие механизма глобальных транзакций, позволяющем атомарно зафиксировать изменения на различных серверах в сети.

7. Поддержка изменения схемы базы данных при наличие работающих с базой данных приложений. При этом, принимая во внимание большой объем хранимых данных, требуется обеспечить отложенное преобразование объектов в новый формат по требованию (т.е. при загрузке экземпляра объекта клиентом).

8. Модульная структура объектного сервера, позволяющая просто создавать различные конфигурации, оптимизированные под требования конкретных приложений, а также исследовать эффективность различный алгоритмов, управляющих работой сервера ООСУБД.

Глава 2

Применение объектной технологии к СУБД

2.1 Ограничения традиционных реляционных СУБД

Для большинства существующих систем управления базами данных характерна жесткая связь СУБД с поддерживаемой ее моделью данных. В частности реляционные СУБД реализует реляционную модель данных. Реляционная модель определяет:

- единую концепцию структуризации данных, называемую отношением (или таблицей) для представления всех данных в базе,
- набор операций над отношениями, называемый реляционной алгеброй (эти операции включают выборку, проекцию, объединение, пересечение...),
- ряд ограничений, например значения колонки таблицы должно быть атомарным.

Реляционные СУБД кроме того предоставляют:

- набор стандартных типов данных, которые могут быть использованы в колонках таблицы (CHAR, DATETIME, FLOAT...),

- непроцедурный язык манипулирования данными, который может быть определен в терминах реляционной алгебры (SQL),
- реализацию этого языка.

Типичный сценарий работы с РСУБД выглядит следующим образом: приложение посылает SQL запрос РСУБД, который определяет критерии выборки данных из различных таблиц базы. В результате выполнения запроса формируется таблица результатов (таблицы реализованы в SQL как мультимножества строк), которая пересылается обратно программе - клиенту. Извлеченные из базы данных строки должны быть отображены на переменные или другие структуры данных языка, на котором написано приложение. Программист затем производит различные манипуляции над этими структурами данных, возможно изменяя их. Если изменения должны быть занесены в базы, программа формирует SQL оператор для модификации таблиц базы данных используя значения переменных приложения для формирования соответствующих аргументов оператора обновления. Так как возможность представления сложных данных и определения операций над ними в РСУБД сильно ограничены, то семантика данных в основном определяется в прикладной программе (даже несмотря на то, что большинство современных РСУБД предоставляют средства для определения процедур хранимых на сервере и для определения всевозможных ограничений целостности).

Такая схема взаимодействия с СУБД оказалась очень эффективной для большинства основных приложений для бизнеса, однако при попытки применить РСУБД для более сложных (в смысле модели данных) приложений, таких как САПР, ГИС, а также для работы с мультимедиа данными, стало ясно, что для удовлетворения потребности этих приложений необходимы расширения традиционных РСУБД.

Одним из элементов, требующих расширения, является набор типов поддерживаемых СУБД. Набор встроенных типов, поддерживаемых большинством современных РСУБД, (целые, символьные строки...) и операций над ними (+, -, среднее арифметическое) был определен запросами простых бизнес приложений. Однако для большого класса приложений такого набора типов недостаточно. Например, в географических приложениях часто необходимо иметь такие типы как точка, линия, полигон и операции над ними, включающие пересечение, определение расстояния и проверку принадлежности. В научных приложениях может потребоваться использование комплексной арифметики, работа с массивами и матрицами. Бизнес приложения так же нуждаются в специализированных типах: например, денежный тип, хранящий код валюты, или тип для представления даты, учитывающий при операция рабочие и выходные дни или предполагающий что все месяцы содержат по 30 дней. Не имея возможности определять свои собственные типы, программист вынужден эмулировать их с использованием стандартных типов предоставляемых СУБД. Однако такой подход ведет к существенному увеличению сложности программы, увеличению ее размера и деградации производительности. При этом попытка изменить представление типа требует подчас полного переписывания приложения. Попытки решить эту проблему добавлением к СУБД различных нетрадиционных типов, специфичных для отдельных областей, страдает отсутствием общности и не способны решить проблему в целом.

Второй областью, где требуется расширение возможностей РСУБД, является поддержка структурированных данных. При моделировании практически любой системы приходится иметь дело с очень сложными взаимосвязями различных компонент системы. Классическим примером является САПР, где проектируемое изделие состоит из множества деталей, с каждой из которых ассоциирован список свойств. Для работы с такими сложными структурами приложения часто используют графы, вершинами в которых являются компоненты сложной структуры, а ребра представлены указателями. Однако так как традиционные СУБД не поддерживают никаких других способов отображения структурирования данных кроме отношений, то либо для сохранения сложных объектов в базе приходится создавать множество таблиц для представления связей между объектами, либо хранить комплексный объект как не интерпретируемую базой данных строку байтов (так называемые большие двоичные объекты – BLOB). Оба подхода имеют серьезные недостатки. При первом подходе приходится искусственно разбивать объект на группы атрибутов, разбросанных по многим отношениям. Более того, манипулирование такими объектами оказывается громоздким и неэффективным. Вопервых, так как нельзя объяснить СУБД, что все эти строки из различных таблиц представляют из себя один объект, то обычно одна операция над объектом требует выполнения нескольких реляционных операций. Во-вторых, эти реляционные операции плохо поддаются оптимизации, так как обычно включают в себя большое число соединений, а в результате извлекают только очень небольшую часть данных из каждого отношения, в отличие от операций ориентированных на работу с множествами, для которых и были созданы реляционные СУБД. При выборе второго подхода не возможно использование механизмов базы данных для доступа к компонентам объекта запакованного в строку байт. К тому же, СУБД не может быть использования для проверки типов и для выполнения полезных административных функций с компонентами такого объекта. И наконец, программист вынужден иметь дело с задачей преобразования представления структуры в памяти (граф) в запакованную форму, пригодную для сохранения в базы данных и обратно.

Даже в традиционных бизнес приложениях, часто желательно иметь возможность определять обобщенные категории объектов. Так отношение СЛУЖАЩИЙ может иметь поля ИМЯ, ВОЗРАСТ, ЗАРПЛАТА, так же как и отношения "выведенные" из него: МЕНЕДЖЕР, АГЕНТ, ВРЕМЕННЫЙ РАБОЧИЙ. Использование наследования при определении новых отношений, позволяет точнее отобразить связь между различными объектами, упростить определение новых сущностей с использованием уже имеющихся. Наиболее полезно использование наследования при наличии абстрактных операций, т.е. таких операций которые могут быть применены для всех объектов, принадлежащих соответствующей категории. Например, можно было бы определить процедуры обрабатывающие данные обо всех служащих, независимо от того, кем конкретно является данный служащий: менеджером, агентом по продажам и.т.д.

2.2 Обзор проблемы и возможные пути ее решения

СУБД существуют для предоставления постоянного хранилища для данных программ. "Сохраняемость" это свойство объекта, определяющее срок существования объекта. В большинстве языков программирования данные присутствуют в памяти только до момента завершения программы. В таких языках для сохранения объекта между запусками программы, программист должен написать код сохраняющий объект либо напрямую в файл, либо используя СУБД. Как было показано в предыдущем параграфе существует разрыв между требованиями приложения, работающего со сложными структурами данных, и возможностями СУБД обеспечить сохраняемость для этих структур. Этот разрыв еще более увеличился по мере роста популярности объектно-ориентированных языков программирования, которые предоставляют дополнительные возможности для определения и манипуляции сложными структурами при отсутствии аналогичных возможностей в РСУБД.

Для решения этой проблемы предпринимались попытки разработать более адекватный, прозрачный интерфейс между языком программирования и СУБД. В этом случае модель базы данных является расширением одного или нескольких языков программирования. Под прозрачностью интерфейса здесь понимается возможность работать с сохраняемыми объектами базы данных как с обычными объектами языка. Отсутствие прозрачного интерфейса к СУБД приводит к усложнению программы и уменьшению производительности:

- Программист сам должен следить когда следует загрузить, а когда сохранить объект в базу.
- Программист должен написать код для трансляции объекта из представления в базе данных в представление в оперативной памяти и обратно.
- Программист сам должен проводить контроль типов. Большинство современных языков программирования имеет продвинутую систему типизации, однако когда данные читаются из внешнего хранилища (файла или базы данных) контроль типов средствами языка обычно не возможен.
- Возможности предоставляемые СУБД для доступа к данным могут быть не адекватны для метода доступа используемого приложением. В результате может быть получен существенный проигрыш в производительности.

Естественно, что при достаточной схожести моделей данных языка программирования, используемого для доступа к данным, и модели СУБД, предоставляющей этот доступ, можно получить значительное увеличение производительности системы в целом. Еще боле существенно то, что при наличии единой модели данных для приложения и СУБД, существенно упрощается и ускоряется разработка системы, так как на всех стадиях проекта: анализ, дизайн, реализация и сопровождение работа ведется в рамках одной модели с использованием единых понятий. В идеале прозрачный интерфейс должен обеспечивать следующие возможности:

- СУБД осуществляет автоматическое преобразование объектов из формата базы данных в представление языка программирования.

- Программист может писать код для работы с сохраняемыми объектами из хранилища точно также, как и обычный код, без изучения каких либо новых концепций.

- Стандартные библиотеки классов могут быть использованы без изменений (или без существенных изменений) для работы с сохраняемыми объектами.

Самые ранние исследования "языков доступа к базе данных" пытались решить проблемы несоответствия между моделями приложения и СУБД путем интеграции языка программирования и СУБД. Эти языки позволяли некоторым специальным типам данных языка программирования быть сохраняемыми (например тип "отношение" в языке Pascal/R [1], "entity types" в ADAPLEX [2]). Эти языки также предоставляют богатые возможности для задания запросов для выборки множеств объектов из базы данных. Также эти языки включали поддержку транзакций, позволяя осуществлять контроль над параллельным доступом к данным. Однако эти языки все еще имели две системы типов (и два адресных пространства) - для обычных объектов в оперативной памяти и для сохраняемых объектов в базе данных, и программист был должен реализовывать функции преобразования между ними.

Более поздние работы над "языками программирования поддерживающими сохраняемость" в основном велись по принципу "ортогональной сохраняемости" (orthogonal persistence), т.е. такой модели, при которой любой экземпляр любого типа данных языка программирования мог быть сохранен в базу данных, а не только объекты выделенных типов. Результатом применения этого принципа была разработка таких языков программирования с поддержкой сохраняемости и менеджеров объектов для них как, Trellis/Owl [3] и Galileo [4].

Ранние работы над решением этой проблемы со стороны СУБД в основном сводились к расширению реляционной модели. Так например, в СУБД добавлялась поддержка новых типов, таких как "точка", "область". Аналогично проблему сложных составных объектов пытались решить также оставаясь в рамках реляционной модели. Например, было предложено расширение языка SQL предоставляющее возможность построения запросов, учитывающих иерархическую структуру данных. Позднее, реляционная модель была расширена для более точного отображения иерархический структур введением вложенных отношений (или отношений не в первой нормальной форме), то есть таких отношений, атрибуты которых могут сами быть отношениями. Реляционная алгебра была соответственно расширена для обеспечения возможности доступа к таким иерархическим отношениям [5, 6, 7]. Так например, разработчик первоначальной реляционной модели E. F. Codd, предложил расширение реляционной модели RM/T [8], которое включает в себя поддержку подклассов.

Еще одним параллельным направлением исследований, было изучение новых моделей данных и СУБД для них, которые будучи часто основаны на реляционной технологии или реляционных принципах, имели ряд по крайне мере внешних отличий с реляционной моделью. Примерами таких моделей были модель “Сущность-Отношение” [9] и семантические модели данных, такие как TAXIS [10], SDM [11] и Daplex [12].

К относительно недавним исследованиям относятся объектно-ориентированные базы данных. ООСУБД объединяют возможности обычных СУБД с свойствами объектно-ориентированных языков программирования, таких как Smalltalk [13]. Хотя некоторые ранние работы над ООСУБД рассматривались как расширение исследований семантических моделей данных, большинство работ над ООСУБД посвящено объединения технологии баз данных с технологией языков программирования. ООСУБД предоставляют наиболее адекватные и эффективные средства для решения проблем упомянутых раннее: создание специфических типов, представление сложных взаимосвязей объектов, наследование. К данному моменту существует ряд коммерческих ООСУБД, некоторые из которых являются вполне зрелыми продуктами.

И наконец новый класс СУБД, объектно-реляционные СУБД, относится к наиболее поздним работам в этой области. ОРСУБД являются с одной стороны результатом работ по расширению реляционных СУБД, а с другой стороны – следствием доведения ООСУБД до уровня коммерческих продуктов. ОРСУБД пытаются объединить возможности объектно-ориентированных и реляционных СУБД. В то время как, многие ООСУБД предоставляют язык запросов только, как вспомогательное средство для совместимости и администрирования данных, ОРСУБД используют реляционные запросы как основной механизм доступа, с добавлением объектов (или абстрактных типов данных) как вспомогательной сущности, которая может быть использована для обобщения реляционных таблиц и типов используемых в них. Появление ОРСУБД наряду с поддержкой непроцедурного языка запросов большинством коммерческих ООСУБД свидетельствует о начале слияние реляционной и объектной технологии доступа к базам данных. Более того, появление этих двух классов продуктов, подчеркивает значение концепции объекта, как ключевого механизма расширения возможностей СУБД.

2.3 Свойства объектно-ориентированных СУБД

Первоначально понятие объекта было использовано в языке программирования Simula [14]. Однако, объектно-ориентированная парадигма не рассматривалась как новый подход в программировании до появления языка Smalltalk [13]. В настоящее время создано множество объектно-ориентированных языков и некоторые из них стали очень популярными. И хотя ведутся многочисленный дискуссии по поводу списка свойств, которым должен удовлетворять язык, чтобы назваться объектно-ориентированным, можно выделить некоторые базовые понятия общие для любой объектной модели. Практически все объектные модели построены на базе двух основных понятий: объекта и сообщения. Объект является самодостаточной сущностью и состоит из

- атрибутов, отражающих внутреннее состояние объекта, которые могут содержать ссылки на другие объекты;
- набора операций с объектом (иногда называемого протоколом), который образует интерфейс объекта с внешним миром.

Все взаимодействия с объектами происходят посредством посылки сообщений объекту с целью выполнить одну из операций над объектом, входящую в его интерфейс. Таким образом, посылка сообщения может рассматриваться как форма косвенного вызова процедуры, только вместо указания имени процедуры идентифицируется объект – получатель и сообщение адресуется ему. Объект получивший сообщение вызывает одну из своих внутренних процедур, называемых методами для исполнения требуемой операции.

В типичной объектно-ориентированной системе объекты с одинаковой структурой объявляются принадлежащими определенному классу. Часто дополнительно вводится понятие протокола, определяющего интерфейс общий для нескольких классов. Во многих объектных языках класс объекта также является объектом, определяющим методы для изучения и изменения свойств класса.

Пользователь может создавать свои собственные классы и, таким образом, расширить язык для удовлетворения нужд конкретного приложения. Новые классы могут быть образованы из уже существующих с использованием “наследования”. Новый класс в данной ситуации называется выведенным или подклассом для существующего класса, который называется суперклассом или базовым классом. Использование наследования позволяет определять протоколы объектов иерархическим способом. Наследование в совокупности с посылкой сообщений предоставляет из себя одну из форм полиморфизма. Полиморфизмом, в данном случае, называется возможность для одного и того же сообщения выполняться по разному в зависимости от класса объекта получателя. Эта возможность позволяет программе в единой манере работать с объектами различных классов.

В объектно-ориентированных языках каждый объект имеет определяемый системой уникальный идентификатор. Отношение между двумя объектами представляется занесением идентификатора второго объекта (ссылки на объект) в один из атрибутов первого. В связи с интенсивным использованием ссылок при объектно-ориентированном подходе, возникает необходимость в инструменте гарантирующем целостность ссылок в процессе создания и удаления объектов. Для этого в большинстве современных объектно-ориентированных языках, кроме разве что C++, используются сборщики мусора. В таких языках нет явного способа удаления объектов, вместо этого память объекта автоматически освобождается сборщиком мусора, после того как не остается ссылок на этот объект.

Ниже отмечены основные достоинства использования объектно-ориентированного подхода при разработке программной системы:

- Объекты представляют наиболее адекватный способ описания сущностей моделируемой системы, поэтому использование объектов одновременно упрощает и улучшает дизайн системы.

- Использование объектов и сообщений способствует модульному подходу к разработке системы, так как в идеале реализация одного объекта не зависит от реализаций других объектов. Недоступность внутреннего состояния (атрибутов) объекта для остальной части приложения повышает надежность программы.

- Использование классов и наследования обеспечивает простой и выразительный способ для представления отношения между различными компонентами системы.

Объектные системы управления базами данных [15, 16] основаны на тех же принципах, что и объектно-ориентированное программирование, но одновременно имеют ряд дополнительных свойств необходимых для управления большими, распределенными, хранилищами объектов. К этими свойствам относятся обеспечение параллельного доступа к данным, поддержка целостности базы данных и устойчивости к сбоям, эффективный доступ к очень большим объемам информации во вторичной памяти.

Основную идею объектных баз данных можно в двух словах сформулировать так: ОСУБД это СУБД которая сохраняет не просто данные, а объекты в смысле вышеизложенного определения: замкнутая совокупность из данных и ассоциированных с ними методов. По сравнению с реляционными базами данных типичная ОСУБД имеет следующие отличительные особенности:

- ОСУБД поддерживает определяемые пользователем структуры данных и операции над ними (классы и методы), вместо того чтобы ограничивать пользователя только встроенными типами и фиксированными операциями над ними.

- ОСУБД поддерживают понятие идентификатора объекта. Это означает, что объект имеет идентификатор не зависящий от значения его атрибутов, который может быть использован для ссылки на объект из других объектов.

В настоящее время для ОСУБД нет стандартной объектной модели и стандартного языка, занимающих тоже положение, что реляционная модель и язык SQL для реляционных баз данных. Над выработкой таких стандартов сейчас ведется работа группой ODMG (Object Database Management Group) образованной консорциумом поставщиков ОСУБД. Также ведется работа в рамках ANSI и ISO стандартов над объектным расширением языка SQL (SQL3). Принятие SQL стандарта содержащего объектные расширения безусловно сыграет важную роль в определении направлений дальнейшего развития ОРСУБД.

Согласно "Манифесту объектно-ориентированных систем управления базами данных" [17], просто исходя из своего названия ООСУБД должна удовлетворять двум критериям: (а) быть объектно-ориентированной и (б) являться СУБД. Согласно критерию (а) [17] ООСУБД должна обладать восемью обязательными свойствами: сложные объекты, уникальные идентификаторы объектов, инкапсуляция (сокрытие реализации), типы или классы, наследование, переопределение методов совместно с поздним связыванием (полиморфизм), расширяемость и вычислительная полнота. Критерию (б) соответствует пять основных требований: сохраняемость, управление вторичной памятью, параллелизм, способность к восстановлению и непроцедурный язык запросов. К возможностям, которые не были признаны обязательными в [17], относятся следующие: виды представления данных (views), административные утилиты, ограничения целостности, эволюция схемы. Рассмотрим эти требования более подробно.

2.3.1 Сложные объекты и идентификаторы объектов

ООСУБД предоставляет возможность пользователю конструировать объекты со сложным состоянием, то есть атрибуты которого содержат ссылки на другие объекты (наряду с атрибутами содержащими значения: числа или строки). Для корректного представления семантики приложения необходимо также отражать сложные взаимосвязи между объектами. Кроме того, ООСУБД должна предоставлять конструкторы для построения сложных объектов, таких как "множество" или "список", из других объектов. В манифесте [17] отмечается, что такие конструкторы должны быть применимы к объектам любых типов, в том числе и к другим сложным (составным) объектам, образуя вложенные структуры.

Сложными называются объекты построенные с помощью конструктора из более простых объектов. Простейшими типами объектов (называемыми также атомарными, то есть не имеющими подкомпонент) являются такие типы как число, символ, логический тип. Типичными представителями сложных объектов являются структуры, множества, списки и массивы.

Говорят что объект имеет идентификатор если существует способ сослаться на объект, независимо от его текущего состояния [18]. Этим ООСУБД коренным образом отличаются от реляционных СУБД, где ключ, по которому производится доступ к данным, является частью кортежа. Манифест [17] полагают наличие у объекта идентификатора необходимым требованием для системы претендующей на название объектно-ориентированная. В тоже время в манифесте баз данных третьего поколения [19] предполагается, что уникальный идентификатор должен присваиваться системой объекту только в том случае, если у объекта нет определенного пользователем первичного ключа.

Основным применением идентификаторов является построение сложных объектов, где они используются для ссылок на различные компоненты объекта. Идентификаторы особенно полезны для реализации концепции разделяемых данных, так как идентификаторы позволяют отличить ситуации, когда два объекта содержат в качестве подкомпонент два одинаковых объекта (то есть имеющих одинаковое состояние и свойства) и когда два объекта содержат ссылки на один и тот же объект.

Как спецификация разработанная ODMG, так и спецификация SQL3 поддерживают понятия сложных значений (не имеющих идентификаторов объектов) и сложных объектов (содержащих ссылки на другие объекты).

2.3.2 Использование абстрактных типов или классов

Манифест [17] требует от ООСУБД поддержки типов или классов. Рассмотрение отличий между этими двумя понятиями выходит за рамки данной работы, поэтому в дальнейшем мы будем понимать под типом или классом нечто, описывающее общие свойства группы объектов. Во многих языках информация о типе объекта используется только во время компиляции и недоступна во время выполнения. Однако для ООСУБД необходимо иметь возможность сохранить информацию о типе объекта в течении всего времени существования объектов, принадлежащих к этому типу. Эта информация необходима для осуществления преобразований при загрузке/записи объекта в базу, а также для контроля целостности ссылок. Согласно требованиям [17], набор типов должен быть расширяемым, т.е. у пользователя должна быть возможность определять свои типы и в системе не должно быть различий в использовании предопределенных типов и типов созданных пользователем.

Понятие абстрактного типа включает в себя инкапсуляцию состояния объекта и его поведения, т.е. сокрытие аспектов реализации протокола объекта от внешнего мира. Эта идея реализуется путем выделения из объекта интерфейсной части, которая доступна пользователям объекта, и реализационной части, которая им недоступна. Это обеспечивает логическую независимость данных, позволяя изменять реализацию объекта не изменяя кода, использующего этого объект. Манифест [17] рассматривает поддержку инкапсуляции, как необходимое требование для ООСУБД, тогда как [19] ограничивается тем, что называет инкапсуляцию "хорошей идеей". Использование этой возможности особенно важно в ситуациях, когда у принадлежащих одному типу объектов может быть несколько реализаций или когда объекты проектируются и реализуются различными группами разработчиков.

Манифест [17] определяет вычислительную полноту языка программирования, как способность написания на нем любой вычислимой функции. В манифесте отмечается, что хотя традиционные языки доступа к базам данным редко обладают вычислительная полнотой (например SQL не обладает этим свойством), в ООСУБД вычислительная полнота часто обеспечивается как следствие использование стандартных языков программирования для манипулирования данными. Вычислительная полнота нужна для гарантирования того, что на языке можно будет описать любое, сколь угодно сложное, поведение объектов. Язык SQL3 обеспечивает расширение языка SQL, обладающее вычислительной полнотой, для предоставления возможности определять операции для типов создаваемых с использованием механизма абстрактных типов SQL3.

2.3.3 Иерархия классов

Еще одним требованием, предъявляемым [17] к ООСУБД, является поддержка наследования, как средства позволяющего воспользоваться при определении состояния и поведения объекта свойствами уже существующего объекта. Используя наследование можно образовать иерархическую структуру классов, делая описание интерфейсов классов и реализацию более компактными в результате выделения общих частей. Манифест [17] различает две формы наследования: иерархия типов и иерархия реализаций. В первом случае случае наследуются интерфейсы объектов, а во втором – внутреннее состояние объектов и реализация их методов.

Множественное наследование является обобщением обычного (одиночного) наследования, позволяя объекту наследовать свойства и операции более чем одного объекта. Таким образом, при множественном наследовании иерархия классов превращается из древовидной структуры в ациклический граф. При использовании множественного наследования возникает проблема неоднозначности при поиски компонент и методов базовых классов. Неоднозначность возникает при наличии в базовых классах компонент или методов с одинаковым именем. Еще одна проблема с множественным наследованием заключается в трактовке ситуации, когда какой либо класс оказывается унаследованным по двум различным траекториям в графе наследования. При этом желаемое поведение часто специфично для конкретного класса. Так например рассмотрим следующие классы:

```
class NamedObject {
    protected:
        char* name;
    public:
        char* get_name() const;
};

class WindowObject : public NamedObject {
    protected:
        int  x; // position within window
        int  y;
    public:
        void draw();
};

class GeografyObject : public NamedObject {
    protected:
        int x; // location at map
```

```
        int y;
    public:
        ...
};

class GeografyObjectView
: public WindowObject, GeografyObject
{
        ...
};
```

В этом примере класс NamedObject определяет интерфейс именованного объекта, WindowObject является общей базой для всех классов отображаемых оконной системой, GeografyObject определяет общую базу для географической объектов, а класс GeografyObjectView является базой для географических объектов умеющих отображать себя в оконной системе. При этом координаты, указанные в классах WindowObject и GeografyObject, относятся к различным системам координат, поэтому необходимо в объекте выведенном из класса GeografyObjectView, будет иметь по два экземпляра переменных x и y. Другая ситуация с классом NamedObject и компонентой name этого класса. Очевидно. что у объекта должно быть одно имя, поэтому в классе GeografyObjectView должен содержаться только один экземпляр класса NamedObject. В связи с отсутствии простой и выразительной модели для представления множественного наследования многие объектно-ориентированные языки либо поддерживают только одиночное наследование (как Smalltalk), либо разрешают множественное наследование только для интерфейсов (Java).

2.3.4 Совместное использование, переопределение и позднее связывание

Манифест [17] требует от ООСУБД поддержки совместно используемых операций (overriding) и переопределенных операций (overloading). Совместное использование представляет из себя форму полиморфизма совмещения (в классификации Л.Карделли и П.Вагнера), и сводится к возможности определения в классе нескольких методов с одинаковым именем, но различающихся по типам передаваемым им параметрам. Переопределение операций относится к полиморфизму включения с использованием наследования и заключается в возможности переопределить в выведенном классе метод базового класса. Для обеспечения этой возможности в объектно-ориентированных языках программирования используется “позднее связывание” (late binding). Предположим, что мы имеем ряд классов, представляющих геометрические объекты: точку, прямую, окружность... Пусть каждый из этих классов реализует интерфейс обобщенного геометрического объекта GeometryObject, имеющего метод Draw. Тогда отображение чертежа, состоящего из различных объектов, можно осуществить циклическим вызовом метода Draw для всех объектов, образующих данный образ:

```
class GeomtryObject {
  public:
    //
    // Абстрактных метод отображения объекта
    //
    virtual void Draw() = 0;
    virtual GeomtryObject* Next();
};

class Point : public GeomtryObject {
  protected:
    int x, y;
    ...
  public:
    void Draw() {
        /* Реализация метода отображения для точки */
        WindowSystem::DrawPoint(x, y);
    }
};

class Line : public GeomtryObject {
  protected:
    int x0, y0;
    int x1, y1;
    ...
  public:
    void Draw() {
```

```
        /* Реализация метода отображения для линии */
        WindowSystem::DrawPoint(x0, y0, x1, y1);
    }
};


class Drawing [
  protected:
    GeometryObject* ListOfGeometryPrimitives;

  public:
    void Draw() {
        /* Рисование чертежа */
        for (GeometryObject* Obj = ListOfGeometryPrimitives;
             Obj != NULL;
             Obj = Obj->Next())
        {
            // Вызов метода отображения для конкретного объекта
            Obj->Draw();
        }
    }
};
```

2.3.5 Сохранение объектов и управление вторичной памятью

Сохраняемым называется объект, продолжающий существовать после завершения создавшего его процесса. Поддержка сохраняемых объектов является неотъемлемой составляющей ООСУБД. Более того, в [17] говорится о том, что в идеале желательна ортогональная поддержка сохраняемости объектов. Ортогональная поддержка сохраняемости объектов означает, что любой объект может стать сохраняемым и это происходит автоматически, без явного копирования, пересылки или преобразования объекта программистом. Обычно эти требования реализуются с использованием понятия достижимости. Для этого в системе выделяется некоторое множество сохраняемых корневых объектов, которые существуют всегда. Любой объект, на который есть ссылки из других сохраняемых объектов становится сохраняемым и существует до тех пор, пока на него есть ссылки из других сохраняемых объектов. Таким образом сохраняемым является любой объект, достижимый из некоторого корневого объекта (то есть существует последовательность ссылок начинающаяся из корня и приводящая к данному объекту). Таким образом, понятие ортогональной сохраняемости тесно связано с проблемой неявного освобождения памяти (сборкой мусора).

В манифесте [17] от ООСУБД также требуется способность управлять базами данных очень большого объема. Для удовлетворения этому требованию, ООСУБД должна включать механизмы управления вторичной памятью, такие как индексирование данных, кластеризация, буферизация, оптимизация запросов. Особо подчеркивается, что эти механизмы должны быть прозрачны для пользователя (то есть прикладной программист не должен писать кода для поддержки индекса или для пересылки данных из внешней памяти во внутреннюю). Из формулировки этого требования [17] не понятно, может ли объектная модель содержать явные конструкции для поддержки кластеров (как это сделано во многих текущих реализациях ООСУБД).

Вопросы управления доступом к вторичной памяти особенно актуальны для приложений работающими с мультимедиа данными, такими как звук, образы изображений, большие фрагменты текстовой информации, так как размер таких объектов обычно очень велик и операции над ними могут потребовать перемещения значительного объема данных. Более того, для мультимедиа объектов предъявляются повышенных требования к времени доступа, так как необходимо соблюдать фиксированную скорость воспроизведения.

2.3.6 Параллельность и восстановление

Одной из причин появления СУБД была необходимость обеспечения параллельного доступа к данным. Поэтому, не удивительно, что в [17] поддержка параллельности включена в список необходимых требований к ООСУБД. Для поддержки параллельного доступа могут быть использованы как классическая модель, основанная на понятии сериализуемых транзакций, так и менее ограничительные альтернативы, необходимые для реализации многих классов приложений, таких как, например автоматизация документооборота.

Различают несколько видов сбоев и поведения системы в случае сбоя.

Сбой может быть вызван ошибками в программе (как в прикладной программе, так и в самой операционной системе) или сбоем в работе аппаратуры. Аппаратный сбой может привести либо к перезагрузке системы с потерей содержимого оперативной памяти, либо к уничтожению части данных во вторичной памяти (например в результате неисправности диска), либо к неправильному поведению системы (сбой в работе процессора). Вероятность возникновения последней ситуации чрезвычайно мала и требует для обнаружения аппаратного и программного дублирования, использования нетривиальных и ресурсоемких алгоритмов. Поэтому такие виды сбоев рассматриваются только в специальных системах, в которых предъявляется повышенные требования к надежности.

Большинство современных СУБД обеспечивает восстановление корректности базы в результате утраты содержимого оперативной памяти (как следствие программного или аппаратного сбоя), а также средства резервного копирования с целью иметь возможность восстановить некоторое предыдущее состояние базы в результате потери данных на внешнем носителе (сбой диска). В распределенных база данных возникает еще одна причина сбоев - разрыв коммуникаций. Так как, в большинстве случаев невозможно отличить разрыв соединения с сервером от недоступности сервера в результате его сбоя, эти типы сбоев обычно обрабатываются единым образом.

Механизмы обеспечивающие параллельный доступ и возможность восстановления часто тесно взаимосвязаны. В большинстве современных СУБД используется понятие транзакции для обозначения атомарной последовательности действий, которые либо должны все быть выполнены, либо результаты выполнения всех из них должны быть аннулированы. Транзакции также используются для управления параллельным доступом к данным. В манифесте [17] отмечается, что для многих новых приложений, для которых в первую очередь и разрабатываются ООСУБД, используемая в традиционных СУБД транзакционная модель не подходит. В частности, транзакции могут быть долговременными, построенными на базе модели взаимодействующих, а не конфликтующих приложений, что означает неприменимость обычного критерия сериализуемости транзакций. Одним из направлений текущих исследований в области баз данных, является разработка новых, более гибких механизмов поддержки параллельного доступа и восстановления данных, основанных на использовании семантики конкретных классов.

Простой пример, приведенный в [20], использует класс ОЧЕРЕДЬ и операции `ПОСТАВИТЬ_В_ОЧЕРЕДЬ` и `ВЗЯТЬ_ИЗ_ОЧЕРЕДИ`. Операция `ПОСТАВИТЬ_В_ОЧЕРЕДЬ`, вставляет новый элемент в конец очереди, а операция `ВЗЯТЬ_ИЗ_ОЧЕРЕДИ` удаляет элемент из начала очереди. С точки зрения стандартного механизма управления параллельном доступом, операции `ПОСТАВИТЬ_В_ОЧЕРЕДЬ` и `ВЗЯТЬ_ИЗ_ОЧЕРЕДИ` являются модифицирующими операциями и требуют от выполняющей их транзакции установки блокировки на запись. Таким образом при выполнении одной из операций, выполнение другой операции будет задержано до окончания первой. Однако, до тех пор пока очередь не пуста, возможно параллельное выполнение двух операций, без взаимного влияния их друг на друга (так как они работают с разными концами очереди). Таким образом, привлечение семантической информации о типе объекта, может привести к увеличению степени параллелизма, по сравнению с традиционными подходами.

Задачи управления параллельным доступом и обеспечения целостности системы в случае сбоя резко усложняются в случае распределенных систем. Это происходит в следствии невозможности получить информацию о глобальном состоянии системы, как результат ограниченной (и часто непредсказуемой) скорости сообщения по линиям связи. Поэтому требуются специальные алгоритмы (типа двухфазной схемы фиксации транзакций) для синхронизации работы в распределенной системе.

2.3.7 Дополнительные возможности ООСУБД

В дополнение к обязательным требованиям к ООСУБД, [17] определяет набор возможностей, которые расширяя функциональные возможность ООСУБД не являются критическими при определении того, может ли система называться ООСУБД. Две из этих возможностей - множественное наследование и особые виды транзакций, уже осуждались выше. Далее мы рассмотрим остальные пункты.

Контроль типов и информация о типе объекта

ООСУБД различаются по степени осуществляемого контроля типов и предоставлении информации о типе объекта. Во многом это зависит от используемой системы типизации. Языки программирования можно разделить на два класса в зависимости от используемой схемы типизации: языки с динамической типизацией (Lisp, Smalltalk...) и языки со статической типизацией (C++, ADA, Java...). Использование той и другой схемы для языка доступа к базе данных имеет свои положительные и отрицательные стороны. Так использование статической типизации позволяет произвести больше проверок на этапе компиляции программы и способствует созданию более надежного и понятного кода. С другой стороны динамическая типизация обеспечивает большую степень гибкости и лучше подходит для систем требующих внесения изменений в схему базы данных при наличие работающих приложений. Вопросы типизации более критичны для объектно-ориентированных СУБД, по сравнению с реляционными, из-за гораздо более развитой системы типов.

Распределенность

Распределенность является ортогональным понятием по отношению к объектно-ориентированности, поэтому [17] не рассматривает распределенность как необходимое свойство ООСУБД. Однако значение распределенных систем все более возрастает и почти все имеющиеся на данный момент коммерческий ООСУБД в той или иной степени поддерживают работу с распределенными объектами. Поддержка распределенности желательна для ООСУБД еще и потому, что объект является естественной единицей распределения информации в сети, по сравнению например, с распределением отдельных строк или столбцов таблиц реляционных баз данных. Большинство существующих СУБД (в том числе и объектно-ориентированных) используют модель клиент/сервер для организации распределенной среды. Сейчас ведутся работы по интеграции архитектуры распределенных объектов CORBA, предложенной OMG, с ООСУБД.

Поддержка версий

Многочисленные приложения, на которые ориентируются ООСУБД, часто включают в себя процесс разработки, и поэтому требуют поддержки работы с различными версиями документов, которые были созданы или находятся в разработке. Многие ООСУБД содержат встроенные средства для поддержки концепции версий для объекта. Однако [17] не рассматривает поддержку версий объектов, как базовую возможность для всех ООСУБД. Для некоторых типов объектов, таких например, как объекты мультимедиа, поддержка различных версий приводит к существенным накладным расходам. Более того, использование версий объектов требует специальной языковой поддержки для выборки объекта с интересующей версией.

2.3.8 Альтернативные направления

Манифест [17] предоставляет разработчикам свободу выбора в отношении некоторых аспектов построения ООСУБД, по поводу которых еще не сложилось однозначного мнения среди различных исследователей. Эти аспекты проектирования СУБД отличаются от необязательных возможностей перечисленных выше, тем, что не ясно, какие из этих альтернатив являются более или менее объектно-ориентированными. Ниже приводится обзор некоторых таких альтернатив.

Парадигма программирования

В манифесте [17] не определяется выбор какой либо парадигмы программирования для ООСУБД. Любая из них (или комбинация нескольких из них) может рассматриваться как допустимый выбор. Традиционной является парадигма императивного программирования. Однако для некоторых приложений (особенно это относится к экспертным системам и системам предназначенным для прямой обработки запросов пользователя), может оказаться полезным использование альтернативных подходов, основанных на использовании правил, схем потоков данных или других парадигм.

Система типизации

Единственным требованием, предъявляемым [17] к системе типизации, является поддержка инкапсуляции. Система типов используемая в ООСУБД может предоставлять другие возможности, такие как параметризованные типы (шаблоны), например `set<T>`, где T обозначает любой тип. Использование параметризованных типов особенно удобно для определения классов - контейнеров. А всевозможные контейнеры объектов активно используются при работе с базами данных.

Другим аспектом, который можно отнести к системе типизации, является выбор типа объектной модели для ООСУБД. Объектные модели можно разделить на два класса: классическую и обобщенную. Классическая модель во многом является подмножеством обобщенной модели, но они основаны на различных метафорах.

Обобщенная объектная модель не делает отличий между получателем сообщения и другими параметрами запроса. В этой модели обработка запроса трактуется как событие, идентифицирующее операцию над объектами, являющимися параметрами данного запроса. Например, метод ПЕРЕВОД(счет1, счет2), осуществляющий перевод денег с одного счета на другой, не принадлежит ни к какому конкретному классу, представляющему банковский счет (таких классов может быть несколько, так как обычно имеется несколько типов счетов). Вместо этого метод ПЕРЕВОД определяется для некоторых пар классов счетов, и при выполнении операции ПЕРЕВОД, объектная система использует информацию о типе обоих аргументов для поиска метода, наиболее подходящего для этой конкретной пары аргументов. В качестве языка, использующего такую модель, в первую очередь следует отметить Lisp. Объектное расширения языка SQL – SQL3, также поддерживает обобщенную объектную модель.

Классическая объектная модель (иногда определяемая как модель, основанная на посылке сообщений) выделяет объект получатель сообщения. В этой модели запрос трактуется как посылка сообщения объекту - получателю с указанием списка параметров сообщения. Метод в данной модели относится к конкретному классу и выполняется в результат получения сообщения объектом, принадлежащему к этому классу.

Однородность

Различные ООСУБД обладают различной степенью однородности по отношению к используемым ими конструкциям. Ключевым моментом в определении степени однородности является вопрос о том, являются ли типы и операций сами объектами или они трактуются специальным образом. На разных уровнях ответы на этот вопрос могут быть разными. Так, например решение о хранение информация о типе в виде объекта на нижнем уровне не зависит о того, является ли тип объектом (first class object) на уровне объектной и языковой модели.

2.4 Обзор существующих ООСУБД

2.4.1 GemStone

GemStone [21, 22, 23, 24] является объектно-ориентированной СУБД разработанной Servio Corporation. Впервые анонсированная в 1987 году, GemStone является самой старой коммерческой ООСУБД на сегодняшний день. GemStone предоставляет интерфейс ко многим языкам программирования, включая Smaltalk-80, Smalltalk-V, C++, C и имеет графическую среду разработки GeODE. Все приложения, независимо от используемого языка, могут одновременно работать с одной и той же базой. В дополнение к этому GemStone использует диалект языка Smalltalk как внутренний язык манипулирования данными, который позволяет осуществлять выполнение методов непосредственно на GemStone сервере. В качестве платформы для GemStone сервера вместе с графической средой и интерфейсами ко всем указанным языкам могут быть использованы рабочие станции и серверы под управлением операционной системы Unix от Sun, HP, IBM, Sequent. Только клиентская версия доступна для различных языков под Windows 3.1, OS/2 и Macintosh.

Объектная модель GemStone практически совпадает с моделью языка Smalltalk [13] (хотя сама система GemStone написана на C и C++). Объекты в GemStone состоят из набора компонент и методов. Каждый объект, сохраненный в GemStone, содержит уникальный идентификатор. В GemStone не поддерживается явное удаление объектов, а вместо этого используется автоматическая сборка мусора для объектов на которые больше нет ссылок. GemStone ООСУБД реализует один из диалектов языка Smalltalk – Smalltalk-DB (прежде называвшийся OPAL), который специализирован для работы с базами данных. Так язык имеет средства для поддержки сохраняемости, параллельного доступа, транзакций и индексирования. Как и обычный Smalltalk, Smalltalk-DB может быть использован для определения классов (язык определение данных - ЯОД), так и для задания методов (язык манипулирования данными - ЯМД). В языке имеется обширная библиотека системных классов. GemStone осуществляет запись в базы данных как состояния объекта, так и его методов и позволяет выполнять эти методы прямо на сервере.

Ниже приведен пример определения класса для GemStone:

```
Object subclass: 'Person'
```

```
    intVarNames: #('priv_age' 'priv_spous' 'priv_children')
    classVars: #( )
    poolDictionary: UserGlobals
    constrants: #[ #[#priv_age, Integer]]
    isInvariant: false.

method Person    // age getter
    age
        ^priv_age %

method: Person
    age: aAge
        priv_age := aAge %
```

Аналогично, класс Part может быть определен следующим образом:

```
Object subclass: 'Part'
    intVarNames: #('privpartID' 'privpartName' 'privsubParts'
                   'privisPartOf' 'privgraphic')
    classVars: #( )
    poolDictionary: UserGlobals
    constrants: #[ #[#privpartId, Integer],
                   #[#privpartName, String]]
    isInvariant: false.
```

Строка intVarNames в описании класса определяет список имен компонент объекта. В отличии от стандартного определения языка Smalltalk, где компоненте класса можно присвоить любой объект, в Smalltalk-DB допускается введение ограничений на класс объекта, который может быть присвоен компоненте. Так например, в классе Part переменной privpartId можно присвоить только целые числа, а переменной privpartName – только строку символов. Механизм ограничений на значения компонент не является общим средством контроля типов, а используется только для более эффективного размещения компонент объекта в базе данных и возможности построения индекса по таким компонентам.

Для поиска объектов в контейнерах активно используется возможность языка Smalltalk работы с блоками (последовательностью операторов) как с обычными объектами. Если еще учесть, что в языке Smalltalk блоку можно передать значения параметров, то мы получаем очень мощное и удобное средство поиска объектов. Так класс множество может определять метод Select (выбрать) принимающий в качестве своего параметра блок, задающий критерий поиска объекта. Тогда имея объект типа множества aggregateParts, содержащий в качестве компонент детали, входящие в некоторый узел, поиск объекта с заданным именем в этом множестве записывается одной строчкой кода:

```
aggregateParts select: [:aPart | aPart partName = 'AND gate']
```

GemStone также предоставляет средства для оптимизации доступа к объектам. Базовый класс Object определяет метод cluster, который наследуется всеми объектами. Реализация этого метода в классе Object просто осуществляет привязку объекта к странице на диске. Для объединения нескольких объектов в кластер необходимо определить специальный метод для индивидуального класса объектов. Например следующим образом можно определить кластер для класса Part:

```
method: Part
    cluster
        super cluster.
        privpartID cluster.
        privpartName cluster.
        privsunParts cluster.
        privisPartOf cluster.
        privgraphic cluster.
        ^false %
```

Посылка сообщения cluster объекту myPart класса Part, вызовет привязку самого объекта myPart к конкретной странице на диске (“super cluster” вызывает метод cluster класса Object, который и делает это), а затем осуществит добавление в этот кластер (привязку к той же или близлежащей странице) объектов, адресуемых из компонент объекта myPart. Так посылка сообщения “privpartName cluster” предписывает объекту класса String, на который ссылается компонента privpartName, разместить себя вблизи объекта myPart.

GemStone построен в соответствии с архитектурой Клиент/Сервер и состоит из следующих компонент:

- Серверные процессы (менеджеры объектов) называемые Gem. Каждый серверный процесс осуществляет выполнение методов определенных на Smalltalk-DB и осуществляет выполнение запросов. Gem сервер содержит как объектную, так страничную кэш. Клиенты подключаются к серверу либо с помощью компоновки (linking), либо открывая сессию на сервере.

- Stone процесс, выполняющий роль монитора для серверов. Stone присваивает идентификаторы вновь созданным объектам, открывает и закрывает клиентские сессии, координирует процесс фиксации транзакций на различных серверах и отвечает за восстановление после сбоя.

- Страничный сервер. Страничный сервер обеспечивает доступ Gem серверов к локальным и удаленным файлам. Этот сервер поддерживает распределение базы по различным узлам сети. Для повышения производительности возможна репликация данных из удаленного хранилища на локальном сервере.

- Приложения (клиенты). Приложения пишутся с использованием либо имеющихся интерфейсов к языкам программирования Smalltalk, C, C++, либо с помощью среды разработки GeODE, описанной ниже. Приложение связано с Gem сервером и либо может быть скомпоновано совместно с сервером, либо выполняться как отдельный процесс.

Из этих компонент могут быть получены различные конфигурации, оптимизированные под требования конкретного приложения. Например, менеджер объектов Gem, страничный сервер и Stone монитор могут быть размещены на одной машине, образуя сервер-ориентированную конфигурацию. Перенеся менеджер объектов Gem на машину клиента, мы получим конфигурацию с “сильным” клиентом. Конфигурация GemStone может включать различные компьютеры в сети и клиенты могут одновременно работать с базами данных, расположенными на различных узлах. GemStone также поддерживает репликацию данных, позволяя держать копии часто используемых объектов на различных серверах.

Так как методы могут быть реализованы на языке Smalltalk-DB, разработчики приложений могут распределять вычисления между сервером, выполняющим реализованные на Smalltalk-DB методы, и рабочей станцией клиента, где выполняется код приложения, написанного с использованием одного из языков Smalltalk, C++ или C. Таким образом, обладая способность хранить и выполнять методы в базе данных, GemStone является активной ООСУБД, позволяя увеличить эффективность обработки запросов, уменьшить количество пересылаемых по сети данных и лучше использовать вычислительные возможности сервера. Хранение методов в базе данных также упрощает администрирование и сопровождение системы. Так например, при хранении бизнес правил в базе данных, нет необходимости в перекомпиляции клиентских приложений при смене этих правил.

Как уже говорилось, GemStone предоставляет интерфейс доступа к базе для языков Smalltalk, C, C++. Интерфейс к языку Smalltalk представляет из себя набор классов, позволяющих приложению общаться с GemStone базой данных почти прозрачным для приложения образом. В этом интерфейсе осуществляется отображение почти всех системных классов языка Smalltalk на классы GemStone сервера. Интерфейс может автоматически генерировать определения Smalltalk классов исходя из определения классов, хранящихся в базе данных, и наоборот, заносить в базу данных определения классов, построенных из классов языка Smalltalk. Имеется две версии интерфейса - одна встроенная, где сервер и клиент выполняются в одном процессе для достижения максимальной производительности, и вторая версия, основанная на механизме вызове удаленных процедур (RPC) для организации взаимодействия клиента с сервером.

Так как язык C не является объектно-ориентированным, то интерфейс для языка C предоставляет из себя набор функций, предоставляющих приложению доступ к GemStone базе данных. Приложение на языке C может осуществлять доступ к данным либо импортировав данные из базы в локальные структуры данных, либо инициировав выполнение метода на сервере. В интерфейсе имеются функции, осуществляющие импорт и экспорт экземпляров объектов произвольного класса.

Интерфейс к языку C++ основан на использовании препроцессора совместно со специальной библиотекой классов и утилитой Registrar, осуществляющей отображении между C++ классами приложения и классами GemStone базы данных. Классы C++ напрямую или косвенно выведенные из сохраняемых классов библиотеке классов GemStone автоматически становятся сохраняемыми. Компоненты классов могут включать встроенные типы языка C++, классы определенные пользователем, классы из библиотеки GemStone (такие как `GS_Object` и `GS_Set`), а также одномерные массивы из этих типов. Заголовочный файл с определениями классов подается на вход Registrar утилиты, которая по определению класса на C++ строит соответствующий описатель класса для базы данных. Также эта утилита генерирует файл, содержащий процедуры преобразования необходимые для обмена объектами между приложением и GemStone базой данных. В интерфейс также входят функции для открытия и закрытия сессии, фиксирования и откатки транзакции, и.т.д. Для доступа к сохраняемым объектам используется два вида указателей: GPointer и DPointer. GPointer представляет из себя идентификатор объекта, ссылка на который автоматически разрешается при обращении. DPointer является обычным указателем C++ (то есть содержит виртуальный адрес объекта в оперативной памяти).

GemStone поддерживает как оптимистический, так и пессимистический протокол управления параллельным доступом. При оптимистическом подходе транзакция имеет дело с персональными копиями GemStone объектов, образующими рабочую область. GemStone пытается записать измененные объекты из рабочей области клиента только при запросе на фиксацию транзакции. В этот момент GemStone проверяет, что за время прошедшее с момента начала транзакции, объекты к которым осуществлялся доступ в процессе этой транзакции (как для чтении, так и для модификации), не были изменены другой зафиксированной транзакцией. Если такие объекты имеются, то GemStone отказывается фиксировать транзакцию, так как транзакция использовала устаревшие экземпляры объектов при внесении изменений. При отказе в фиксации транзакции GemStone оставляет рабочую область без изменений, предоставляя приложению возможности сохранить результаты работы транзакции перед ее откаткой. После этого открывается новая транзакция, и создается новая рабочая область, содержащая последние версии экземпляров объектов. Приложение затем может попытаться еще раз зафиксировать транзакцию, используя новые версии объектов. Транзакции не вносящие в базу данных изменения, всегда могут быть зафиксированы, так как не меняют содержимое рабочей области. Такой оптимистический протокол синхронизации обеспечивает большую степень параллелизма при выполнении различных транзакций, в случае когда вероятность возникновения конфликта между транзакциями достаточно мала.

GemStone также поддерживает традиционную пессимистическую модель управления параллельным доступом, основанную на блокировках. Использование этого протокола снижает степень параллелизма выполнения транзакций, а так же увеличивает нагрузку на сеть в результате посылки синхронизационных сообщений. Однако при пессимистическом подходе транзакция гарантировано будет зафиксирована (если только не возникнет тупиковая ситуация). Приложение может создавать транзакции, использующие один из этих двух подходов, в зависимости от особенностей поведения конкретных объектов.

GemStone предоставляет набор оптимизированных классов для реализации очередей и наборов объектов, минимизирующих количество конфликтов при параллельном доступе. Также имеются классы для реализации индексов, множеств, счетчиков и хэш таблиц, обеспечивающих высокую степень параллелизма. GemStone также поддерживает рассылку сервером уведомлений об изменении объектов клиентам, а также позволяет приложениям посылать сигналы другим приложениям без внесения изменений в базу. Этот сигнальный механизм может быть использован для организации взаимодействия различных приложений.

GemStone также поддерживает механизм версий объектов. Этот механизм предоставляет несколько предопределенных схем использования версий (например, линейные версии), а также позволяет определять специфичные для конкретного приложения схемы. GemStone использует механизм версий для поддержки изменения схемы базы данных.

Для быстрой разработки, реализации и сопровождения приложения совместно с GemStone сервером поставляется среда разработки GeODE (GemStone Object Development Environment). GeODE включает в себя следующие компоненты: менеджер приложений, дизайнер схемы базы данных, дизайнер форм, графический построитель программ, библиотеку классов для разработчика приложений, утилиты для отладки, просмотра и инспектирования програм. Приложения, созданные с помощью GeODE, хранятся в базе данных и могут позваны из программы, написанной на любом языке для которого определен интерфейс с сервером GemStone.

2.4.2 ONTOS DB

ONTOS DB является продуктом корпорации Ontos и представляет из себя ООСУБД основанную на использовании языка C++. В ONTOS DB была сделана попытка дополнить язык C++ такими свойствами СУБД как сохраняемость, совместный доступ к данным, поддержка целостности и наличие непроцедурного языка запросов. ONTOS DB использует архитектуру клиент-сервер и поддерживает физическое распределение база данных между различными узлами сети. ONTOS DB поставляется с C++ интерфейсом, позволяющим использовать стандартные компиляторы C++ для написания приложений работающих с базой данных, и с SQL интерфейсом, предоставляющим средства для доступа к базе данных посредством запросов, как в интерактивном режиме, так и из приложений. Механизмы управления параллельным доступом и транзакциями предоставляют средства для согласованного параллельного доступа к базе данных нескольких пользователей. В дополнении к этому поддерживается разделение данных между несколькими взаимодействующими процессами.

Внутренняя объектная модель ONTOS DB основана на объектной модели языка C++. Описания классов и определения методов даются на языка C++ (с минимальными расширениями). Объекты ONTOS DB состоят из списка атрибутов и функций. Каждый объект имеет 64-разрядный идентификатор объекта, уникальный для базы данных. Атрибут объекта либо содержит непосредственное значение, либо идентификатор другого объекта, что позволяет моделировать сколь угодно сложные связи между объектами.

Взаимодействие с базой данной ONTOS DB осуществляется посредством библиотеки классов. Корневым классом в иерархии классов является класс `OC_Object`. Этот класс является общей базой для всех сохраняемых объектов. В классе `OC_Object` определен конструктор для создания объекта в базе данных и деструктор для явного его удаления (в отличие от GemStone, использующего сборщик мусора для освобождения памяти, в ONTOS DB используется явное удаление объектов). Также в классе `OC_Object` имеются атрибуты определяющие имя объекта и физический адрес объекта в базе данных. Также как и GemStone, ONTOS DB предоставляет набор классов – контейнеров, таких как `OC_Dictionary`, `OC_Set`, `OC_Array` и `OC_List`. Например словари реализуются как хэш таблицы или Б* деревья. Ниже приводится пример определения класса в ONTOS DB:

```
class Person : public OC_Object {
  protected:
    int          priv_age;
```

```
    OC_Reference priv_spouse;
    OC_Set*      priv_children;

  public:
    Person(char* name=NULL, int age=0);
    Person(OC_APL* theAPL);
    virtual OC_Type* getDirectType();

    int          age() const { return priv_age; }
    void         age(int age) { priv_age = age; }

    Person*      spouse() {
        return (Person*)priv_spouse.getReferent(this);
    }
    void         spouse(Person* aPerson) {
        priv_spouse.reset(aPerson, this);
    }

    void         display();
};
```

Приведенных в примере класс Person, объявлен как выведенный из класса `OC_Object`, что делает его сохраняемым ONTOS DB классом. Переменная `priv_children` объявлена, как имеющая тип `OC_Set*`, то есть содержащей прямой указатель (адрес в оперативной памяти) на объект класса `OC_Set`. С другой стороны, компонента `priv_spouse`, объявлена как имеющая тип `OC_Reference`, и содержит абстрактный указатель ONTOS DB (различия между этими двумя типами указателей будут описаны ниже).

В публичной части класса, содержащей описания методов, следует обратить внимание на первые три метода. Первый из них является обычным конструктором объекта языка C++. Второй конструктор используется ONTOS DB для инициализации объекта при чтении с диска и должен быть определен в каждом сохраняемом классе. После конструкторов объявлен виртуальный метод `getDirectType`, который возвращает описание класса объекта в базе данных.

Как и многие другие языки программирования, C++ не имеет стандартного механизма для получения информации об определении класса в момент выполнения программы. Библиотека классов ONTOS DB содержит классы (их можно назвать метаклассами) для описания компонент класса, методов класса и их аргументов, а также всего класса в целом. Эти классы используются для описания схемы базы данных системно-независимым способом и также могут быть использованы внутри приложения для получения информации о классе объекта во время выполнения. Эти классы генерируются автоматически по C++ описанию классов специальной утилитой classify. Классы, используемые для описания методов, позволяют перенести ключевые алгоритмы из приложения на сервер. Поведение приложения впоследствии может быть изменено без перекомпиляции самих приложений. Например программа обработки текстов может поддерживать несколько языков, поместив языково-зависимые функции в базу данных.

Основные аспекты адресации и кластеризации объектов в ONTOS DB во многом совпадают с GemStone. Как и GemStone, ONTOS DB позволяет осуществлять прямой доступ внутри одномерного пространства базы данных с использованием идентификаторов объектов. Объект является единицей адресации. Контейнеры в свою очередь тоже являются объектами. Значения атрибутов объектов, за исключением некоторых специальных случаев, представлены ссылками на другие объекты. Такая стратегия хорошо подходит для реализации сложных структур данных, с многочисленными связями между объектами. Однако для эффективного доступа к компонентам объекта при таком подходе, необходимо использование кластеризации. Как и GemStone, ONTOS DB предоставляет программисту средства явного указания кластера для объекта, а также специальные утилиты для перемещения объектов между кластерами в процессе эксплуатации базы.

ONTOS DB включает поддержку распределенной архитектуры с использованием модели клиент-сервер. В этой модели серверная часть управляет хранилищами данных, а клиентская часть предоставляет приложениям интерфейс для отображения данных в виртуальное пространство процессов. Клиенты и сервер могут располагаться либо на различных узлах локальной сети и соединиться по LAN, или быть запущенными на одном и том же узле и использовать стандартные средства взаимодействия процессов. Третьей компонентой архитектуры является регистрационный процесс Registry. Registry содержит информацию о всех базах данных в сети и их серверах. Этот процесс также поддерживает списки доступа для пользователей. Для установления соединения между клиентом и сервером предусмотрен специальный глобальный сервис Binder, который взаимодействует с Registry для получения по логическому имени базы данных адресов серверов ассоциированных с этой базой.

База данных может быть расположена на одном узле или выть распределена по нескольким узлам сети. Управление каждой базой данных осуществляет первичный сервер. В случае если база данных распределена по нескольким узлам, каждый из этих узлов имеет вторичного сервера для управления локальным хранилищем. Задачей каждого серверного процесса является организация доступа к данным в локальном хранилище и обработка запросов клиентов. В дополнении к этому первичный сервер отвечает за привязку клиента к одному из локальных серверов и осуществляет выполнение всех глобальных для базы данных операций, таких как открытие и закрытие базы, и управление распределенными транзакциями.

Интерфейс между клиентом и сервером задается достаточно небольшим количеством функций и классов. Основными функциями являются открытие и закрытие базы данных, начало, фиксация и откатка транзакций, функции загрузки и сохранения объектов.

В ответ на запрос серверу на загрузку объекта (или группу объектов), клиенту отсылаются экземпляры запрошенных объектов. Эти объекты затем активизируются в приложении клиента и работа с ними ведется как с обычными структурами языка C++. Когда приложение заканчивает работу с объектом, объект деактивируется, память из под него освобождается, и, если объект был изменен, модифицированная версия объекта пересылается обратно на сервер.

ONTOS DB предоставляет низкоуровневые средства управления доступом к объектам посредством менеджеров хранилища. Эти компоненты управляют размещением объекта на диске и в памяти, осуществляют создание и удаление объектов, Менеджеры хранилища также осуществляют трансляцию ссылок `OC_Reference` на объекты. Менеджеры хранилищ представлены в ONTOS DB объектами, выведенными из класса `OC_StorageManager` и оптимизированными под различные типы приложений:

- Стандартный менеджер хранилища `OC_StandardSM` управляет активизацией, кластеризацией и блокировкой индивидуальных объектов. Этот класс используется по умолчанию.

- Групповой менеджер хранилища OC_Group управляет активизацией, кластеризацией и блокировкой фиксированных по размеру сегментов, называемых страничной группой. Страничная группа используется для хранения объектов одного типа. Осиновым преимуществом использования групп является повышение производительности при работе со сложными объектами, представленными в виде множества небольших, ссылающихся друг на друга объектов. Так как при работе с такими сложными объектами скорее всего потребуется загрузить все его компоненты, то эффективнее это сделать одной операцией для всей группы объектов, чем посылать отдельный запрос для каждого объекта – компоненты. Такая ситуация типична для САПР и ГИС приложений.

- Менеджер объектов в памяти OC_InMemorySM используется ONTOS DB для поддержки временных, не сохраняемых объектов, который потом возможно будут преобразованы в сохраняемые. Пользователи не могут создавать такие объекты самостоятельно, но могут получать ссылки на экземпляры, созданные ONTOS DB.

- Менеджер версий OC_VersionSM осуществляет доступ к версиям объектов. Этот менеджер осуществляет только разрешение ссылок и управление конфигурацией, используя методы других менеджеров для доступа к объектам.

- Менеджер внешних хранилищ OC_ExternalSM предоставляет средства для доступа к внешним источникам данных, таким как реляционные или иерархические СУБД, отображения и конвертирования данных из этих источников в объекты. Используя наследование, возможна разработка собственных классов под конкретные требования приложения и внешние хранилища.

Ниже представлен шаблон простейшего приложения на языке C++, работающего с ONTOS DB. В нем проиллюстрированы некоторые ключевые аспекты взаимодействия приложения с ONTOS DB.

```
OC_open("personDB");     // открытие базы данных
OC_transactionStart();  // начало транзакции

// активизация объекта класса Person
```

```
Person *myPerson = (Person*)OC_lookup("Frank");

... // код осуществляющий изменение объекта

myPerson->putPbject();  // деактивизация объекта

OC_transactionCommit(); // фиксация транзакции

OC_close();             // закрытие базы данных
```

Объекты пересылаются между приложением и базой данных операциями активизации и деактивизации. Доступ к корневым объектам осуществляется по имени функцией `OC_lookup`. Объекты базы данных используют идентификаторы объектов для представления ссылок между объектами. Когда объект активизируется, все содержащиеся в нем ссылки на уже загруженные объекты преобразуются в прямые указатели в оперативной памяти приложения на соответствующие объекты. Деактивизация представляет из себя обратный процесс: все прямые указатели преобразуются обратно в идентификаторы адресуемых объектов. Поскольку процесс активизации оказывает большое влияние на производительность системы в целом, то программисту предоставляется контроль над единицей активации: либо единственный объект, либо логический кластер объектов. Способ кластеризации объектов также контролируется приложением. Все операции с базой данных производятся в рамках транзакций для обеспечения корректного доступа к данным при параллельной работе нескольких приложений.

Форма виртуальных ссылок на объекты, используемых при преобразовании объекта при пересылки из базы данных в память приложения, задается программистом при определении класса. При использовании абстрактных ссылок (`OC_Reference`) доступ к объектам осуществляется через вторую степень косвенности и корректность ссылок поддерживается системой. Если объект, на который указывает ссылка, отсутствует в памяти приложения (не активирован), то он будет автоматически загружен при обращении по ссылке. Таким образов активизация объекта прозрачна для программиста, ценой использования лишней степени косвенности. Предполагается, что приложение будет в основном пользоваться ссылками этого типа, так как только они автоматически гарантируют корректность доступа к объектам. Однако имеется возможность использовать и прямые ссылки (адрес в виртуальной памяти), если приложение может гарантировать корректность ссылки. В этом случае при активизации объекта, идентификаторы уже загруженных объектов заменяются на прямые ссылки в памяти. Обращение по таким ссылкам осуществляется без дополнительных накладных расходов и проверок, и в этом отношении прямые ссылки эквивалентны обычным указателям языка С. Однако программист должен убедится, что не использует ссылку на не активизированный объект. Если адресуемый объект отсутствует в памяти, то он может быть активизирован явно с помощью вызова соответствующей функции.

Механизм управления транзакциями ONTOS DB содержит ряд усовершенствований по сравнению с традиционными реализациями. Во первых поддерживаются вложенные транзакции. Каждый уровень вложенности обеспечивает атомарность. Это позволяет достаточно просто реализовать обобщенный механизм откатки. ONTOS DB также поддерживает совместную работу нескольких процессов. Несколько процессов могут участвовать в одной транзакции. Такая разделяемая транзакция объединяет изменения внесенные различными процессами в единую атомарную последовательность изменений, предоставляя процессам в тоже время общую область для обмена информацией. ONTOS DB как и GemStone поддерживает оптимистическую и пессимистическую модель управления параллельным доступом. В дополнении к этому, ONTOS DB позволяет пользователю определять свои схемы синхронизации доступа, обеспечивающие возможность одновременного изменения объекта несколькими процессами.

2.4.3 ObjectStore

Объектно-ориентированная СУБД ObjectStore [25, 26, 27, 28] разработана корпорацией Object Design Inc (ODI). Объектно-ориентированный подход в этой системе объединен с глубоко интегрированными в язык свойствами СУБД, такими как сохраняемость, управление транзакциями (совместный доступ и восстановление), распределенный доступ, ассоциативные запросы и административные утилиты. Система предоставляет интерфейсы для таких языков программирования, как C, C++ и Smalltalk, а также обладает графическим интерактивным дизайнером схемы, инкрементальным компилятором и компоновщиком, отладчиком и графическим интерактивным средством просмотра базы данных. ObjectStore содержит средства расширения языка C++ для работы с множествами.

Объектная модель, предоставляемая пользователям ObjectStore, в основном определяется языком программирования, используемым для взаимодействия с базой. Интерфейс к языку C++ использует объектную модель C++, включающую поддержку иерархии классов, атрибутов объектов, прямых ссылок между объектами и инкапсуляцию представления данных. Одной из основных задач, которую ставили перед собой разработчики ObjectStore, было сведению к минимуму изменений, которой нужно внести в "родные" приложения C и C++ для работы с базой данных ObjectStore. Для этого в ObjectStore предусмотрены различные средства. Например сохраняемость определяется в терминах индивидуальных объектов и переменных, а не на уровне типов данных. Это означает, что объект любого типа C++ может быть объявлен сохраняемым. При этом нет необходимости наследовать специальный класс для обеспечения сохраняемости. Это позволяет функциям написанным для работы с обычными (не сохраняемыми) объектами, также манипулировать объектами из базы данных. Это упрощает процесс адаптации программ для работы с ObjectStore и позволяет воспользоваться коммерческими C++ библиотеками.

ObjectStore предоставляет также набор средств расширяющий возможности языка C++. Одним из таких расширений является поддержка контейнеров объектов, обеспечиваемых на уровне библиотеки классов. Типы контейнеров, предоставляемых этой библиотекой, включают обычные множества, множества с повторяющимися элементами, списки и массивы, определенные как параметрический классы (например класс `os_Set<T>`), и как обычные, не параметризованные классы (например `os_list`). ObjectStore позволяет пользователю определять характер использования контейнеров (например, частоту вставки новых элементов) наряду с различными параметрами настройки, и выбирает наиболее эффективное представление для класса основываясь на этой информации. Для работы с контейнерами предусмотрены всевозможные средства итерирования, просмотра и поиска.

ObjectStore поддерживает двунаправленные отношения между объектами. Отношение представляется парой инвертированных указателей между связанными объектами. Могут быть объявлены отношения типа один-к-одному, один-ко-многим и многие-ко-многим. ObjectStore поддерживает целостность этих отношений. Например, если удаляется какой либо участник отношения, то инвертированный указатель на него автоматически будет установлен в нуль. Возможно и более сложное поведение. Например, ниже приводится пример содержащий шаблон определения отношения между департаментом и работающим в нем персоналом. В этом примере при добавлении сотрудника E в департамент D, поле, определяющее департамент для сотрудника D, будет автоматически изменено для ссылки на D. Аналогично, при переводе сотрудника из департамента D в другой департамент D2 и изменении поля департамента E для данного сотрудника с D на D2, произойдет автоматическое удаление E из списка сотрудников `D->employees` и добавление в `D2->emploees`:

```
class employee {
  public:
    ...
    department* dept
        inverse_member department::employees;
};

class department {
  public:
    ...
    os_Set<emloyee*> employees
        inverse_memner employee::dept;
};
```

Доступ к ObjectStore осуществляется посредством библиотеки интерфейсных функций и классов, совместимого с популярными средами разработки C и C++. ObjectStore также предоставляет расширенную версию C++ (ObjectStore ЯУД), который обладает встроенной поддержкой таких средств работы с базой, как запросы и отношения.

В работе [28] описывается интерфейс ObjectStore к языку Smalltalk. Этот интерфейс обеспечивает работу с базой данных посредством определения ряда новых Smalltalk классов и переопределения существующих классов из системного пакета. Также, как и в случае с C++, при проектировании интерфейса с языком Smalltalk, основное внимание уделялось возможности прозрачного для программиста доступа к базе. База данных создается посылкой сообщения классу ObjectDatabase. Доступ к базе данных всегда осуществляется в рамках транзакций. Далее приводится пример из [28] иллюстрирующий простую программу, открывающую базу данных, создающую объект с информацией о новом сотруднике и добавляющей его в список сотрудников департамента.

```
| db dept emp |
db := ObjectDatabase open:'/company/records.db'.
ODBTransaction transact: [
        dept := db at: 'department'.
        (emp := Employee newIn: db)
                name: 'Fred'
                salary: 1000.
        dept addEmployee: emp
]
```

Свойство сохраняемости объекта в интерфейсе ObjectStore к языку Smalltalk является ортогональным к типу объекта. Это означает, что любой экземпляр объекта произвольного класса может стать сохраняемым и не требуется добавления никакого дополнительного кода для извлечения сохраняемого объекта из базы данных. Объект становится сохраняемым неявно. Объекты созданные во время транзакции автоматически становятся сохраняемыми, если на них имеются ссылки из других сохраняемых объектов по окончанию транзакции. Это позволяет использовать существующие прикладные и системные библиотеки классов с минимальными изменениями.

ObjectStore использует клиент/сервер архитектуру, позволяющую одному серверу обслуживать несколько рабочих станций клиентов. Также возможна работа одного клиента сразу с несколькими базами данных на различных серверах. Сервер ObjectStore предоставляет долговременное хранилище для сохраняемых объектов. Объекты сохраняются в хранилище в том же формате, в котором они присутствуют в виртуальной памяти клиента, что устраняет накладные расходы на преобразование объекта при пересылке объекта с клиента на сервер и обратно. Объекты располагаются на страницах, при этом размер объекта может быть больше размера страницы и пересекать границу страниц. Сервер никак не трактует содержимое страницы, он просто отвечает за пересылку страниц клиенту (от клиента) и запись измененных страниц на диск. Сервер также отвечает за синхронизацию совместного доступа к объектам и за восстановление базы данных после сбоя, используя обычные для

СУБД методы. ObjectStore сервер использует двухфазную схему блокировок, с поддержкой блокировок на чтение и запись для каждой страницы. Механизм восстановление базы данных основан на ведении журнала транзакций с использованием протокола “запись в журнал до записи в базу данных” (write-ahead log). Транзакции включающие более одного сервера координируются с использованием двухфазной схемы фиксации транзакций. Сервер также поддерживает резервное копирование на вспомогательные носители, такие как магнитные ленты.

Для каждого компьютера клиента ObjectStore запускает процесс управления “кэш памятью клиента” – локальной областью для хранения данных отображенных или ждущих своей очереди для отображения в виртуальное пространство памяти приложения. Менеджер кэш памяти всегда выполняется на машине клиента в единственном экземпляре, даже если на компьютере запущено несколько приложений. Коммуникация между сервером и клиентом осуществляется посредством локальной сети, в случае если сервер и клиент находятся на различных узлах сети, или с использованием более быстрых механизмов, таких как разделяемая память или локальные сокеты, при расположении клиента и сервера на одной машине.

Наряду с использованием обычных файлов операционной системы для хранения базы данных, ObjectStore предоставляет возможность использовать специализированную файловую систему под управлением менеджера директорий ObjectStore. Эта файловая система имеет логическую организацию независящую от конкретной используемой операционной системы (и более того может использовать различные файловые системы).

Создатели ObjectStore пытались насколько это возможно предоставить пользователю концепцию единого пространства сохраняемых и временных объектов, вместо того, чтобы заставлять программиста иметь дело с двумя различными пространствами и явно управлять перемещением объектов между двумя этими пространствами. Это означает, что разрешение ссылок, управление кэш памятью, и.т.п., осуществляется самой системой и прозрачно для программиста. Более того, основываясь на требованиях такого класса приложений, как САПР, задачей архитектуры ObjectStore было обеспечение такого же времени разрешения ссылки на сохраняемый объект, как и на обычный объект (естественно, после того как сохраняемый объект был загружен в пространство памяти процесса). Поэтому необходимо было устранить накладные расходы на проверку присутствия сохраняемого объекта в памяти. Для этого в ObjectStore используются возможности современных операционных систем определять специфичные для приложения способы контроля доступа к виртуальной памяти. Такие средства имеются в большинстве современных версиях системы Unix, а также в OS/2 и Windows NT. Использование "отображения виртуальной памяти" выделяет ObjectStore среди других ООСУБД.

Как описывается в [25], интерфейс к виртуальной памяти позволяет ObjectStore устанавливать для страниц виртуальной памяти следующие режимы доступа: запрет доступа, только чтение и чтение/запись. Когда приложение ObjectStore обращается по ссылке к объекту, который отсутствует в памяти клиента (и следовательно для соответствующей страницы запрещен доступ к ее содержимому), аппаратура фиксирует ошибку при доступе и операционная система передает управление обработчику сигналов ObjectStore. Как отмечалось выше, ObjectStore поддерживает на клиенте кэш часто используемых страниц. Когда в приложении происходит обращении к защищенной странице, ObjectStore осуществляет поиск требуемой страницы в кэш памяти и инициирует загрузку страницы с сервера, в случае отсутствие ее в кэш памяти. После того как страница окажется в кэш памяти клиента и права доступа к ней в приложении изменяются на "только для чтение", происходит возврат из обработчика ошибки доступа, повторное обращение по ссылке, которое теперь будет успешным. Последующие обращения для чтения по этому адресу или другим адресам на той же странице больше не будут вызывать нарушений защиты и будут выполняться с такой же скоростью, что и обращения по обычным указателям. Попытка записи по данному адресу опять приведет к нарушению защиты страницы. Обработчик этого прерывания установит блокировку на запись для соответствующей странице на сервере и после этого сменит режим доступа к странице на "чтение/запись". В то время, пока страница находится в кэш памяти без блокировки страницы на сервере, другой процесс может изменить соответствующую страницу, делая устаревшим экземпляр страницы в кэш памяти. Механизм поддержки когерентности содержимого кэш памяти использует механизм оповещения (callbacks) между клиентами для гарантирования того, что транзакция всегда видит последнюю версию страницы.

В этой схеме есть один недостаток: размер базы данных ObjectStore может превышать размер виртуального адресного пространства (4 Гигабайта для большинства современных архитектур), более того при работе одновременно с несколькими базами данных может произойти перекрытие используемых ими диапазонов адресов виртуального пространства. Поэтому ObjectStore не отображает все пространство базы данных на виртуальную память целиком, а вместо этого производится отображение части базы данных, с использованием карты страниц, определяющей соответствие между страницами базы данных и страницами виртуальной памяти. При загрузке объекта, содержащего ссылки на не отображенные страницы базы, происходит выделение для них новых страниц виртуальной памяти. Карта отображения страниц инициализируется в конце каждой транзакции и при старте следующей транзакции заполняется заново. Таким образом, суммарное количество данных, к которым может быть осуществлено обращение в пределах одной транзакции, ограничено размером виртуального пространства процесса.

Несмотря на то, что ObjectStore поддерживает абстракцию единого адресного пространства для сохраняемых и обычных объектов, остаются несколько аспектов которые требуют участия программиста. В частности ссылки между различными объектами можно разбить на несколько классов:

1. указатели из обычных объектов на обычные объекты
2. указатели из объектов одной базы данных в объекты той же самой базы данных
3. указатели из объектов одной базы данных в объекты другой базы данных
4. указатели из обычных объектов на сохраняемые объекты
5. указатели из сохраняемых объектов на обычные объекты

Указатели первых двух видов всегда корректны. Использование указателей оставшихся трех категорий допустимо только внутри транзакции. ObjectStore гарантирует согласованность копий объектов в памяти клиента и в базе данных во время выполнения транзакции. По окончанию транзакции измененные объекты записываются обратно в базу данных и становятся доступными для других приложений. Указатели из сохраняемых объектов на временные объекты не являются допустимыми вне тела транзакции, так как база данных не может гарантировать доступность этого объекта (приложение может удалить этот объект или просто закончить выполнение). Запрет на использование указателей из временных объектов на сохраняемые объекты вне транзакции объясняется тем, что постоянные объекты могут быть удалены другим приложением.

ObjectStore позволяет явно определять кластера объектов для повышения вероятности того, что группы тесно взаимосвязанных объектов, будут вместе пересланы с сервера на клиент, и таким образом будут уже присутствовать в памяти процесса при обращении к ним. Например, в приложении САПР объекты, соответствующие деталям некоторой сборки, могут быть помещены в один кластер. Таким образом, при обращении к объекту представляющему одну из деталей, в память клиента будут загружены все объекты соответствующие деталям сборки и последующие обращения к ним будут происходить без накладных расходов.

В ObjectStore реализован как стандартный механизм транзакций, основанный на двухфазных блокировках, так и дополнительные формы транзакций, необходимые для некоторых классов приложений, такие как долгоживущие и вложенные транзакции, а также механизм оповещения клиентов и другие формы взаимодействия между различными процессами. ObjectStore поддерживает версии объектов и предоставляет возможность получения резервных копий базы данных без ее остановки.

ObjectStore использует метаобъектный протокол (МОП) для получения информации о схемы базы данных в процессе выполнения программы. Не ясно однако, можно ли использовать MOP для изменения различных аспектов объектной модели как в языке CLOS. ObjectStore также поддерживает динамическое создание типов и внесение изменений в схему базы данных в процессе выполнения программы.

Наряду с поддержкой навигации между объектами стандартными средствами языка C++ (или Smalltalk), ObjectStore также предоставляет возможность итераторов и ассоциативных запросов. Поясним это на примере с классом "Служащие":

```
class employee {
    public:
        char* name;
        int   salary;
};
```

```
class department {
  public:
    os_Set<employee*> employess;

    void add_employee(employee *e) {
        employees->insert(e);
    }
    bool works_here(employee *e) {
        return employee->contains(e);
    }
};
```

Для итерации по элементам множества может быть использована конструкция "foreach". Например, следующий цикл увеличивает на 10% заплату всем служащим департамента:

```
department* d;
...
foreach (employee*, e, d->employess)
    e->salary *= 1.1;
```

В отличие от встроенного SQL, который использует свой собственный синтаксис и семантику, отличную от используемого языка программирования, подход к построению запроса в ObjectStore предполагает более тесную интеграцию с языком программирования. Запрос является просто выражением, которое используется как критерий выбора объекта из одного или более контейнеров, и возвращает в качестве результата новый контейнер. Например список всех служащих с зарплатой больше $100,000 может быть получен следующим запросом:

```
os_Set<employee*>  all_employees; // список всех служащих
...
os_Set<employee*>& overpaid_employees =
        all_employees [: salary >= 100000 :];
```

ObjectStore использует специальные скобки `[:   :]` для обозначения запроса. Содержащееся в них выражение является предикатом применяемым ко всем элементам контейнера.

ObjectStore является одной из самых популярных ООСУБД. Одна из версии ObjectStore используется в качестве менеджера хранилища в среде NeXTStep, а другая - в качестве хранилища объектов в среде DOE (Distributed Object Everywhere) для операционной системы Solaris корпорации SunSoft.

2.4.4 VERSANT

ООСУБД VERSANT является продуктом фирмы Versant Object Technology и создана для поддержки многопользовательских приложений в распределенной среде. VERSANT использует обобщенную внутреннюю модель на базе которой строятся интерфейсы к различным языкам программирования. В VERSANT нет специального внутреннего языка определения и манипулирования данными. Первоначально доступ к СУБД осуществлялся через интерфейс к языку C, посредством вызовов функций, предоставляющих доступ к внутренней объектной модели. Интерфейс C++ содержит классы, инкапсулирующие вызовы этих функций. Внутренняя модель является пассивной, в то смысле, что методы объектов не хранятся в базе данных, а содержатся в приложениях, осуществляющих работу с базой. В результате VERSANT не сталкивается с проблемой выполнения методов, реализованных на различных языках программирования, а отвечает только за перемещение объектов между базой данной и приложениями.

Каждому объекту VERSANT присваивается логический идентификатор объекта (loid). Этот идентификатор остается неизменным до тех пор пока объект присутствует в базе, и повторно не используется даже после удаления объекта. Неизменность идентификатора каждого объекта является ключевым фактором для многих возможностей VERSANT, таких как постоянные ссылки, ссылки между базами данных и прозрачной миграции объектов между базами данных.

VERSANT предоставляет интерфейс к языкам программирования C, C++ и Smalltalk. Каждый интерфейс состоит из нескольких библиотек функций и предопределенных типов. Когда с использованием интерфейса к языкам программирования определяется новый класс, VERSANT создает объект типа описатель класса для сохранения информации о новом классе в базе данных. В интерфейсе к C++ VERSANT также ассоциирует с каждым классом объект типа PClass, содержащий дополнительную информацию о классе необходимую для работы C++ интерфейса.

Каждый сохраняемый объект и его класс ассоциированы с базой данных. Объект может перемещаться между разными базами, но в любой момент времени он привязан к конкретной базе данных. Так миграция объекта включает перемещение объекта с определением его типа, конкретный объект с описанием класса может быть продублирован в нескольких базах данных. Если в базе данных уже существует класс с таким же именем, но определенный по другому, то попытка миграции класса будет блокирована.

Интерфейс VERSANT к языку C++ обеспечивает доступ к базе данных посредством трех C++ классов:

- PDOM (Persistent Distributes Object Manager) менеджер сохраняемых распределенных объектов определяет общие методы для общения с базой: начало сессии, открытие базы данных, фиксация или откатка транзакций.
- PClass (Persistent Class) класс сохраняемого объекта определяет для класса такие методы, как выборка (запрос среди всех экземпляров объектов класса), создание индекса, и.т.п.
- PObject (Persistent Object) сохраняемый объект определяет методы общие для всех сохраняемых объектов: копирование и удаление объекта, получение идентификатора объекта...

Каждый класс приложения, требующий наличия свойства сохраняемости для своих объектов, должен быть выведен из класса PObject. Процесс компиляции и компоновки в VERSANT включает вызов специальной утилиты “sch2db”, которая строит описания классов для базы данных по определениям классов в программе. Схема построения программы на VERSANT C++ аналогична ONTOS C++. Перед изменением объекта должен быть вызван метод `dirty()`, определенный в классе PObject и отмечающий объект как модифицированный. Все модифицированные объекты будут записаны в базу данных при фиксации объемлющей транзакции.

При использовании C++ интерфейса, объекты могут быть сделаны сохраняемыми путем переопределения оператора создания объекта new или ссылкой на объект из другого сохраняемого объекта. После загрузки корневого объекта из базы данных, последующие перемещения объектов с диска в память прозрачны для программиста и происходят при обращении по ссылки к объекту. Объекты могут ссылаться друг на друга с использованием идентификаторов объектов. VERSANT предоставляет специальные C++ классы для представления одиночных и множественных ссылок между объектами: Link и LinkVstr (массив переменной длины из ссылок). Ссылки могут быть как типизованными (ссылаться только на объекты определенного класса) или не типизованными. Также могут быть определены двунаправленные ссылки. Преимуществом использования абстрактных ссылок VERSANT является то, что в отличие от прямых указателей, они обеспечивают автоматическую загрузку адресуемого объекта.

В отличие от C++, программа на языке Smalltalk содержит полный набор метаданных с описанием атрибутов и методов классов. Эта информация используется для создания описания VERSANT классов, поэтому нет необходимости в специальной утилите для построения описания схемы базы данных по описанию классов в программе. Когда объект неизвестного VERSANT класса помещается в базу данных, автоматически создается определение этого класса для VERSANT. Для того, чтобы объект Smalltalk стал сохраняемым, ему необходимо послать метод becomePersistent и, при последующей фиксации транзакции, он будет записан в базу. Также становятся сохраняемыми и заносятся в базу данных все объекты, на которые есть ссылки из сохраняемых объектов.

При посылке сообщения сохраняемому объекту, осуществляется автоматическая блокировка объекта на чтение и пересылка из базы данных в Smalltalk приложение. Таким образом интерфейс прозрачен по отношению к операциям чтения. После загрузки объекта в память программы, он может быть изменен обычными средствами языка Smalltalk. Однако для того, чтобы изменения были сохранены в базе, объекту необходимо послать сообщение vDirty. Это метод помечает объект, как измененный и меняет режим блокировки с “блокировки на чтение” на “блокировку на запись”. Изменения будут записаны в базу данных при выполнении следующей операции фиксации транзакции.

Масштабируемая распределенная архитектура VERSANT (VSDA) обладает такими свойствами, как прозрачное распределение данных, блокировки на уровне объектов, динамическое изменение схемы базы данных и встроенная поддержка кооперативных вычислений. В VERSANT термины процесс клиента и процесс сервера являются относительными понятиями и определяются ролью, которою играет процесс при взаимодействии нескольких процессов. Термин сервер относится к машине, на которой идет серверных процесс, предоставляющий доступ многочисленным пользователям к одной или нескольким базам данных. Термин клиент обозначает приложение, осуществляющее запросы к серверам баз данных.

Часть системы VERSANT, выполняющуюся на той же машине, что и приложение клиента называют VERSANT менеджером. VERSANT менеджер реализует внутреннюю объектную модель, отвечает за управление кэш памятью объектов клиента, осуществляет координацию распределенных транзакций (с использованием двухфазной схемы), работу с версиями объектов, обработку запросов, управление долгоживущими транзакциями и управление сессиями.

Часть системы VERSANT, выполняющуюся на машине на которой храниться данные, называют VERSANT сервером. VERSANT сервер служит интерфейсом между VERSANT менеджерами и операционной системой и отвечает за буферизацию страниц, блокировку объектов, доступ к диску, фиксацию транзакций и восстановление после сбоев. Для каждого клиента базы данных создается отдельный серверный процесс. Различные серверные процессы взаимодействуют между собой с использованием разделяемой памяти.

VERSANT осуществляет распределение работы между клиентом и сервером. Кэширование объектов осуществляется на клиенте, что значительно повышает эффективность работы приложения и уменьшает нагрузку на сеть. А обработка запросов ведется на сервере, что также способствует уменьшению сетевого трафика, так как клиенту посылаются только результаты выполнения запроса.

В архитектуре VERSANT клиент и сервер могут располагаться на одном компьютере или быть разделены в сети. Каждый сервер может поддерживать несколько глобальных баз данных, а клиент работать с несколькими серверами. Объекты могут содержать ссылки на объекты с других узлов и могут прозрачно для приложения мигрировать с одного узла на другой. Допустимо также создание частных баз данных, доступ к которым имеет только один клиент. Клиенты и серверы связаны друг с другом посредством абстрактного сетевого протокола VERSANT, который изолирует их от деталей конкретно используемого сетевого протокола.

Механизм транзакций в VERSANT отличается значительной гибкостью. Во первых, транзакции могут быть кратковременными и долговременными. Кратковременные транзакции соответствуют обычным транзакциям, используемым в базах данных. Кратковременная транзакция начинается с доступа к объекту и заканчивается фиксацией или откаткой. Распределенные транзакции затрагивающие объекты из различных баз данных, управляемых различными серверами, фиксируются с использованием двухфазного протокола для обеспечения свойства атомарности транзакции. Долговременная транзакция представляет из себя серию из нескольких кратковременных транзакций. Точки начала и конца долговременной транзакции должны быть явно специфицированы в программе. Во время долговременной транзакции объекты могут быть затребованы из глобальной базы данных в частную, при этом допускается установка различных типов блокировок. Например приложение может захотеть получить уведомление об одном из следующих событий: объект стал доступным, мягкая блокировка (soft lock) была снята, создана новая версия объекта, к объекту было обращение, объект был изменен или удален. VERSANT также позволяет определять пользовательские типы блокировок, путем указания имени для обозначения соответствующего режима и определения относительного приоритета и совместимости с другими типами блокировок.

VERSANT поддерживает использование версий объектов. При загрузке в приложение объекта имеющего версии, на него не накладывается никаких блокировок, поэтому другие клиенты могут также затребовать этот объект для чтения или модификации. При возврате измененного объекта в базу данных создается версия, являющаяся потомком для первоначальной версии объекта. При внесении нескольких параллельных изменениях возникают параллельные версии объекта. Контроль за версиями объектов осуществляется системой автоматически.

В VERSANT используется объектное расширение языка SQL, позволяющее определять запросы, выбирающие в базе данных объекты в соответствии с критерием выборки. Эти объекты могут в дальнейшем быть использованы как для обычной для объектно-ориентированного программирования навигации по ссылкам, так и служить аргументами других SQL запросов.

Операторы VERSANT Object SQL объединяют синтаксис запросов SQL с использованием выражений C++. Например следующий запрос осуществляет печать всех имен всех служащих:

```
SELECT e.get_name()
```

```
FROM Employee e;
```

Для поиска во всех подклассах данного класса предусмотрена специальная форма оператора выборки. Ниже приведен запрос осуществляющий построение массива указателей на всех служащих, зарплата которых превышает $750:

```
SELECT FROM SUBCLASS
Employee *e WHERE
e->>get_salary() > 750; /* операция ->> используется вместо -> */
                        /* при поиска среди подклассов         */
```

VERSANT поддерживает изменение схемы базы данных во время выполнения. Соответствующие изменения базы данных производятся в "отложенном" (lazy) режиме: объекты на диске могут иметь формат, соответствующий старому классу, и будут преобразованы к новому формату в момент обращения к объекту. Изменения схемы базы данных, затрагивающие некоторый класс, распространяются на все его подклассы. VERSANT так же предоставляет доступ к схеме базы данных и позволяет определять новые классы во время выполнения программ.

2.4.5 ITASCA

ITASCA является распределенной активной объектно-ориентированной СУБД, разработанной компанией Itasca Systems. ITASCA имеет черты свойственные традиционным ООСУБД, такие как использование постоянного хранилища для данных и схемы, управление параллельным доступом, блокировки, управление транзакциями, различные уровни защиты, ведение журнала и восстановление базы данных как в случае сбоя процессора, так при сбое дисков. К дополнительным свойствам ITASCA можно отнести поддержку динамического изменения схемы базы данных, распределенный механизм версий объектов, управление распределенными транзакциями, выполнение распределенных запросов, оповещение об изменениях, миграцию объектов и масштабируемую архитектуру.

ITASCA реализует свою собственную объектную модель, обладающую свойствами объектных моделей многих других ООСУБД. Каждому объекту присваивается уникальный идентификатор объекта, в который включается информация о классе объекта и базе данных в которой объект был создан. Атрибуты формируют состояние объекта, в то время, как методы определяют его поведение. Класс объединяет объекты имеющих одинаковый набор атрибутов и методов. Допускается выведение подклассов из данного класса и образование иерархии классов. ITASCA также поддерживает множественное наследование. Классы в ITASCA в свою очередь также являются объектами. Поддержка полиморфизма в ITASCA обеспечивается возможностью определения нескольких операций с одинаковым именем и поздним связыванием методов.

ITASCA предоставляет возможность создания составных объектов, которые строятся из объектов компонент (потомков). ITASCA поддерживает инвертированные ссылки между родительским объектом и объектами потомками. Объекты-потомки могут принадлежать одному родителю или разделяться между несколькими родителями. Они также могут быть объявлены как независимыми, так и зависимыми от родительского объекта. В последнем случае потомок будет удален при удалении родителя.

ITASCA является активной базой данных в том смысле, что система хранит и выполняет методы в базе данных. Так как ITASCA использует собственную объектную модель и не пользуется определением или реализацией объектов в приложении, то эта ООСУБД является языково-независимой. Объекты в ITASCA представлены в независимом формате, который распознается каждым из интерфейсом прикладных программ (API). Объекты созданные с использованием одного языка программирования, могут быть использованы в программе написанной на другом языке. Разработчики могут использовать для создания приложений такие языки как C++, CLOS, C или Common Lisp. Приложения написанные на различных языках могут вызывать методы хранящиеся в базе данных ITASCA и написанные на языке Common Lisp. В дополнение к этим хранящимся на сервере методам, приложение может содержать собственные методы для работы с данными.

ITASCA представляет из себя полностью распределенную ООСУБД, позволяющую иметь произвольное число клиентов и серверов. База данных ITASCA состоит из разделяемых и локальных сегментов. Разделяемая база данных содержит объекты, которые доступны всем приложениям. Локальная база данных содержит объекты, которые являются локальными для конкретного пользователя или группы пользователей. Поскольку серверы в ITASCA являются распределенными, то каждый сервер имеет собственный сегмент разделяемой базы данных. Кроме того, ITASCA сервер может иметь произвольное число локальных баз данных.

Каждый экземпляр объекта в ITASCA записывается на один из узлов. Система или пользователь могут осуществлять миграцию объекта с одного узла на другой для увеличения локальности данных. Доступ к перемещаемым данным остается прозрачным для приложения, которому не нужно знать о физическом расположении объекта в распределенной базе.

ITASCA дублирует описание схему базы данных на всех узлах для увеличения производительности. Описание схемы также включает код методов. Управление изменениями схемы осуществляется автоматически на всех узлах сети, включая узлы недоступные в момент изменения схемы. Автоматическое отслеживание изменений схемы, включая модификацию реализаций методов, существенно упрощает задачу администрирования базы данных.

В ITASCA используется децентрализованная модель управления базой, то есть отсутствует единственный центральный сервер, который может стать узким местом в системе и привести к нарушению работы всех узлов в случае сбоя этого сервера. Это очень важно для распределенной системы с повышенными требованиями к надежности.

ITASCA предоставляет возможность динамического изменения схемы базы данных. Авторизованные пользователи могут добавлять и удалять атрибуты из классов или менять отношения наследования в любой момент времени. Все эти изменения могут быть сделаны интерактивно, без остановки системы и переформатирования базы данных.

Модель задания прав доступа в ITASCA тесно связана с иерархией классов. Она позволяет задавать права доступа для любого уровня иерархии классов с помощью как списков разрешения доступа, так и списков запрета доступа (положительная и отрицательная авторизация). Например, предоставление доступа ко всем объектам кроме одного конкретного требует только двух авторизации: глобального разрешения для всех объектов и последующего запрета для конкретного объекта. Авторизация распространяется также на классы, экземпляры классов, атрибуты и методы. Администрирование базы данных упрощается за счет наследование прав доступа.

Долговременные транзакции позволяют пользователю затребовать объекты из глобальной, распределенной базы данных в собственную локальную базу. После этого объект может быть изменен в локальной базе без затрагивания глобальной базы данных. Эти изменения могут быть зафиксированы в локальной базе. Затем пользователь может вернуть измененные объекты обратно в глобальную базу данных. Система сохраняет состояние долговременной транзакции, поэтому это состояние может быть восстановлено в случае сбоя в процессе выполнения такой транзакции.

ITASCA поддерживает контроль над версиями объектов. При создании новой версии объекта формируется дерево версий. ITASCA позволяет иметь альтернативные версии объекта, когда несколько версий имеют общего родителя в дереве версий. ITASCA использует обобщенные версии для динамической адресации самой последней версии или версии по умолчанию, без всякого вмешательства пользователя.

Оповещения об изменениях в ITASCA основано либо на использовании флагов, либо на посылке сообщений. Оповещение с помощью флагов представляет из себя пассивную схему оповещения, т.е. приложение должно само производить опрос этого флага для выяснения того, был ли объект изменен. Оповещение с помощью посылки сообщений является активной схемой оповещения, при этом соответствующий метод будет выполнен сразу же по получению сообщения об изменении объекта.

Менеджер памяти в ITASCA использует как буфер страниц, так и буфер объектов. В ITASCA реализована стандартная для баз данных схема управления буфером страниц. Размеры страницы и объекта никак не связаны между собой: на странице может располагаться несколько объектов и один объект может занимать несколько страниц. Запрошенный объект перемещается из страничного буфера в буфер объектов. Таким образом буфер объектов содержит часто используемые объекты, что позволяет увеличить производительность сервера. Кэширование объектов осуществляется также и в приложениях клиентов, что существенно повышает скорость работы приложений и уменьшает нагрузку на сеть.

Для работы с мультимедиа данными в ITASCA предусмотрен менеджер больших объектов. Объекты большого размера можно разделить на два класса: линейные, использующие последовательный доступ, такие как текст или аудио данные, и пространственные - имеющие двухмерную структуру, такие как битовые образы. ITASCA предоставляет ряд встроенных классов для представления таких типов данных, как звуковая запись, видео образы и текст.

Архитектура ITASCA легко поддается расширению. Многие операции нижнего уровня определены как методы классов, которые могут быть переопределены. К таким методам относятся методы для создания, изменения и удаления объектов, образования новой версии объекта, загрузки и выгрузки объекта...

Выполняемый образ ITASCA включает в себя компилятор и интерпретатор языка Lisp. Локальный интерфейс к языку Lisp позволяет приложению напрямую загружать Lisp функции в базу данных. Удаленный интерфейс к языку Lisp делает возможным доступ к базе данных с различных узлов сети. Интерфейс к языку C предоставляет возможность обращения к объекту по его идентификатору и может быть использован для построения интерфейсов к другим языкам (например к C++) путем вызова внешних функций C.

2.4.6 Objectivity/DB

Объектно-ориентированная СУБД Objectivity/DB была разработана компанией Objectivity Inc. Objectivity/DB поддерживает полностью распределенную архитектуру, позволяющую прозрачно выполнять операции в среде объединяющей различные базы данных, схемы, пользователей, компьютеры, операционные системы и сетевые протоколы. Интерфейсы к языкам программирования включают интерфейс, основанный на использовании библиотеки классов для C++, функциональный интерфейс для языка C, и язык запросов SQL++. Многочисленные административные утилиты и средства для разработки предоставляют как интерактивный, так и программный интерфейс, а также позволяют использовать продукты других фирм

Объектная модель Objectivity/DB основана на модели языка C++. Библиотека классов Objectivity/DB содержит стандартные классы для представления строк, словарей, отношений, и.т.п. Разработаны также библиотеки классов для отдельных областей применения. Объекты имеют ассоциированные с ними методы и набор атрибутов, которые могут быть скалярными типами, структурами, массивами фиксированной или переменной длины.

В Objectivity/DB также присутствуют концепции, отсутствующие в C++. Объекты могут быть связаны вместе с использованием динамических ассоциаций. Механизм ассоциаций поддерживает однонаправленные и двунаправленные отношения типа один-к-одному, один-ко-многим и многие-ко-многим. Составные объекты могут быть образованы логическим объединением объектов в ассоциации. Составные объекты могут быть включены в качестве компонент в другие составные объекты. Для составных объектов определены операции блокировки и удаления, действующие так, как будто бы имеют дело с одним объектом. Для определения ассоциаций между объектами в Objectivity/DB имеется высокоуровневый язык определения объектов (ODL). По этим определениям автоматически генерируются определения C и C++ методов для работы с составными объектами.

Интерфейс Objectivity/DB к C++ позволяет использовать для создания приложения стандартные компиляторы и отладчики, без необходимости использовать специальный препроцессор, обеспечивая доступ к базе данных посредством переопределенных операторов C++ (таких как оператор `new, ->, ...`) и специальных классов Objectivity/DB. Суперклассом всех сохраняемых объектов является класс `ooObj`. Для представления ссылок между объектами также используется специальный класс, аналогичный абстрактным указателям ONTOS.

Для Objectivity/DB также разрабатывается интерфейс с языком Smalltalk, который будет поддерживать сборку мусора, изменение схемы во время выполнение, миграцию объектов и управление версиями объектов. Сохраняемость в этом интерфейсе будет определена через достижимость из корневых объектов базы данных. Предполагается разделение классов между C++ и Smalltalk приложениями.

Objectivity/DB реализует распределенную архитектуру клиент/сервер. Для обеспечения работы в распределенной среде Objectivity/DB использует сетевую файловую систему NFS, вместе с объектным менеджером клиента и менеджером блокировок. Если сервером является Unix система, то Objectivity/DB использует NFS сервера, однако замещая стандартного NFS клиента на своего собственного. Клиент Objectivity/DB выполняет большую часть работы в этой архитектуре. Он взаимодействует с файловым сервером и менеджером блокировок для предоставления приложению согласованного представления о базе данных. Любая платформа, поддерживающая NFS, может быть использована в качестве сервера базы данных. Менеджер блокировок, который может располагаться на любой машине в сети, управляет параллельным доступом к разделяемым объектам, обеспечивая блокировки на уровне страниц. Менеджер определений типов осуществляет управление доступом к информации о всех класса, определенных в базе данных.

Иерархию хранилищ в Objectivity/DB образуют четыре сущности: объекты, контейнеры, базы данных и федеративные базы данных. В основе иерархии лежит объект. Объекты могут создаваться из C++ или SQL. Объекты имеют 64 разрядные идентификаторы и записываются на страницы файла базы данных. Контейнер представляет из себя набор страниц. Контейнер является основной единицей блокировки и кластеризации. Когда программа осуществляет доступ к объекту, клиент сначала ищет страницу, содержащую объект, в локальной кэш памяти клиента. Если страница отсутствует в кэш памяти, клиент посылает запрос на сервер для загрузки страницы в память. Тесно взаимосвязанные объекты могут быть кластеризованы на одну и ту же страницу для увеличения производительности. База данных состоит из контейнера, используемого для системных объектов, и ряда контейнеров, определенных пользователем. База данных отображается на файл и ее размер определяется только способностью конкретной операционной обрабатывать большие файлы. Федеративной базой данной называется множество баз данных, распределенных по различным серверам в сети. 64-разрядный идентификатор объекта состоит из четырех 16-разрядных целых чисел, соответствующих базе данных, контейнеру, странице и номеру объекта на странице.

Архитектура предоставляет единую логическую модель для работы с различными базами данных в гетерогенной сети. Операции над базой данных выполняются прозрачно по отношению к распределенной среде, включая двухфазную схему фиксации транзакций, распространение методов и управление версиями объектов. Объекты могут перемешаться между базами данных, не затрагивая работающий с ними приложений. Возможно определение различных схем базы данных, не затрагивающих других пользователей, и одновременное их использование совместно с разделяемой схемой базы данных. Это позволяет определить собственную модель для локальной группы приложений, оставляя возможность взаимодействия с другими группами. Базы данных могут быть отключены от федеративной базы данных и использоваться автономно, с последующем подключением к этому либо другому глобальному окружению. Возможен также доступ к внешним данным, при определении пользователем соответствующих методов доступа.

Objectivity/DB хранит объекты в базе в независимом от платформы виде. При загрузке объекта в приложение, осуществляется автоматическое преобразование в соответствующие локальные типы данных. Objectivity/DB автоматически отводит под объект столько места, сколько необходимо для записи любого представления объекта.

Objectivity/DB поддерживает как кратковременные, так и долговременные транзакции. Кратковременные транзакции основаны на использовании обычных (временных) блокировок, накладываемых процессом на время выполнения транзакции. Долговременные транзакции основаны на использование постоянных блокировок. По умолчанию для управления параллельным доступом используется подход, основанный на сериализации транзакций, путем наложения двухфазных блокировок, но также имеются средства расширения этой модели такие как, разделяемые блокировки (Много-Читателей-Один-Писатель), блокировки без ожидания и с таймаутом, позволяющие реализовать более сложные схемы синхронизации.

Поддержка версий может быть индивидуально разрешена или запрещена для каждого объекта, а также ограничиваться линейными версиями или допускать наличие параллельных версий в случае параллельного изменения объекта несколькими процессами. Ссылки на объект с версиями могут указывать на конкретную версию объекта или на версию используемую по умолчанию. Конфигурации объектов обеспечиваются применением механизма ассоциации к версиям объектов. При создании новой версии объекта, структура конфигурации в которой объект участвует может быть автоматически изменена, путем автоматического копирования, перемещения или удаления ассоциаций.

Изменение схемы и формата объектов основано на создании версий объектов с определением классов. Каждый раз при изменении определения класса, создается новая версия объекта, представляющего класс. После этого экземпляры объектов могут относиться к старой или новой версии класса. Для преобразования объекта к новому формату пользователю необходимо определить специальный метод, осуществляющий это преобразование, который затем будет позван системой.

2.4.7 O2

Объектная СУБД O2 является разработкой компании O2 Technology. O2 предоставляет интерфейсы к языкам C и C++, объектный язык запросов, генератор пользовательских интерфейсов и графическую среду разработки, включающую отладчик, графический редактор схемы и содержимого базы данных.

Объектная модель O2 интересна, тем что в ней предпринята попытка объединить объектную и реляционную технологию, путем введения как понятия объекта в обычном для объектно-ориентированных систем смысле, так и структурированных значений. Структуры могут содержать объекты, точно также как объекты могут содержать структуры, при этом допускается любая степень вложенности. Однако структуры отличаются от объектов тем, что:

1. структуры не инкапсулируют значения своих компонент
2. структуры не имеют идентификатора
3. у структуры нет методов, манипуляции со структурами осуществляются операторами

Имеется набор предопределенных операторов для структур типа кортеж, множество, список. В результате кортежи O2 ведут себя точно также, как кортежи в реляционной модели, за исключением того что, кортежи в O2 могут содержать объекты и другие структуры. Ниже приведены примеры определения структур в O2:

```
type Document: tuple(title: string,
                     printdate: date,
                     author: tuple(firstname: string,
                                   middlename: string,
                                   lastname: string),
                     components: set(Section));

type Section: tuple(title: string,
                    number: integer,
                    components: set(Paragraph));
```

В определении структуры “Document”, компонента “author” является вложенной подструктурой, а не ссылкой на объект. А компонента “components” определена как вложенное множество кортежей другого типа - “Section”. Типы всех структур, строятся рекурсивно, начиная с предопределенных атомарных типов и используя конструкторы типов, такие как кортеж, множество, список.

Объекты в O2 обладают идентификатором, состоянием и поведением, определяемым набором его методов. Каждый объект принадлежит некоторому классу (а не типу). Пример определения класса приведен ниже:

```
class City
    type tuple (name: string,
                map:  Bitmap,
                hotels: set(Hotel))

    method how_many_vacancies(star: integer): integer,
           build_new_hotel(h: Hotel)
end;


class Hotel
    type tuple (name: string,
                read stars: integer,
                read free_rooms: iteger)

    method reserve_room: boolean,
           check_out
end;
```

Объекты могут ссылаться на другие объекты посредством идентификаторов. Аналогично объект или структура могут состоять из других подструктур. Однако подструктура, являясь частью объемлющего объекта или структуры, не может быть разделена с другими объектами, в то время как объект, на который есть ссылка, не зависит от объекта, содержащего эту ссылку, и поэтому на него могу ссылаться несколько объектов. В O2 поддерживается инкапсуляция состояния объекта. Атрибуты объекта по умолчанию объявляются недоступными извне, если явно не указан спецификатор доступа только на чтение (read) или общего доступа (public). Методы также могут объявляться как доступные или не доступные вне класса.

Реализация метода отделена от его описания внутри класса. Ниже приведено тело метода `how_many_vacancies`, описанного выше. Метод реализован на языке O2C, однако возможна реализация и на языках C или C++.

```
method body now_many_vacancies(star:integer) in class City
{
        int number = 0;
```

```
        o2  Holel h;

        for (h in self->hotles where h->free_rooms > 0
             && h->stars == star)
        {
            number += h->free_rooms;
        }
        return numbers;
}
```

При выведении из другого класса, класс наследует его тип и методы. Конфликты имен атрибутов или методов при наследовании могут быть разрешены переопределением их в подклассе.

Объекты становятся сохраняемыми при прямом или транзитивном присоединении их к сохраняемому "корню". Эти корни определяются при построении схемы базы данных путем присвоения им имен. Эта модель называется "сохраняемость по достижимости" (persistency by reachability) и обладает следующими свойствами:

1. Когда объект делается сохраняемым, автоматически становятся сохраняемыми, все объекты на которые он ссылается.

2. Объекты на которые больше никто не ссылается автоматически вычищаются сборщиком мусора.

3. Программист не должен решать является ли объект сохраняемым в месте создания объекта. Вместо этого объект может стать сохраняемым, когда это действительно необходимо.

Архитектура O2 состоит из следующих компонент:

1. O2Engine, языково-независимой менеджер базы данных, поддерживающий объектную модель данных.

2. O2Look, графическое средства для создания просмотра и редактирования объектов базы данных.

3. O2SQL, SQL-подобный объектный язык запросов, включающий оптимизатор запросов и возможность вызова методов.

4. O2Tools, среда разработки, включающая графические редакторы и построители форм, отладчик...

5. C и C++ интерфейсы.

6. O2C, язык программирования 4-ого поколения (расширение C++)

O2Engine можно разбить на три уровня. На верхнем уровне находится менеджер схемы, ответственный за создание, извлечение, модификацию и удаление классов, методов и глобальных имен. Он также отвечает за обработку семантики наследования и проверку корректности схемы. На среднем уровне расположен менеджер объектов, который обрабатывает идентификаторы объектов и посылку сообщений, осуществляет операции над структурами. На этом уровне также реализуется модель сохраняемости по достижимости, сборка мусора, стратегии индексирования и кластеризации. На нижнем уровне находится менеджер дисков, в качестве которого используется расширенная версия Wisconsin Storage System (WiSS). WiSS обеспечивает доступ к последовательным файлам состоящим из записей, индексы реализованные в виде Б деревьев или хэш таблиц, средства кластеризации, а также работу с сохраняемыми объектами. Объекты размещаются на страницах, которые являются основной структурной единицей на этом уровне. WiSS реализует стандартную (одноуровневую) модель транзакций. Для синхронизации совместного доступа используются блокировки, а для откатов и восстановления транзакций ведется журнал транзакций с опережающей записью.

В O2 реализована разновидность архитектуры клиент/сервер, называемая страничным сервером, в которой сервер имеет дело только со страницами и не понимает семантики объектов. Основным преимуществом этой архитектуры является максимальное использование ресурсов рабочих станций клиентов, в то время как сервер выполняет только те операции, которые может сделать только он. Так как обмен между сервером и клиентами ведется страницами, то на первый взгляд может показаться, что это ведет к пересылки большого количества ненужной информации по сети. Однако посылка пакета содержащего 4Kb данных занимает не намного больше времени, чем посылка пакета со 100 байтами данных. А при нормальной работе механизма кластеризации, такой подход может привести к существенному увеличению производительности, так как с большой вероятностью значительная часть объектов на странице будет затребована приложением при дальнейшей работе.

Так как сервер не понимает семантики объектов, то O2Engine не может выполнять запросы и методы на сервере. Однако для клиента возможен индексный доступ к большим множествам объектов, так как алгоритмы индексации реализованы поверх кэш памяти страниц клиента. Эксперименты O2 Technology с предшествующими архитектурами объектных серверов показали, что значительная часть накладных расходов вызвана синхронизацией содержимого кэш памяти (необходимо избегать ситуаций, когда на сервере и в кэш памяти клиентов находятся различные версии объектов).

Разделение между клиентом и сервером происходит на уровне WiSS, имеющем дело со страницами. WiSS разбит на клиентский и серверный процесс следующим способом: серверный процесс состоит из (i) уровня, обеспечивающего доступ к дисковым страницам и управление распределением ресурсов диска, (ii) уровня кэш памяти страниц, (iii) средств для управления параллельным доступом и восстановления базы данных. Доступ к серверу осуществляется посредством RPC протокола. Клиентский процесс состоит из уровня управления кэш памятью страниц и всех вышестоящих уровней. Кроме того, локальный менеджер блокировок осуществляет кэширование блокировок, накладываемых клиентом. Запросы на ввод/вывод к нижнему уровню WiSS на машине клиента преобразуются в RPC запросы к серверу.

Объекты и структуры представлены в O2Engine в виде записей. Формат объекта на диске и в памяти идентичен, что позволяет избежать накладных расходов на преобразование объектов. Идентификаторы записей используются как постоянные идентификаторы для объектов и как указатели на конструируемые структуры. На диске записи содержат объекты и структуры, ссылающиеся друг на друга посредством физических идентификаторов, т.е. идентификаторов, определяющих позицию на диске. Эти идентификаторы содержат 2-ух байтовый идентификатор тома, 4-байтовый номер страницы в томе, и 2-ух байтовое смещение на странице. O2Engine использует двухуровневый механизм адресации: идентификаторы записей на диске и управляющие блоки в памяти. Каждый управляющий блок соответствует некоторой записи, содержащей объект или структуру. При обращении к сохраняемому объекту по ссылке, осуществляется поиск данного идентификатора записи в хэш таблице отображения из идентификаторов записей в адреса управляющих блоков. Если такой элемент в хэш таблице уже существует, то используется соответствующий управляющий блок. В противном случае создается новый управляющий блок. Указатели внутри объекта при этом не изменяются для прямого указания на блок, так как иначе пришлось бы обратно заменять их на идентификаторы записей при отсылке станицы обратно на сервер.

Алгоритм восстановления базы данных после сбоя основан на ведении журнала и требует участия как клиента, так и сервера. Запись в журнал осуществляется по принципу опережающей записи (то есть данные гарантированно попадают в журнал до записи на страницы базы данных) и требует при восстановлении только прокрутки вперед завершившихся транзакций. Контроль за параллельным доступом осуществляется посредством иерархического, двухфазного протокола блокировок файлов и страниц. Глобальный менеджер блокировок идет на севере, а на клиента локальный менеджер кэширует установленные блокировки.

O2 поддерживает гибкий C++ интерфейс. Можно либо экспортировать O2 классы в C++ или, наоборот, импортировать C++ классы в O2. При экспорте в C++ происходит автоматическая генерация C++ классов из O2 классов. Также можно экспортировать структуру класса, при этом автоматически сгенерируются C++ методы для чтения и записи объекта в базу данных. Вызов метода чтения объекта происходит неявным образом, при первом доступе к объекту, а метод записи зовется, когда необходимо записать измененный объект в базу (при фиксации транзакции). После того, как объект был прочитан в память, доступ к нему осуществляется, как к обычному C++ объекту. Для каждого C++ класса X генерируется класс `O2_X`, который используется в качестве постоянной ссылки на объект класса X. Таким образом, вместо обычного указателя на класс `X (X* p)` в программе должен пользоваться сгенерированный тип `(O2_X* p)`. При импорте C++ классов в O2, осуществляется генерация соответствующих определений O2, а также методов для чтения/записи и соответствующих классов ссылок.

2.4.8 MATISSE

ООСУБД MATISSE является многопользовательской системой, построенной в соответствии с архитектурой клиент/сервер, которую разработала компания ADB. MATISSE имеет адаптируемую объектную модель, которая может быть расширена пользователем. Интерфейс к прикладным программам MATISSE состоит из набора C функций, которые могут быть использованы в любом языке программирования. MATISSE также предоставляет интерактивный редактор объектов, который осуществляет прямой доступ к базе.

MATISSE реализует объектную модель, которая не зависит ни от какого конкретного языка программирования. Эта модель построена как отдельный уровень над более примитивной микромоделью. Более точно, MATISSE поддерживает следующие уровни:

микромодель: Микромодель является нижним уровнем в базе данных MATISSE. В ней реализована очень простая схема устройства объектов и связей между ними. Микромодель используется для построения шаблонов.

шаблоны или метамодель: Шаблон или метамодель это схема, построенная с использованием микромодели, которая описывает конкретную модель данных или объектную модель, то есть набор правил по которому строится схема базы данных. Единственной метамоделью, в настоящее время существующей в MATISSE, является объектно-ориентированная метамодель. MATISSE подчеркивает, что другие типы моделей могут быть определены на базе микромодели, например реляционная модель.

схема: Схемой называется набор описателей данных/объектов, которые используются в приложении.

Микромодель MATISSE построена в соответствии с концепцией семантических сетей с расширениями для поддержки отношений, структур, индексов и триггеров (процедур зовущихся при изменении отношений). Объектно-ориентированная метамодель MATISSE является семантической сетью, в которой определены понятия класса, отношения, атрибутов, сообщений и методов. Все понятия MATISSE представлены в виде объектов, сохраняемых в базе данных.

Объект в MATISSE имеет неизменный идентификатор и принадлежит одному классу (информацию о котором можно получить во время выполнения). В MATISSE, как и во многих объектных системах, класс является реализацией типа. Интерфейс класса образуется методами определенными в данном классе. Для класса также можно указать атрибуты и отношения. Методы являются независимыми объектами и могут быть записаны на произвольном языке. В дополнении к методам, определенным для конкретного класса объектов, MATISSE также поддерживает универсальные методы, которые могут быть применены ко всей базе данных в целом. Методы могут разделяться между несколькими классами.

Сложные структуры данных могут быть определены либо встраиванием вложенных структур в объект MATISSE, либо с использованием идентификаторов объектов для представления отношений. Связанные объекты не рассматриваются как единое целое в том смысле, что удаление и другие операции автоматически не распространяются на связанные объекты, если только не определены соответствующие триггеры. В MATISSE определены встроенные конструкторы типов, такие как список и массив. Возможно определение и других конструкторов, но пользователь должен обеспечить соответствующее поведение, задав необходимые ссылки и триггеры.

Классы, включая класс "Class", и их составляющие, сами в свою очередь являются объектами, т.е. информация о методе хранится в экземпляре объекта класса "Method", информация об атрибутах - в экземпляре объекта класса "Attribute", и.т.д. Эти объекты физически хранятся на сервере и могут быть изменены точно также, как и обычные объекты. Это позволяет реализовать изменение схемы базы данных во время выполнения программы. MATISSE позволяет использовать любой атрибут произвольного класса в качестве индекса.

Класс "Relationship" содержит определение параметров отношения, а также определенные пользователем триггеры, которые срабатывают до и после добавления или удаления ссылки. Автоматически поддерживаются инвертированные отношения. Целостность ссылок обеспечивается запретом на удаление объекта до тех пор, пока не удалены все ссылки из этого объекта на другие объекты. Всевозможные ограничения и триггеры могут быть определены также для атрибутов и классов, позволяя пользователю создавать свои собственные сколь угодно сложные ограничения целостности.

MATISSE поддерживает стандартное "специализирующее" наследование. Подкласс состоит изо всех атрибутов, отношений, сообщений и методов суперклассов и самого подкласса. В MATISSE допускается множественное наследование, при этом не разрешаются конфликты имен для наследуемых атрибутов и отношений, хотя допускается переопределение методов. Поиск метода при множественном наследовании осуществляется в соответствии с правилом "слева направо сначала в глубину" обхода иерархии базовых классов.

В дополнению к наследованию, различные формы представительства (delegation) могут быть реализованы посредством определенных пользователем триггеров и методов. Универсальные методы не зависят от иерархии наследования и могут быть позваны из любого метода или функции работающей с базой данных.

Структура и поведение объектов MATISSE определяется метамоделью MATISSE. Пользователь может поменять метамодель и, соответственно, поменяется объектная модель системы. Например класс Attribute может быть изменен для предоставления возможности чтения и записи данных из внешней реляционной базы данных. Это позволит любому атрибуту любого объекта быть описанным, как принадлежащему внешней базе данных. Доступ к такому атрибуту будет осуществляться прозрачно для пользователя. Аналогичным образом поддерживаются изменения в схеме базы данных, с тем ограничением, что целостность базы данных может быть проверена менеджером объектов. Все изменения могут вноситься динамически, при работающих приложениях.

ООСУБД MATISSE построена в соответствии с технологией клиент/сервер и состоит из следующих компонент:

1. Сервера базы данных (Server Engine), поддерживающего хронологические версии объектов, многопоточность и прозрачную репликацию данных.

2. Библиотек Server Engine.

3. Библиотеки клиента скомпонованной с приложением клиента.

4. Средств администрирования базы.

Семантика микромодели и метамодели поддерживается библиотекой клиента. Server Engine отвечает только за сохранение данных и контроль параллельного доступа. Взаимодействие с библиотекой клиента осуществляется посредством интерфейса к языку С. В этом интерфейсе определены функции для выполнения всех операций с базой, таких как открытие транзакции, создание и модифицирование экземпляров объектов и навигация по отношениям. Server Engine использует многопоточную архитектуру и поэтому выполняет операции вода/вывода на диск асинхронно и параллельно. Для распределения памяти на диске Server Engine использует различные стратегии (например автоматическую кластеризацию небольших объектов) для повышения производительности. Клиент и сервер взаимодействуют с помощью RPC вызовов. Объекты хранятся на диске сервера, а кэшируются в памяти клиента.

Управление параллельным доступом осуществляется с использованием двухфазных блокировок и двухфазного протокола фиксации транзакций. По умолчанию используется неявная блокировка объектов. Однако пользователь может контролировать последовательность блокировок, и к тому же, благодаря наличию версий объектов, блокировки не требуются для большинства операций чтения. Кэш клиента сбрасывается по окончанию транзакции, за исключением объектов явно задержанных пользователем в кэш памяти (pinned objects). Это гарантирует корректность представления клиента о базе данных, т.е. то, что клиент видит только последние версии объектов (за исключением задержанных объектов).

Все объекты хранятся на диске в разделах (buckets), которые являются атомарными единицами передачи информации между клиентом и сервером. Размер раздела выбирается с целью оптимизации производительности системы. Объекты на диске содержат ссылку на связанные с ним методы. Код методов находится на рабочей станции клиента и выполняется в приложении клиента.

В MATISSE имеется мощная система поддержки версий объектов. При изменении объекта всегда создается новая версия объекта, без изменения других версий или идентификатора объекта. Поддержка версий ортогональна объектной модели и осуществляется компонентой Server Engine. MATISSE поддерживает версии для всех объектов, включая объекты схемы, благодаря чему пользователь может получить целостное представление о состоянии базы в любой момент времени. Состояние базы данных определяется логическим временем, которое увеличивается каждый раз после фиксации транзакции. При обращении к базе данных клиент указывает логическое время, определяющее какое состояние базы данных он хочет получить. Пользователь видит все объекты, существовавшие в момент, соответствующий специфицированному логическому времени, в состоянии на этот момент времени. Это относится как к самим данным, так и к объектам схемы базы данных.

MATISSE поддерживает явное удаление объектов вместо сборки мусора. Удаление ненужных версий объектов происходит под контролем пользователя. Немедленное удаление старых версий фактически означает отказ от использования версий объектов. Обычно сбор ненужных версий осуществляется через заданные пользователем промежутки времени, после вызова пользовательской функции, отмечающей используемые версии. Составление отчетов или создание резервной копии может быть произведено без остановки системы с использованием представления базы данных, соответствующего определенному логическому времени.

Глава 3

Общие принципы построения GOODS

Распределенные СУБД можно квалифицировать по способу расположения данных и обработки запросов на два класса: данные пересылаются к месту обработки и запрос передается к месту расположения данных. Возможен естественно и промежуточный вариант, где решение о том, что выгоднее переслать данные или запрос, принимается динамически на основании различных оптимизационных критериев.

Реляционные базы данных в основном используют второй способ – запрос на языке SQL пересылается на сервер, выполняется там и результаты выполнения запроса пересылаются обратно клиенту. То же относится и к хранимым процедурам, которые также выполняются локально на сервере. Выполнение запроса в месте хранения данных имеет ряд существенных достоинств:

- Высокая производительность, так как возможен прямой доступ к данным.
- Минимизация трафика по сети, так как от клиента передается только запрос и возвращается только результат его выполнения, а все промежуточные результаты локальны для сервера.
- Невысокие требования к ресурсам рабочих станций клиентов. Так как основной объем работы выполняется на сервере, то возможно создание "облегченных" рабочих мест клиентов (т.е. оснащенных

небольшим количеством оперативной памяти и недорогим центральным процессором).

- Использование централизованного сервера облегчает задачи поддержания корректности базы и обеспечения секретности.

Однако эта модель малоэффективна в задачах использующих навигационный подход, то есть осуществляющих обход данных хранимых в базе, а также при работе со сложными структурами данных. Кроме этого при такой модели взаимодействия клиента и сервера основная вычислительная нагрузка ложится на сервер, оставляя клиенту только формирование запросов и отображение полученных результатов, что делает сервер и коммуникации сервера узким местом, ограничивающим производительность системы.

Очевидно, также что модель с "сильным" сервером не подходит для реализации взаимодействия однородных рабочих станция в локальной сети. Более того, современные персональные компьютеры обладают достаточной вычислительной мощностью и имеют достаточный для выполнения сложных задач объем ресурсов (память и диски). Часто оказывается, что общий объем оперативной памяти и суммарная производительность персональных компьютеров, подключенных к мощному серверу, оказывается в несколько раз больше, чем ресурсы сервера.

Объектно-ориентированные СУБД в свою очередь с самого начала были ориентированы в основном на пересылку объектов к клиенту и выполнение метода в контексте клиента. Основными областями применения этих ООСУБД были системы автоматизированного проектирования и информационные системы, характеризующиеся сложной структурой данных и навигационным характером работы с ними. Выполнение кода методов клиентом позволяет лучше использовать ресурсы рабочих станций, но резко осложняет задачи синхронизации и поддержания корректности. К тому же необходимость пересылки каждого объекта приводит к значительному увеличению загрузки сети и поэтому производительность многих ООСУБД ограниченна пропускной способностью сети. Система в которой основная часть работы выполняется на рабочих станциях, а сервер используется для хранения объектов и выполнения только тех операций, которые требуют глобальной координации (управление блокировками, фиксация транзакций, распределение памяти...) называется системой с "активными клиентами".

Альтернативой пересылки объектов к месту выполнения метода является модель удаленного вызова процедур RPC или ее адаптация к объектно-ориентированной модели RMI (удаленный вызов метода). При этом тела методов на клиенте заменяются на специальные заглушки, которые запаковывают свои аргументы и пересылают сообщение на сервер, содержащее код метода и его параметры. При получении такого сообщения сервер в свою очередь зовет специальный метод, который занимается распаковкой аргументов и передает управлении на тело вызванного метода. Такая модель достаточно прозрачна для приложения (нет разницы между локальными и удаленными вызовами) и сохраняет преимущества локальной обработки запроса. Модель RPC широко используется для реализации различных сетевых серверов (примером может служить сетевая файловая система NFS корпорации Sun), для взаимодействия между объектами в современных распределенных операционных системах, а также в приложениях для Internet (спецификация RMI разработана Sun для языка Java).

Однако при таком подходе плохо выразимы транзакционные операции над множеством объектов. Выполнение удаленного метода может потребовать вызова метода для другого объекта на другом сервера и, таким образом, вычисления как бы мигрируют от одного сервера к другому. Если теперь необходимо обеспечить атомарность операции: т.е. либо изменения будут выполнены для всех участвующих в операции объектов, либо результаты выполнения всех методов будут аннулированы, то для этого потребуются сложные и ресурсоемкие алгоритмы.

При проектировании системы GOODS было решено остановится на модели активных клиентов: то есть сервер отвечает только за хранение объектов и за синхронизацию доступа к базе. Выполнение методов осуществляется на клиентах и результаты переносятся в хранилища базы данных по завершению транзакции. Такой подход по мнению автора наиболее подходит для разработки сложных программных систем, широко использующих интерактивный и навигационный доступ к информации, имеющей сложную и неоднородную структуру. Примером таких систем могут быть системы автоматизированного проектирования, информационные и поисковые системы, географические системы, системы имитационного моделирования.

Архитектура GOODS рассчитана на систему состоящую из одного или нескольких серверов, каждый из которых осуществляет хранение и управляет доступом к объектам в одном из фрагментов базы данных.

Клиент представляет базу данных как набор хранилищ, каждое из которых ассоциировано с одним из сервером. Все сервера для одной базы данных соединены друг с другом, что позволяет им выполнять различные глобальные (затрагивающие всю базу данных) операции – например сборку мусора. Вероятность выхода из строя одного из серверов или коммуникаций между серверами предполагается небольшой, так как в этом случае не возможно выполнение алгоритмов, требующих взаимодействия всех серверов. Однако выход из строя или нарушение связи произвольного количества серверов не должно препятствовать остальным серверам обрабатывать запросы клиентов, не использующих объектов с недоступных серверов. При перезагрузке сервера после сбоя или после восстановления соединения, сервер должен нормально включиться в работу и синхронизировать свое состояние с остальными серверами.

Основной код, реализующий алгоритмы обработки данных, находится в приложениях, выполняемых на компьютерах клиентов. Эти приложения написаны на одном из универсальных языков программирования и используют специальные надстройки над языком, обеспечивающие прозрачный доступ к базе данных. Под термином "прозрачный доступ" в данном случае понимается то, что клиент может работать с объектами базы данных точно так же, как и с обычными (локальными) объектами. Естественно, что полной прозрачности достичь невозможно: необходимы методы специфичные для работы с базой (синхронизация, управление транзакциями...). Также часто полезно иметь возможность явного распределения объектов между серверами.

Так как для выполнение метода объекта необходимо сначала загрузить объект с сервера, то эффективная работа объектной СУБД невозможна без кэширования объектов на клиенте. При этом встает проблема синхронизации содержимого кэш памяти различных клиентов. Задача синхронизации доступа к объектам является одной из наиболее сложных и критичных в объектных СУБД. Это объясняется во первых сложной структурой данных - множество объектов произвольным образом ссылающихся друг на друга (по сравнению например с однородной структурой таблиц реляционных СУБД). Во вторых алгоритмы синхронизации в распределенных средах требуют обмена большим числом сообщений, что еще больше увеличивает нагрузку на сеть. И наконец требования различных приложений к протоколам и средствам синхронизации весьма различны. В стандарте ANSI SQL-92 [MS,ANSI] определяются четыре уровня изолированности:

1. Незафиксированное чтение (read uncommited)
2. Зафиксированное чтение (read commited)
3. Повторимое чтение (repeatable read)
4. Сериализуемость

Если в одних приложениях требуется, чтобы представление базы данных для клиента оставалось неизменным в течении всей транзакции (повторимое чтение), то для приложения управляющего потоком документооборота в офисе такое требование может оказаться излишним и затрудняющим разработку системы.

Классическим подходом синхронизации доступа к объектам является наложение на объекты различных блокировок. Концепции транзакций и механизма блокировок хорошо документированы в литературе [29, 30, 31, 32]. Для определения понятия сериализуемых транзакции введем несколько определений. Транзакцию можно представить как последовательность операций чтения и записи в объекты базы данных. При параллельном выполнении нескольких транзакций их операции могут быть перемешаны. Линейная последовательность операций над элементами данных возникающая при выполнении множества транзакций называется *историей*. *Последовательной историей* называется *история* выполнения не пересекающихся транзакций. Две операции в *истории* конфликтуют, если они принадлежат разным транзакциям, затрагивают один и тот же объект и хотя бы одна из них изменяет объект. По истории можно построить граф зависимости, вершинами в котором являются операции транзакций. Если операция o1 принадлежащая транзакции T1 конфликтует с операцией o2 из транзакции T2, то в граф зависимости добавляется ребро `<o1,o2>`. Две истории называются *эквивалентными* если они имеют одинаковые множества транзакций и их графы зависимости эквивалентны. История называется *сериализуемой*, если она эквивалентна некоторой *последовательной истории*.

Есть известная теорема, которая показывает, что для достижения сериализуемости достаточно, чтобы перед выполнением любой операции над объектом, он был сначала блокирован и блокировка не снималась бы до конца транзакции. Недостатки такого подхода заключаются в том, что он существенно снижает параллелизм выполнения транзакций и может привести к возникновению тупиковых ситуаций. Тупиковая ситуация возникает в случае если два или более процессов взаимно блокировали ресурсы друг друга. Пусть, например, процесс А блокировал объект O1, а процесс Б блокировал объект O2. Пусть затем процесс А пытается заблокировать объект O2 а процесс Б объект O1. Ни один из этих процессов не сможет продолжить выполнение, так как они будут ждать друг друга. Борьба с тупиками может вестись тремя способами:

1. Предотвращение тупиков
2. Избежание тупиков
3. Распознавание тупиков

Предотвратить возникновение тупиков можно потребовав, чтобы транзакция сразу блокировала все необходимые ей ресурсы. К сожаления данное решение трудно применить на практики, так как множество объектов, затрагиваемых транзакцией, обычно определяется динамически в процессе выполнения транзакции и не известно в момент ее старта.

Разработано несколько алгоритмов для избежания тупиков, например "алгоритм банкира". Но для работы этого алгоритма необходима информация о том какие блокировки процесс собирается установить в будущем, что также не подходит для большинства реальных приложений.

Распознавать возникновение тупика можно либо используя таймауты, либо анализируя граф зависимостей. Вершинами в этом направленном графе являются процессы работающие с базой. Если процесс П1 блокировал объект О, а процесс П2 пытается установить на этот объект блокировку не совместимую с блокировкой процесса П1, то в графе проводится ребро `<П2,П1>`. В случае использования разделяемых блокировок из одной вершины могут выходить несколько ребер. Тупиковая ситуация соответствует циклу в графе ожиданий.

При обнаружении тупика система должна предпринять меры для выхода из тупиковой ситуации. Для этого, например, можно откатить транзакцию одного или нескольких процессов блокировавших друг друга. Выбор кандидата для прерывания также представляет из себя интересную задачу.

В распределенных средах процесс обнаружения тупиковых ситуаций осложняется тем, что серверы отвечающие за управление механизмом блокировок расположены на разных машинах. Проблема заключается в том, что в недетерминированной распределенной системе невозможно получить снимок глобального состояния системы, соответствующий какому то определенному моменту времени. Поэтому для анализа графа ожиданий нужно либо выделить отдельного сервера для управления блокировками и глобального выявления тупиков, что может привести к возникновению “узкого места” в системе, и как следствие к деградации ее производительности. Либо можно использовать специальные алгоритмы позволяющие собрать корректное представление о состоянии системы. К сожалению большинство из известных алгоритмов, используемых для распознавания тупиковых ситуаций в распределенных системах, страдает тем, что обнаруживает “мнимые” тупики, то есть ситуация отображенная в графе ожидания соответствует различным моментам времен и цикла, который присутствует в графе, на самом деле не существует.

Более того в распределенной системе неоправданно долгое ожидание процессом установки блокировки может быть вызвано не только возникновением тупиковой ситуации, но и просто аварийным завершением или зависанием клиента, блокировавшего объект, или нарушением связи с этим процессом. Поэтому более адекватным и эффективным средством борьбы с неоправданно большими ожиданиями в распределенных системах является использование таймаутов для выявления подвисших блокировок.

У ООСУБД, использующей блокировки, есть два выбора:

1. возложить всю работу по установке и снятию блокировок на программиста, пишущего приложение.

2. отслеживать доступ к объектам и самостоятельно устанавливать и снимать блокировки

Первый подход существенно усложняет разработку приложений, чреват большим количеством ошибок и мешает повторному использованию кода. Последнее объясним на примере. Допустим у нас есть метод foo класса A блокирующий объект при входе в метод и разблокирующий при выходе:

```
class A {
        foo() {
                lock();
                ...
```

```
                unlock();
        }
};
```

Теперь допустим мы реализуем выведенный класс B, переопределенный метод `foo` которого выполняет какие-то действия, а потом зовет метод базового класса:

```
class B : public A {
        foo() {
                ...
                A::foo();
                ...
        }
};
```

В этом случае возможно неправильное поведение программы так как метод `foo` класса B работает с незаблокированным объектом. Конечно введение вложенных блокировок решает данную задачу.

```
class B : public A {
        foo() {
                lock();
                ...
                A::foo();
                ...
                unlock();
        }
};
```

Однако такой подход не снимает проблему в целом. При этом не возможно использование стандартных библиотек классов, так как в них не предусмотрены средства синхронизации.

Второй подход – автоматическое блокирование ресурсов, требует глубокой интеграции СУБД в среду разработки программ (в компилятор или интерпретатор конкретного языка). Кроме того этот подход страдает отсутствием гибкости. Как уже отмечалось, для различных приложение (и даже для различных классов объектов) может потребоваться своя стратегия синхронизации объектов.

Поэтому при проектировании системы GOODS было решено предоставить программисту возможность определения собственной политики синхронизации объектов, но при этом отделить код управляющий синхронизацией от собственно кода приложения. Таким образом появляется возможность задавать свою политику синхронизации для различных классов приложения не модифицируя код методов класса. Вынесение синхронизационных алгоритмов в отдельные классы позволяет использовать наследование при определении различных стратегий синхронизации доступа. Такой подход так же уменьшает вероятность ошибок синхронизации, так как весь код, отвечающий за синхронизацию, собран в одном месте и его проще проанализировать и отладить.

Для задания политики синхронизации объектов в системе GOODS используются так называемые *метаобъекты*. *Метаобъектом* называется такой объект, который управляет поведением объекта. Технология использования метаобъектов (metaobject protocol) нашла широкое применение в современных объектно-ориентированных языках, таких как Common List и Smalltalk. Обсуждение различных аспектов использования метаобъектов дается в работе [33].

Понятие сохраняемости объекта в GOODS является ортогональным понятию класса объекта, то есть объект любого класса может быть как сохраняемым, так и обычным, временным. Сохраняемым объект становится при создании ссылки на него из другого сохраняемого объекта. Этот подход называется сохраняемостью по достижимости. Для каждого хранилища имеется корневой объект, который создается автоматически при инициализации хранилища. Пользователь может заменить тип этого объекта с помощью оператора `become`. Корневой объект всегда является сохраняемым и в системе имеется специальный метод для получения ссылки на корневой объект для каждого хранилища базы данных. Далее множество сохраняемых объектов получается замыканием множества объектов доступных из корневых объектов (то есть в него включаются все объекты, на которые есть ссылки из корневых объектов, все объекты, на которые есть ссылки из этих объектов. . .) Удаление из базы данных объектов на которые больше не осталось ссылок осуществляется системой автоматически с помощью процесса сборки мусора.

Глава 4

Метаобъекты в GOODS

4.1 Аспектно-ориентированное программирование

С момента появления первого компьютера основным способом построения программных систем остается разбиение системы на составные компоненты, такие как процедуры, модули, объекты, клиенты и серверы... Необходимо отметить, что все эти понятия напрямую соответствуют блокам исполняемого кода программы. Однако все возрастающая сложность программного обеспечения приводит к тому, что современная сложная система должна предоставлять решения для сразу нескольких классов задач, таких как управление распределенными данными, синхронизация совместного доступа, защита информации, обеспечение надежности... При попытки воспользоваться при проектировании такой системы стандартным подходом разбиения на модули разработчик сталкивается с рядом проблем. По какому принципу производить разбиение? Какой модуль должен управлять распределением данных? Какой модуль отвечает за восстановление системы после сбоя? Очевидно, что эти различные аспекты построения системы находят отражение во всех компонентах системы. Эти аспекты могут рассматриваться и проектироваться в достаточной степени изолированно друг от друга, но в конечном счете они должны быть объединены вместе на уровне исполняемого кода.

Но так как образующие систему модули строятся в соответствии с принципом разбиения системы, основанном на выделении блоков исполняемого кода, то модуль в свою очередь становится спутанным клубком кода, отвечающего за самые различные аспекты функционирования системы. Это смешение аспектов во многом и определяет сложность и ненадежность большинства существующих программных систем. Более того, в этой ситуации не поможет увеличение степени модульности системы без изменения самого принципа деления на модули. Для решения этой проблемы и было предложена новая парадигма аспектно-ориентированного программирования [34]. В этой парадигме, как это следует из ее название, основной упор делается на выделение абстракций, моделирующих различные аспекты поведения системы, с их последующим автоматическим объединением для получения исполняемого кода.

Реализация различных аспектов функционирования системы может производится с использованием разных подходов, языков программирования... Для объединения всех этих аспектов в единое целое удобно воспользоваться метаобъектным протоколом. Метаобъект позволяет управлять поведением объекта и, таким образом, определять различные аспекты взаимодействия с объектом: синхронизация доступа, сохраняемость, авторизация доступа... Правильно спроектированный интерфейс метаобъектов делает возможным использование наследования и полиморфизма для комбинирования метаклассов, реализующих различные аспекты функционирования системы.

4.2 Интерфейс метаобъектов

Применительно к системе GOODS метаобъекты отвечают за синхронизацию доступа к объектам, как в рамках одного процесса, так и между различными клиентами. Другой задачей метаобъектов является управление кэш памятью объектов загружаемых из базы данных. Рассмотрим эти задачи более подробно.

4.2.1 Управление совместным доступом к объектам различных клиентов

Как уже отмечалось в предыдущей главе задача синхронизации доступа к данным в распределенной системе может решаться различными способами: с использованием блокировок, временных меток... Внесение изменений в базу данных осуществляется посредством транзакций, которые являются основным средством как для управления совместным доступом, так и для обеспечения корректности базы данных и восстановления ее после сбоя.

Для определения транзакции необходимо указать точку начала и конца транзакции (в некоторых системах транзакция автоматически рестартует после завершения, поэтому нет необходимости открывать транзакцию). Точки фиксации или откатки транзакции должны быть указаны программистом явно внутри исполняемого кода. Однако при таком подходе возникает вероятность не учесть всех возможных траекторий выполнения программы и забыть зафиксировать изменения. Например, рассмотрим следующую функцию:

```
void sorted_array::insert(key_t key, object* obj)
{
    db.begin_transaction();

    int l = 0, r = size;
    while (l < r) {
        int i = (l + r) >> 2;
        if (key > e[i].key) l = i+1; else r = i;
    }
    assert(r == l && e[r].key >= key);
    if (e[i].key == key) {
        e[i].counter += 1;
        return;
    }
    assert(size + 1 <= max_size);
    memmove(&buf[r+1], &buf[r], size - r);
    buf[r].key = key;
    buf[r].counter = 1;

    db.commit_transaction();
}
```

Ошибка, имеющаяся в этом методе, заключается в том, что транзакция не закрывается при том условии, что элемент найден в массиве. Эту ошибку очень трудно будет найти, так как она скажется много позже, после возврата из этого метода, и может привести к очень интересным эффектам. Еще одной возможной ошибкой является изменение объектов вне транзакции.

Так как задача управлением доступом к объектам различных клиентов представляет из себя отдельный аспект функционирования системы, то было решено использовать метаобъекты для управления транзакциями и блокировками объектов. Метаобъект может отследить начало доступа к объекту, проконтролировать корректность версии объекта в кэш памяти клиента, определить момент, когда можно произвести фиксацию транзакции и переслать изменения на сервер. Таким образом метаобъект осуществляет полный контроль над выполнением транзакции в процессе клиента и обеспечивает необходимый приложению уровень изолированности.

Алгоритм управления совместным доступом клиентов к объектам зависит от используемого подхода (оптимистический или пессимистический) и выбранной степени изолированности. Различные метаобъекты по разному организуют совместный доступ к объекту. Так при использовании пессимистического подхода объект перед обращением к нему блокируется (в разделяемом или эксклюзивном режиме) для предотвращения конфликтов доступа. При оптимистическом подходе возможно возникновение конфликтов, которые выявляются в момент фиксации транзакции. В этом случае необходимо предусмотреть средства для разрешения конфликта (конечно в общем случае это не представляется возможным).

Различные уровни изолированности соответствуют разным моделям организации взаимодействия между приложениями: от конкурентной до кооперативной. В первом случае считается, что каждое приложение выполняет независимые действия и требуется максимально изолировать приложения друг от друга, то есть создать иллюзию, что приложение работает с базой данных в монопольном режиме. При кооперативном подходе к организации взаимодействия между приложения предполагается, что приложения участвуют в решении общей задачи, при этом может потребоваться, чтобы приложение знало о существовании других приложений и умело с ними взаимодействовать.

В системе GOODS предоставляется несколько базовых метаобъектов реализующих наиболее общие стратегии синхронизации доступа. Используя эти метаобъекты, программист может задавать специализированные алгоритмы, учитывающие специфику приложения и семантику отдельных классов.

4.2.2 Синхронизация доступа внутри процесса

В настоящее время повышенное внимание при разработки программного обеспечения уделяется параллельным вычислениям, то есть созданию алгоритмов, распадающихся на независимые потоки управления, которые могут выполняться одновременно. Повышенный интерес к параллельным архитектурам во многом определяется тем, что такие системы обладают гораздо большей производительностью по сравнению с традиционными однопроцессорными конфигурациями и обладают высокой степенью масштабируемости. При недостатке вычислительных ресурсов системы, можно расширить систему добавив новый процессор. Особенно хорошие результаты при применении многопроцессорных систем получаются для задач обладающих высокой степенью внутреннего параллелизма. К таким задачам, например, относится выполнения реляционных запросов сервером базы данных, поиск информации, обработка изображений... Расширение круга задач, поддающихся распараллеливанию, является важным шагом на пути созданию более эффективного программного обеспечения.

Однако параллелизм важен не только для многопроцессорных архитектур. Современные графические интерфейсы пользователя часто требуют одновременного выполнения нескольких действий: проведения вычислений, отображения результатов, обработки запросов пользователя и запросов от других приложений и аппаратуры. Параллельное выполнение на системах с одним процессором моделируется путем переключения управления между процессами. Большинство современных операционных систем поддерживает несколько моделей параллелизма: как классическую модель, основанную на использовании изолированные друг от друга процессов, общающихся только посредством специальных средств межпроцессного взаимодействия, так и многопоточную модель, позволяющую в рамках одного процесса организовать несколько потоков управления, которые выполняются независимо друг от друга, но разделяют общее адресное пространство. Эти потоки управления внутри процесса в зависимости от способа их реализации называются в различных системах по разному: легковесные процессы, нити управления... В системе GOODS для их обозначения был выбран термин “задача”.

Преимуществом использования задач в рамках одного процесса по сравнению с отдельными процессами, является во-первых то, что задачи разделяют общее адресное пространство и таким образом легче организовать их взаимодействие, и во-вторых порождение задачи и переключение между задачами осуществляется значительно эффективнее, чем при выполнении этих операций с процессами (это объясняется тем, что с процессом связано гораздо больше контекстной информации, чем с задачей, поэтому переключение контекстов занимает больше времени). Система GOODS предоставляет системно-независимый интерфейс для работы с задачами, реализация которого осуществляется наиболее эффективным для данной системы способом.

Для построения параллельных программ необходимо предоставить программисту средства для синхронизации различных задач. Обычно в качестве таких синхронизационных примитивов используют семафоры, мутексы (mutex), события, мониторы... Так как переключение контекста задач может произойти в любой момент, и более того, на многопроцессорной системе возможно одновременное выполнение нескольких задач, то все структуры данных, доступные нескольким задачам, должны быть защищены с использованием одного из синхронизационных примитивов.

Очевидно что синхронизация доступа к данным различных задач предоставляет из себя один из аспектов реализации программной системы. Если вызов синхронизационных примитивов осуществляется непосредственно в методах разделяемого объекта, то получившийся код становится громоздким и непонятным. Например рассмотрим класс стек, реализованный с использованием средств синхронизации потоков управления, соответствующих стандарту Posix:

```
template<class T>
class stack {
  private:
    const int max_size = 100;
    int   top;
    T     buf[max];

    pthread_mutex_t mutex;
    pthread_cond_t  push_wait;
    pthread_cond_t  pop_wait;

  public:
    void push(T elem) {
        pthread_mutex_lock(&mutex);
```

```
        while (top == max) {
            pthread_cond_wait(&push_wait, &mutex);
        }
        buf[top++] = elem;
        pthread_cond_signal(&pop_wait);
        pthread_mutex_unlock(&mutex);
    }
    T pop() {
        pthread_mutex_lock(&mutex);
        while (top == 0) {
            pthread_cond_wait(&pop_wait, &mutex);
        }
        elem = buf[--top];
        pthread_cond_signal(&push_wait);
        pthread_mutex_unlock(&mutex);
        return elem;
    }
};
```

Как видно на этом примере, большую часть методов класса занимает код, управляющий совместным доступом к объекту. За этими конструкциями уже плохо прослеживается сущность данного класса. При использовании более сложных структур данных чем стек, эта проблема принимает все больший и больший масштаб. Еще одним неприятным моментом является то, что программист вынужден повторно записывать код, отвечающий за синхронизацию, для каждого класса.

В соответствии с принципами аспектно-ориентированного программирования управление параллельным доступом к объекту различных задач должно быть отделено от собственных методов объекта и выполняться на уровне метаобъекта. Все базовые реализации метаобъектов в GOODS предоставляют только один способ синхронизации доступа между задачами: взаимное исключение. При вызове метода объекта в рамках некоторой задачи, выполнение метода осуществляется в мониторе, ассоциированным с данным объектом. То есть происходит "захват" объекта данной задачей. Ни одна другая задача не сможет инициировать выполнение этого или другого метода объекта и будет блокирована до тех пор, пока первая задача не освободит объект. Освобождение объекта произойдет при выходе из метода. Задача, которая захватила объект, может рекурсивно вызвать этот метод или другой метод этого класса, при этом объект станет свободным, только при выходе из последнего активного метода.

Описанный подход отличается достаточной степенью универсальности и подходит для широкого круга приложений. Однако ему свойственен и ряд недостатков. Так этот подход достаточно сильно ограничивает степень параллелизма приложения и может привести к возникновению тупиковой ситуации (представим, что задача A вызвала метод объекта O1, а из него произошло обращение к объекту O2, в то время как другая задача начала выполнять метод объекта O2 и из него обратилась к объекту O1). Однако достоинством использования метаобъектного протокола является то, что всегда имеется возможность переопределить стандартный способ и заменить его на наиболее адекватный для данного приложения подход.

Однако для организации взаимодействия задач не достаточно только блокировать объект, для исключения несогласованного доступа. Необходимо так же иметь примитив, позволяющий задаче подождать выполнения некоторого условия, а также примитив для оповещения задачи о наступлении соответствующего события. Для этого в протокол объекта GOODS были включены методы `wait()` и `notify()`. Если задача хочет подождать наступления некоторого события, она завет метод `wait()`. При этом ассоциированный с объектом монитор освобождается, позволяя другим задачам получить доступ к этому объекту. Для продолжения работы данной задачи необходимо, чтобы какая либо другая задача оповестила ждущую задачу о наступлении интересовавшего ее события. Это можно сделать с помощью метода `notify()`, который “пробуждает” спящую задачу. После этого задача должна заново блокировать монитор (для этого остальные задачи должны освободить его). После чего выполнение задачи может быть продолжено. Подход к синхронизации доступа задач к объектам в системе GOODS имеет много общего с моделью синхронизации доступа в языке Java (при этим все методы неявно объявляются как synchronized).

У этого подхода к синхронизации объектов имеется существенный недостаток: задача может повторно входить в монитор, связанный с объектом, несколько раз (при вложенных вызовах методов одного объекта) или последовательно блокировать мониторы сразу нескольких объектов. При этом вызов метода `wait()` снимает только один уровень блокировки с монитора объекта. То есть, при вложенных вызовах методов одного объекта, монитор этого объекта останется заблокированным и после того, как текущая задача будет приостановлена в результате выполнения метода `wait()`. Это приведет к возникновению тупиковой ситуации, так как никакая другая задача не сможет получить доступ к данному объекту и "разбудить" спящую задачу. Кажущееся очевидным решение об освобождении при выполнении метода `wait()` всех ресурсов, блокированных данной задачей, на самом деле является неприемлимым, так как означает, что вызов любого метода может привести к тому, что другая задача получит доступ к объекту и изменит его состояние.

Так как монитор занимает достаточно заметное количество памяти, то создание для каждого объекта отдельного монитора может привести к существенным дополнительным расходам памяти. Для более эффективного использования памяти в системе GOODS используется прием, называемый "турникетом". В объекте хранится только ссылка на монитор. Монитор назначается объекту, только в момент обращения к объекту. После окончания обращения к объекту монитор остается прикрепленным к объекту, но может быть отобран для использования с другим объектом. Таким образом, суммарное количество используемых мониторов равняется максимальному числу объектов, к которым одновременно осуществлялось обращение (то есть были вызваны методы). Очевидно, что это число ограничено количеством задач в процессе, умноженном на максимальную глубину стека вызовов, и значительно меньше общего количества объектов.

Для поиска свободного монитора для объекта используется метод перемещаемого указателя. Ссылки на все созданные мониторы заносятся в массив, который расширяется по мере добавления в него новых элементов. При необходимости выделить монитор для объекта происходит сканирование массива, начиная с текущей позиции до тех пор, пока не будет найден не занятый монитор (то есть не используемый в данный момент для блокировки объекта). Если свободный монитор не найден, то создается новый монитор. Если же такой свободный монитор найден, то ассоциированный с ним объект помечается как не имеющий монитора. После чего монитор переназначается новому объекту и указатель текущей позиции в массиве перемещается на элемент, указывающий на данный монитор. При достижении конца буфера указатель текущей позиции перемещается на начало массива. Таким образом присвоенный объекту монитор может быть перераспределен другому объекту только после того, как указатель совершит полный оборот по массиву. Поэтому при последовательных обращениях к объекту с большой вероятностью не нужно будет каждый раз заново выделять для него монитор, так как привязка объекта к монитору сохранится.

4.2.3 Управление кэш памятью объектов

Как уже отмечалось, кэширование объектов является критическим аспектом для эффективной работы объектно-ориентированных СУБД. Поэтому выбор алгоритма замещения объектов в кэш памяти может сильно повлиять на производительность системы в целом. Широко используемый в операционных системах для выбора кандидата на замещение алгоритм последнего использованного (Last Recently Used) часто оказывается неподходящим для СУБД. Так при использовании этого алгоритма замещения, последовательный поиск в списке из большого числа объектов приведет к выбрасыванию из кэш памяти всех остальных объектов. Так как, с большой вероятностью, почти все из просмотренных объектов больше не будет использоваться, то очевидно что такое поведение алгоритма замещения нельзя назвать оптимальным. С другой стороны, невозможно выбрать алгоритм замещения, которой оптимально ведет себя в любых ситуациях. Часто гораздо лучших результатов можно достичь с привлечением семантической информации о приложении. Так стратегия замещения может меняться для объектов различных классов. Поэтому было принято решение вынести алгоритм управления кэш памятью объектов в методы метаобъекта.

4.2.4 Описание протокола

Текущая реализация GOODS использует язык C++ как язык программирования приложений для работы с базой данных. Но в качестве такого языка возможно использование любого (даже не объектного) языка. При этом сохранятся общие подходы к организации взаимодействия с хранилищем объектов, а также реализации базовых классов, осуществляющих связь приложения с сервером. Так как формат хранения объектов в базе не зависит от используемого языка и архитектуры машины, то возможна совместная работа с базой данных приложений, написанных на различных языках и выполняемых на машинах с разной архитектурой.

Выбор языка C++ обусловлен его популярностью и распространенностью, а также тем, что применение ООСУБД во многом связано с классом приложений с высокими требованиями к производительности (САПР, информационные системы). Поэтому такие приложения обычно пишут с использованием языков, позволяющих достичь высокой производительности при работе программы, а таким требованиям в данный момент в наибольшей степени соответствует C++. Также очень перспективным в этом отношении является язык Java, разработанный корпорацией Sun, и представляющий из себя очищенную модификацию C++. Так как модель и возможности языка Java во многом схожи с C++ (за исключением разве что встроенной в язык сборки мусора и многопоточности), то перенос среды GOODS для работы с языком Java не представляет сложностей. Более того, текущая реализация для C++ поддерживает независимую от операционной системы многопоточную модель и сборку мусора для объектов GOODS. В дальнейшем изложении классы и примеры будут написаны с использованием C++.

Метаобъект в GOODS реализует следующие абстрактные методы:

```
class metaobject {
  public:
    //
    // Make transient object persistent as a result of
    // referencing to it from persistent object
    //
    virtual void make_persistent(hnd_t hnd, hnd_t parent_hnd) = 0;

    //
    // This method is called by object destructor and performs cleanup of
    // deleted object.
    //
    virtual void destroy(hnd_t hnd) = 0;

    //
    // Intertasking object locking
    //
    virtual void lock(hnd_t hnd) = 0;
    virtual void unlock(hnd_t hnd) = 0;

    //
    // Intertasking synchronization primitives
    //
    virtual void wait(hnd_t hnd) = 0;
    virtual bool wait(hnd_t hnd, time_t timeout) = 0;
    virtual void notify(hnd_t hnd) = 0;
```

```
//
// Access to object
//
virtual void begin_read(hnd_t hnd) = 0;
virtual void end_read(hnd_t hnd) = 0;

virtual void begin_write(hnd_t hnd) = 0;
virtual void end_write(hnd_t hnd) = 0;

//
// Instance of object is deteriorated
//
virtual void invalidate(hnd_t hnd) = 0;

//
// Signal event when persistent object is modified by another client
//
virtual void signal_on_modification(hnd_t hnd, event& e) = 0;

//
// Insert persistent object to object cache. Replacement policy
// for objects should be implemented by this method
//
virtual void put_in_cache(hnd_t hnd) = 0;
//
// Prevent object from replacement algorithm activity.
//
virtual void get_from_cache(hnd_t hnd) = 0;

//
// Specify boundaries of nested transaction. Parent transaction will
// not be commited until all nested transactions are finished.
//
virtual void begin_transaction() = 0;
virtual void end_transaction() = 0;

//
// Transaction management
//
enum abort_reason { aborted_by_server, aborted_by_user };

virtual void commit_transaction() = 0;
virtual void abort_transaction(const database* dbs,
                               abort_reason reason = aborted_by_user) = 0;
```

```
    //
    // This methods are called after transaction completion
    // (normal or abnormal) for each object involved in transaction
    //
    virtual void commit_object_changes(hnd_t hnd) = 0;
    virtual void abort_object_changes(hnd_t hnd, abort_reason reason) = 0;

    //
    // This method is called for all object from transaction list
    // when no changes were made within transaction
    //
    virtual void release_transaction_object(hnd_t hnd) = 0;

    //
    // Specify flags for transaction object
    //
    virtual int get_transaction_object_flags(hnd_t hnd) = 0;

  protected:
    inline object_header* get_header(hnd_t hnd) {
        return (object_header*)hnd->obj;
    }

    //
    // Insert object in database transaction list
    //
    virtual void insert_in_transaction_list(hnd_t hnd,
                                            hnd_t parent_hnd = 0) = 0;
    //
    // Remove object from database transaction list
    //
    virtual void remove_from_transaction_list(hnd_t hnd) = 0;
};
```

Большинство методов метаобъекта принимает в качестве параметра идентификатор объекта внутри программы `hnd`. Идентификаторы используются вместо прямых указателей на объекты потому, что в результате перезагрузке модифицированного объекта с сервера или при изменении класса объекта оператором `become`, расположение объекта в оперативной памяти может поменяться. Ниже методы метаобъекта описываются более подробно.

`void make_persistent(hnd_t hnd, obj_storage* storage)`

Данный метод вызывается при занесении нового объекта в базу данных в процессе фиксации транзакции. В GOODS используется модель сохраняемости по достижимости, то есть сохраняемым становится любой объект, на который есть ссылки из других сохраняемых объектов. Таким образом, при фиксации транзакции исследуются все ссылки из измененных сохраняемых объектов, и если такая ссылка указывает на не сохраняемый объект, то он и все адресуемые из него объекты также включаются в транзакцию. Метод make_persistent приписывает объект к конкретному хранилищу (либо к явно специфицированному посредством метода cluster_with, либо, при отсутствии явного указания, к хранилищу объекта из которого была ссылка).

```
void destroy(hnd_t hnd)
```

Вызов метода destroy осуществляется деструктором объекта. Этот метод отвечает за освобождение всех структур, связанных с объектом, и удаление его из кэш памяти.

```
void lock(hnd_t hnd);
```

Установить межзадачную блокировку экземпляра объекта. Вызов этого метода накладывает на экземпляр объекта эксклюзивную вложенную блокировку, блокируя доступ к объекту других задач.

```
void unlock(hnd_t hnd);
```

Снимает межзадачную блокировку экземпляра объекта, разрешая доступ к объекту другим задачам.

```
void wait(hnd_t hnd);
```

Ожидание оповещения объекта. Данный метод снимает с объекта эксклюзивную блокировку и приостанавливает текущую задачу до поступления оповещения от какой либо другой задачи. После получения оповещения блокировка объекта восстанавливается и выполнения приостановленной задачи продолжается. Для корректной работы этого метода необходимо, чтобы объект перед вызовом этого метода был блокирован со степенью вложенности 1. В случае, если к объекту было несколько вложенных обращений, метод `wait` снимет только один уровень блокировки и объект по прежнему останется блокированным. Таким образом никакой другой процесс не сможет получить к нему доступ и послать оповещение.

```
bool wait(hnd_t hnd, time_t timeout);
```

Метод действует аналогично предыдущему, однако время ожидания ограничено таймаутом. Если за специфицированный промежуток времени оповещения не поступило, метод восстанавливает блокировку и возвращает false. В случае поступления оповещения до истечения таймаута возвращается true.

```
void notify(hnd_t hnd);
```

Послать оповещение ждущей задаче. Если оповещения дожидаются несколько задач, то разбужена будет только одна из них. Описанный механизм оповещения во многом похож на аналогичный механизм языка Java. Он обладает достаточной простотой и мощностью, но может привести к возникновению тупиковых ситуаций.

```
void begin_read(hnd_t hnd)
```

Метод зовется при доступе к объекту для чтения. Этом может быть либо вызов константного (не изменяющего содержимого объекта) метода, либо получение значения компоненты объекта. В случае вызова метода доступ к компонентам объекта внутри метода осуществляется напрямую (через this указатель). Метод `begin_read` осуществляет как управление доступом к объекту различных клиентов базы данных, так и синхронизацию обращений к объекту различных задач внутри одного клиента.

```
void end_read(hnd_t hnd)
```

Завершение доступа к объекту (выход из метода). Этот метод снимает с объекта установленные блокировки и определяет момент, когда можно произвести фиксацию транзакции на сервере. Если вызов `end_read` соответствует обращению к объекту с нулевым уровнем вложенности, то производится проверка достоверности экземпляра сохраняемого объекта и удаление устаревшего объекта из памяти клиента. Если экземпляр сохраняемого объекта не является устаревшим, то `end_read` с нулевым уровнем вложенности возвращает объект в распоряжение алгоритма замещения объектов в кэш памяти.

`void begin_write(hnd_t hnd)`

Метод зовется при доступе к объекту для модификации. Этом может быть либо вызов метода, либо ссылка на компоненту объекта из левой части оператора присваивания. В случае вызова метода доступ к компонентам объекта внутри метода осуществляется напрямую (через this указатель). Сохраняемый объект помечается как измененный и, при последующей фиксации транзакции, будет послан на сервер.

`void end_write(hnd_t hnd)`

Завершение модификации объекта (выход из метода) Этот метод снимает с объекта установленные блокировки и определяет момент, когда можно произвести фиксацию транзакции на сервере. Если вызов `end_write` соответствует обращению к объекту с нулевым уровнем вложенности, то производится проверка достоверности экземпляра сохраняемого объекта и удаление устаревшего объекта из памяти клиента. Если экземпляр сохраняемого объекта не является устаревшим, то `end_write` с нулевым уровнем вложенности возвращает объект в распоряжение алгоритма замещения объектов в кэш памяти.

`void invalidate(hnd_t hnd)`

С помощью этого метода сервер информирует клиента о том, что объект был изменен на сервере другим процессом и экземпляр объекта в кэш памяти клиента является устаревшим.

```
void signal_on_modification(hnd_t hnd, event& e);
```

Метод позволяет заказать получение события при изменении объекта другим клиентом. Указанный объект фиксируется в кэш памяти процесса и при получении от сервера сообщения об изменении данного объекта, событие `e` переводится в сигнальное состояние.

```
void put_in_cache(hnd_t hnd)
```

Метод зовется при помещении объекта в кэш объектов. Этот метод реализует алгоритм замещения объектов в кэш памяти. Обычно накладывается ограничение на суммарный размер, занимаемый всеми объектами в кэш памяти. Когда, в результате загрузки нового объекта, это ограничение нарушается, алгоритм замены выбирает кандидатов для выталкивания из кэш памяти до тех пор, пока суммарный размер объектов не станет меньше заданного значения.

Обычно в качестве стратегии замещения при управлении кэш памятью используют стратегию вытеснения наиболее давно неиспользовавшегося объекта (LRU). К сожалению, применительно к базам данных, эта стратегия не всегда обеспечивает хороший результат. Так при поиске среди большого множества объектов, происходит фактически полное обновление кэш памяти и выталкивание нужных объектов. Поэтому использование более интеллектуальных алгоритмов замещения может существенно улучшить работу кэш памяти. Так например, можно поднять приоритет тех объектов, к которым было более одного обращения и не позволять объектам, к которым было ровно одно обращение (а это типично для поиска), вытеснить их из кэш памяти. Другой подход может использовать семантику объектов и определять различные приоритеты для разных классов объектов.

В GOODS стандартный метаобъект реализует двухступенчатую модификацию стандартной LRU дисциплины. Для этого поддерживается два списка. Перед обращением к объекту (метод `begin_read` или `begin_write`) объект исключается из списка для предотвращения его выталкивания в момент, когда к его компонентам имеется прямой доступ из метода. После окончания доступа к объекту (метод `end_read` или `end_write`) объект помещается в голову списка. Кандидаты для замещения всегда берутся из конца списка. Таким образом на базе списка можно реализовать стандартную LRU дисциплину.

Для поддержки модифицированной LRU стратегии используются биты “useful” и “accessed” в маске состояния объекта. Бит “accessed” устанавливается при первом доступе к объекту. Бит “useful” устанавливается при извлечении объекта из списка и если при этом объект находится не в начале списка и бит “accessed” установлен. Это условие служит хорошей аппроксимацией для выбора объектов к которым происходило неоднократное обращение. При этом у объекта, к которому было несколько последовательных обращений (и таким образом, бравшемуся всегда из головы списка), не устанавливается бит “useful” и он не попадает во второй “привилегированный” список. Таким образом объекты, просматриваемые при последовательном поиске и требующие обычно двух обращение (первое при проверке условия поиска и второе при получении ссылки на следующий элемент), не вытесняют из кэш памяти многократно используемые объекты, такие как корневые страницы Б-деревьев, хэш таблицы,...

В Sybase System 11 Sql сервере предлагается подход к управлению кэш памятью противоположный традиционным политикам замещения: “извлечь и выбросить”. Согласно этой стратегии прочитанные с диска буфера не будут больше использоваться в ближайшем будущем и поэтому являются хорошими кандидатами на замещение. Прочитанный с диска блок помещается в конец списка, соответствующий наиболее давно неиспользованным объектам. Таким образом Sybase сервер предотвращает опустошение кэш памяти в результате последовательного сканирования таблицы. По мнению автора используемый в GOODS подход позволяет достичь лучших результатов, так как сохраняет общий принцип работы LRU дисциплины.

```
void get_from_cache(hnd_t hnd)
```

Метод зовется при удалении объекта из кэш памяти или в результате фиксации объекта в кэш памяти. Удаление объекта производится в результате вытеснения объекта из кэш памяти алгоритмом замещения. Фиксация объекта производится при вызове метода для данного объекта и снимается после выхода из метода. Фиксация объекта в памяти необходима для предотвращения выталкивания объекта из памяти в результате работы алгоритма замещения в то время, как на объект существуют прямые ссылки (через this указатель) из активных методов.

`void begin_transaction()`

Выполнение каждого метода в GOODS рассматривается как вложенная транзакция. Транзакция фиксируется при завершении всех вложенных транзакции, то есть когда не осталось активных методов, позванных для GOODS объектов. Может потребоваться увеличить область действия транзакции. Для этого предусмотрены методы метаобъекта, позволяющие явно указать начать начало и конец вложенной транзакцию. Так например, при выводе экранной формы для внесения изменений пользователем, программа может открыть вложенную транзакцию, а по нажатию кнопки "Сохранить" завершить эту транзакцию.

`void end_transaction()`

Завершение выполнения вложенной транзакции. При этом не обязательно происходит фиксация транзакции. Момент завершения транзакции определяется конкретным метаобъектом и может быть, например, отложен до завершения выполнения всех активных методов.

`void commit_transaction()`

Метод осуществляет фиксацию текущей транзакции. Реализация базового метаобъекта осуществляет фиксацию транзакции при выходе из последнего активного метода сохраняемого объекта во всех задачах и при закрытии всех открытых явно (посредством метода `begin_transaction`) подтранзакций. При этом происходит фиксация транзакций во всех базах данных с которыми осуществлялась работа. Метаобъекты могут определять свои подмножества объектов вовлеченных в транзакцию и задавать произвольные алгоритмы ее фиксации.

`void abort_transaction(const database* dbs, abort_reason reason)`

Метод осуществляет откатку текущей транзакции. Требование откатить транзакцию может исходить от пользователя или от сервера. Сервер инициирует откатку транзакции в результате обнаружения конфликтов при использовании оптимистического протокола синхронизации, либо при выходе из строя одного из серверов, участвующего в транзакции.

`void commit_object_changes(hnd_t hnd)`

Данный метод зовется при успешной фиксации транзакции для каждого объекта в ней участвовавшего. Метод отвечает за удаление объекта из списка транзакции и разрешает алгоритму замещения объектов в кэш памяти вытолкнуть при необходимости этот объект из кэш памяти.

`void abort_object_changes(hnd_t hnd, abort_reason reason)`

В отличие от предыдущего метода, этот метод выполняется при ненормальном завершении транзакции для каждого объекта в ней участвовавшего. В качестве второго аргумента метода передается причина откатки транзакции: транзакция абортирована пользователем или сервером.

`void release_transaction_object(hnd_t hnd)`

Если в процессе выполнения транзакции не был изменен ни один объект, то посылки транзакции на сервер не происходит. Вместо этого клиентом осуществляется освобождение всех блокированных транзакцией ресурсов. Для этого метод `release_transaction_object` зовется для каждого из участвовавших в транзакции объектов.

`int get_transaction_object_flags(hnd_t hnd)`

При фиксации транзакции на сервере, север может произвести над объектом ряд операций, которые задаются передающейся вместе с объектом маской флагов операций. К возможным операциям относятся:

tof_update: Запись измененного объекта в базу данных.

tof_validate: Проверка достоверности экземпляра объекта, то есть проверка того, что инициировавший данную транзакцию клиент обладает последней версией этого объекта. Обычно эта проверка выполняется при оптимистическом способе синхронизации.

tof_unlock: Автоматически разблокировать объект при успешной фиксации транзакции. Этот флаг используется при пессимистическом подходе к управления совместным доступом, основанном на наложении блокировок на объекты. Использование автоматического разблокирования объектов позволяет избежать явной посылки запросов на разблокирование всех включенных в транзакцию объектов.

Установка флага `tof_update` происходит автоматически для всех измененных объектов. Таким образом данный метода осуществляет выбор только из `tof_validate` и `tof_unlock`. Допускаются любые комбинации из этих флагов.

`object_header* get_header(hnd_t hnd)`

Получение указателя на защищенную часть класса `object`. Так как в программе может использоваться произвольное число метаобъектов, то невозможно заранее объявить всех метаобъектов "друзьями". Поэтому используемые метаобъектами компоненты объекта выделены в отдельный класс, который наследуется с квалификатором private и доступен только "друзьям" класса объект, которым является абстрактный класс метаобъект. Этот класс содержит метод предоставляющий всем выведенным из него классам доступ к защищенной части объекта.

`void insert_in_transaction_list(hnd_t hnd)`

Занести объект в список объектов, участвующих в транзакции. Участвующий в транзакции объект не может быть вытеснен из кэш памяти объектов до окончания транзакции. Обычно в список помещаются только измененные сохраняемые объекты, однако при использовании некоторых протоколов синхронизации, может потребоваться передать информацию на сервер обо всех объектах, к которым во время транзакции были обращения. Стандартная реализация помещает измененные объекты в начало списка, а неизмененные в конец. Таким образом все измененные объекты расположены подряд и серверу не требуется перемещать объекты в пересланном ему клиентом буфере для записи тела транзакции в журнал транзакций.

`void remove_from_transaction_list(hnd_t hnd)`

Исключить объект из списка объектов участвующих в транзакции. Этот метод зовется при завершении (фиксации или откатке) транзакции.

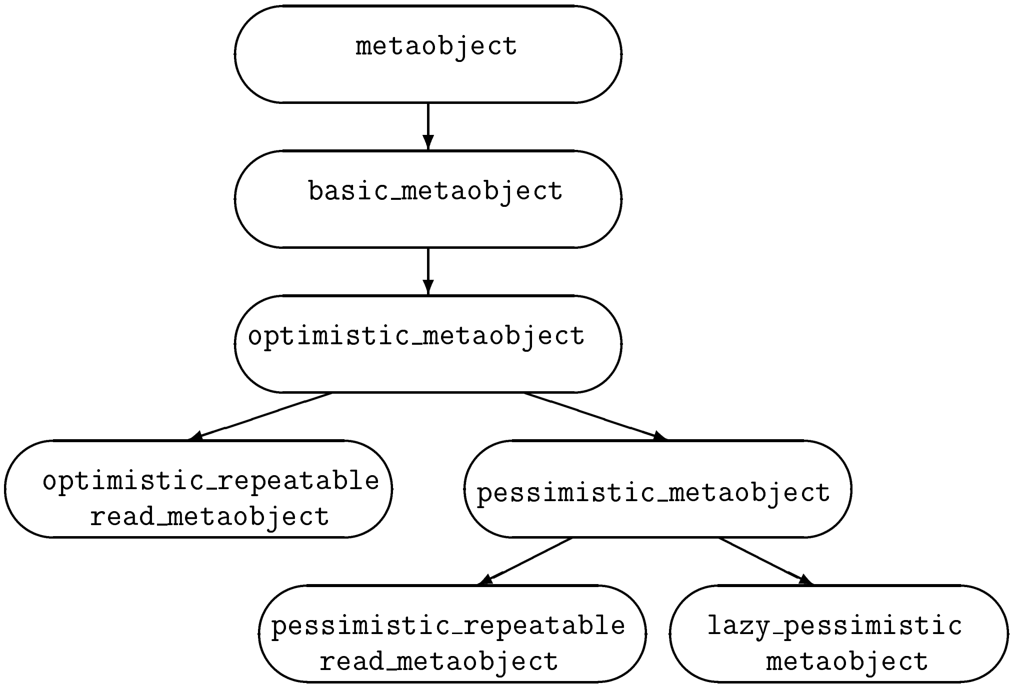

Рис. 4.1: Иерархия стандартных метаобъектов

4.3 Стандартные метаобъекты

Как уже отмечалось, основное преимущество использования метаобъектов заключается в возможности задания собственных стратегий управления доступом к объектам для каждого класса. Однако в большинстве случаев подходит один из стандартных способов синхронизации доступа. В системе GOODS разработан ряд метаобъектов, реализующих наиболее общие протоколы синхронизации. Метаобъекты, реализующие эти протоколы, выведены один из другого и обеспечивают различные уровни изолированности.

4.3.1 Базовый метаобъект

Базовый метаобъект является общей базой для всех стандартных метаобъектов и определяет наиболее общие стратегии для управления кэш памятью и транзакциями. В данном классе реализуется двухступенчатая LRU дисциплина для замещения объектов в кэш памяти. Вначале замещаются те объекты, к которым было не более одного обращения. Для часто и редко используемых объектов отдельно задается ограничение на суммарное колличество памяти, занимаемой объектов. Таким образом предотвращается вытеснение из кэш памяти нужных объектов в результате последовательного поиска среди большого количества объектов и быстрое выбрасывание только что загруженных объектов из кэш памяти в следствии того, что большое количество приоритетных объектов (к которым было более одного обращения) привело к уменьшению памяти доступной для остальных объектов.

В этом же классе реализуется алгоритм управления фиксацией транзакций (в том числе и распределенных транзакций). Протокол фиксации распределенных транзакций будет описан ниже в разделе 4.4. Также предоставляется возможность пользователю определять свой обработчик для ситуации, при которой происходит откатка транзакции сервером. Такой обработчик может быть использован при оптимистическом подходе к синхронизации транзакций для рестарта абортированной транзакции.

В базовый метаобъекте реализованы методы `lock` и `unlock`, используемые для синхронизации дотупа к объекту различных задач внутри процесса. Используется политика взаимного исключения, при которой только одна задача может одновременно работать с объектом.

```
class basic_metaobject : public metaobject {
  protected:
    //
    // Lists of cached objects sorted in LRU order
    // Two separate list are used:
    //  one for objects accessed one or zero times
    //  and one for objects accessed more than once(with 'useful' bit in state)
    // Objects from first queue are replaced first.
    static object_l2_list lru[2];
    //
    // List of objects accessed within current transaction
    //
    static object_l2_list trans_objects;
    //
    // Total size of loaded objects.
    // Sizes are calculated separately for objects accessed 0 or 1 times and
    // for objects acccessed more than once.
    //
    static size_t         used_cache_size[2];
    //
    // Limit for total size of objects in cache
    // Limits are specified separately for objects accessed 0 or 1 times and
```

```
  // for objects acccessed more than once.
  //
  static size_t         cache_size_limit[2];

  typedef void (*abort_hook_t)(metaobject*);
  abort_hook_t abort_hook;

  static event          trans_commit_event;
  static bool           trans_commit_wait_flag;
  static bool           trans_commit_in_progress;

  static monitor_turnstile turnstile;

  struct notify_item {
      notify_item* chain;
      hnd_t       hnd;
      event&       e;
      notify_item(hnd_t id, event& evt, notify_item* list)
      : chain(list), hnd(id), e(evt) {}
  };
  static notify_item* notification_hash[];

  //
  // This virtual methods can be used for metaobject specific
  // transaction object lists. Method 'commit_transaction'
  // tries to commit transactions in all databases.
  // Methos "abort_transaction" abort transaction in concrete database.
  //
  virtual void commit_transaction(object_l2_list& trans_objects);
  virtual void abort_transaction(const database* dbs,
                                 object_l2_list& trans_objects,
                                 abort_reason reason);

  virtual void insert_in_transaction_list(hnd_t hnd,
                                          hnd_t parent_hnd = 0);
  virtual void remove_from_transaction_list(hnd_t hnd);

  virtual void reload_object(hnd_t hnd);
  virtual void update_object(hnd_t hnd);

public:
  virtual void destroy(hnd_t hnd);

  virtual void lock(hnd_t hnd);
  virtual void unlock(hnd_t hnd);
```

```
    virtual void wait(hnd_t hnd);
    virtual bool wait(hnd_t hnd, time_t timeout);
    virtual void notify(hnd_t hnd);

    virtual void make_persistent(hnd_t hnd, hnd_t parent_hnd);

    virtual void invalidate(hnd_t hnd);

    virtual void signal_on_modification(hnd_t hnd, event& e);

    virtual void put_in_cache(hnd_t hnd);
    virtual void get_from_cache(hnd_t hnd);

    virtual void commit_transaction();
    virtual void abort_transaction(const database* dbs, abort_reason reason);

    //
    // Primary cache size (limit0) should be greater then maximal cluster size
    // otherwise sluster can never be loaded
    //
    static void  set_cache_limit(size_t limit0, size_t limit1) {
        cache_size_limit[0] = limit0;
        cache_size_limit[1] = limit1;
    }
    //
    // Set hook which will be called if transaction will be aborted by server.
    // Hook is called with global mutex locked and should not release this lock
    //
    void set_abort_transaction_hook(abort_hook_t hook) { abort_hook = hook; }

    basic_metaobject();
};
```

4.3.2 Оптимистический подход

При данном подходе возможен конфликт доступа, когда два клиента пытаются одновременно изменить содержимое одного объекта. В этом случае решение о том, какая версия объекта будет сохранена в базе, остается за приложением. Простейшая стратегия разрешения конфликта сводится к обновлению экземпляров объектов и повторному выполнению транзакции. Данный подход лучше всего работает, когда вероятность возникновения конфликта небольшая, а последствия его возникновения могут быть легко устранимы.

Проверка корректности производится на сервере, который для всех объектов, участвующих в транзакции, определяет версию экземпляра объекта, имеющуюся у клиента. Транзакция фиксируется только в том случае, если изменения произведены клиентом над последними версиями объектов. В этом метаобъекте не проверяется достоверность остальных (не модифицированных) объектов, к которым осуществлялись обращения в процессе выполнения транзакции. Эти объекты могут быть вытеснены из кэш памяти или заменены на более новые версии в процессе выполнения транзакции. Ниже приводится описание класса реализующего данный протокол:

```
class optimistic_metaobject : public basic_metaobject {
  protected:
    static unsigned n_nested_transactions;

  public:
    virtual void begin_read(hnd_t hnd);
    virtual void end_read(hnd_t hnd);

    virtual void begin_write(hnd_t hnd);
    virtual void end_write(hnd_t hnd);

    virtual void begin_transaction();
    virtual void end_transaction();

  protected:
    virtual int  get_transaction_object_flags(hnd_t hnd);

    virtual void commit_object_changes(hnd_t hnd);
    virtual void abort_object_changes(hnd_t hnd, abort_reason reason);

    virtual void release_transaction_object(hnd_t hnd);
};
```

Часто недостаточно гарантировать достоверность только измененных объектов. Так как изменения были рассчитаны исходя из значений других объектов, то иногда необходимо потребовать, чтобы экземпляры всех объектов, к которым осуществлялся доступ в процессе выполнения транзакции, относились к последней версии. Для этого была разработана версия оптимистического протокола, гарантирующего повторные чтения объектов (то есть идентичность состояния объекта в процессе всего выполнения транзакции).

Для этого осуществляется фиксация в кэш памяти всех (а не только измененных) объектов, к которым происходили обращения в процессе выполнения транзакции. По завершению транзакции на сервер отсылается не только измененные объекты, но и идентификаторы немодифицированных объектов, к которым осуществлялись обращения. Сервер проверяет достоверность всех объектов и фиксирует транзакцию только в том случае, если ни один из этих объектов не был изменен за время выполнения данной транзакции. Этот класс переопределяет метод `begin_read` базового класса `optimistic_metaobject`:

```
class optimistic_repeatable_read_metaobject : public optimistic_metaobject {
  public:
    virtual void begin_read(hnd_t hnd);
};
```

4.3.3 Пессимистический подход

Недостатками оптимистического подхода являются возможность возникновения конфликта и отсутствие универсального алгоритма для его разрешения. В случае, когда вероятность возникновения конфликта достаточно велика или не допустимо прерывание транзакции в результате конфликта, становится выгоднее использовать пессимистический протокол. Пессимистический протокол основан на блокировании объектов перед использованием. Для обеспечения сериализуемости транзакций необходимо, чтобы наложенные блокировки сохранялись до конца транзакции.

В простейшей версии метаобъекта, реализующей пессимистический протокол синхронизации, осуществляется эксклюзивная блокировка только изменяемых объектов. Объекты остаются блокированными и фиксированными в кэш памяти до конца транзакции. Однако объекты, доступ к которым осуществляется только на чтение, не блокируются и могут быть изменены другими транзакциями. Не модифицированные объекты могут быть вытолкнуты из кэш памяти в результате работы алгоритма замещения или по получению сообщения об устаревании объекта (если для этого объекта нет активных методов). При фиксации транзакции происходит автоматическое разблокирование измененных объектов.

Так как читаемые объекты не блокируются, возможно возникновение конфликта. Допустим метод, осуществляющий модификацию объекта, был позван из метода обращающемуся к объекту только для чтения.

Пусть в момент посылки запроса на блокировку объекта пришло сообщения об изменении объекта другим клиентом. Удалить устаревший экземпляр объекта из кэш памяти не возможно, так как имеется активный метод, работающий с данным объектом. Таким образом мы имеем дело с конфликтом, который не может быть автоматически разрешен. Для предотвращения такой ситуации необходимо, чтобы все методы, в результате выполнения которых объект может быть изменен, были объявлены как модификаторы (без спецификатора const). Для задания поведения приложения при возникновении конфликта в данном метаобъекте определяется виртуальный метод `solve_write_conflict`, который зовется при обнаружении конфликта. Если этот метод возвращает значение true считается, что конфликт успешно разрешен в пользу объекта данного приложения. Таким образом, при фиксации транзакции изменения, сделанные данным приложением, затрут изменения, произведенные до этого другой транзакцией. При возврате значения false происходит откатка текущей транзакции. Реализация метода `solve_write_conflict` в данном метаобъекте расматривает такую ситуацию как фатальную ошибку и вызывает аварийное завершению программы.

При использовании пессимистического протокола возможно использование блокировок без ожидания. Для этого в метаобъект добавлено поле, определяющее атрибуты блокировки. При использовании блокировки без ожидания и невозможности сразу же установить блокировку зовется метод `solve_lock_conflict`, который определяет дальнейшую последовательность действий. Например, этот метод может проинформировать пользователя о том, что объект уже используется другим приложением. Система будет продолжать попытки блокировать объект до тех пор пока метод `solve_lock_conflict` возвращает значение true. При возврате false система абортирует транзакцию.

Ниже приводится описание класса, реализующего описанное поведение:

```
class pessimistic_metaobject : public optimistic_metaobject {
  protected:
    int lck_attr; // attributes of locking (lck_nowait,...)

    virtual void storage_lock(hnd_t hnd, lck_t lck);

    //
    // This function is called when modification method was called from
    // readonly method and object found to be deteriorated after locking.
```

```
    // So it is impossible to update object instance (as there are active
    // invokations for this object). If this method returns 'true'
    // then version of object in databsae will be overwritten by
    // local object instance after commite of transaction.
    // If method returns 'false' transaction will be aborted.
    //
    virtual bool solve_write_conflict(hnd_t hnd);

    //
    // This method is called when lock request is failed
    //
    virtual bool solve_lock_conflict(hnd_t hnd);

    virtual int  get_transaction_object_flags(hnd_t hnd);

    virtual void release_transaction_object(hnd_t hnd) = 0;

    virtual void abort_object_changes(hnd_t hnd, abort_reason reason);

  public:
    virtual void begin_write(hnd_t hnd);

    pessimistic_metaobject(int attr = 0) { lck_attr = attr; }
};
```

При работе интерактивных программ довольно часто может возникнуть ситуация, при которой несколько пользователей одновременно пытаются редактировать один и тот же объект. При этом ни оптимистический подход, ни пессимистический подход не позволяют наиболее удобно организовать доступ к такому объекту. При использовании оптимистического подхода сильно ограничены возможности разрешения конфликта, так как конфликт обнаруживается только при завершении транзакции, когда пользователь уже закончил редактирование. С другой стороны использование пессимистического подхода и наложение блокировок может привести к такой ситуации, когда один пользователь начал редактировать объект и на неопределенное время блокировал всех остальных пользователей. Для синхронизации доступа при таком типе взаимодействия был разработан следующий подход, объединяющий свойства оптимистического и пессимистического подхода и названного ленивым пессимистическим подходом.

При использовании этого метаобъекта блокировка объекта осуществляется после выхода из метода, осуществившего изменение. Если объект был изменен другой транзакцией во время выполнения этого метода, то возникает конфликт, для разрешения которого зовется виртуальный метод `solve_write_conflict`. Если метод возвращает значение true, то изменения, внесенные в объект данным процессом, будут записаны в базу данных при фиксации текущей транзакции. Если метод возвращает false, то текущая транзакция абортируется. Реализация метода `solve_write_conflict` в данном классе в свою очередь зовет для разрешения конфликта виртуальный метод класса object `on_write_conflict` и возвращает результат выполнения этого метода. Ниже приведено описание этого класса:

```
class lazy_pessimistic_metaobject : public pessimistic_metaobject {
    virtual bool solve_write_conflict(hnd_t hnd);

  protected:
    virtual void begin_write(hnd_t hnd);
    virtual void end_write(hnd_t hnd);

    lazy_pessimistic_metaobject(int attr = 0)
    : pessimistic_metaobject(attr) {}
};
```

Для обеспечения повторного чтения объектов и предотвращения конфликтов был разработан метаобъект, реализующий следующую разновидность пессимистического подхода к синхронизации. Все участвующие в транзакции сохраняемые объекты блокируются. Объекты, доступ к которым осуществляется только для чтения, блокируются в разделяемом режиме, который допускает наложение разделяемых блокировок другими процессами. Перед модификацией объекты блокируется в эксклюзивном режиме, что исключает наложение разделяемых или эксклюзивных блокировок другими процессами. Блокировки автоматически снимаются при фиксации транзакции на сервере. Для этого серверу передаются не только тела модифицированных объектов, но и идентификаторы объектов, доступ к которым осуществлялся только для чтения.

Такой подход обеспечивает степень изолированности 4 — повторное чтение и сериализуемость. Однако применение данного протокола синхронизации ведет к значительному ограничению параллелизма при выполнении транзакций, увеличению числа сообщений, требующихся для установки всех необходимых блокировок, и, следовательно, к уменьшению производительности системы. Кроме того используемая стратегия может привести к возникновению тупиковых ситуаций, в случае, если объект был сначала блокирован в разделяемом режиме, а затем тип блокировки был увеличен до эксклюзивного. В этом случае два процесса, одновременно повышающие уровень своих блокировок для одного и того же объекта, могут взаимно блокировать друг друга.

Для предотвращения возникновения тупиков при повышении уровня блокировки с разделяемой до эксклюзивной, в данном метаобъекте предусмотрена возможность установки эксклюзивной блокировки даже при доступе только на чтение. Если через параметр `read_lock` конструктора передается значение `lck_exclusive`, то объект будет блокирован в эксклюзивном режиме при любом типе доступа. Метаобъект такого типа используется, например, для корневого объекта хранилища для синхронизации момента инициализации базы данных различными приложениями.

Описание класс приведено ниже:

```
class pessimistic_repeatable_read_metaobject : public pessimistic_metaobject {
  protected:
    lck_t read_lock;
  public:
    virtual void begin_read(hnd_t hnd);

    pessimistic_repeatable_read_metaobject(int attr = 0,
                                           lck_t read_lock_mode = lck_shared)
    : pessimistic_metaobject(attr) { this->read_lock = read_lock_mode; }
};
```

4.3.4 Указание метаобъекта для объекта

В системе GOODS объект принадлежит к конкретному классу и управляется конкретным метаобъектом. Как класс, так и метаобъект для данного объекта могут быть изменены. Понятия класса и метаобъекта ортогональны, то есть не зависят друг от друга. Объект произвольного класса может иметь любой метаобъект. Однако, часто стратегия управления доступом определяется единым образом для всех экземпляров данного класса. В таком случае удобно связать метаобъект с классом.

В GOODS допустимы три способа указания метаобъекта для объекта:

1. при определении класса

2. при создании экземпляра объекта

3. использование значения по умолчанию

При определении каждого класса указывается метаобъект, который будет использоваться для экземпляров создаваемых объектов этого класса, если только метаобъект явно не будет указан в конструкторе создаваемого объекта и не определено значение метаобъекта по умолчанию. Значение метаобъекта по умолчанию задается статической переменной, которая может быть изменена в любой момент, и определяет метаобъект для всех создаваемых впоследствии объектов, для которых метаобъект не указан в конструкторе явно.

4.4 Концепция транзакций в GOODS

Транзакцией в GOODS, также как и в большинстве других баз данных, называется атомарная последовательность действий, переводящая базу данных из одного непротиворечивого состояния в другое. Атомарность транзакции заключается в том, что все операции, входящие в транзакцию, либо должны быть выполнены, либо результаты всех этих операций должны быть отброшены. Определение начала и коннца транзакции выглядит в различных системах по разному. Так например, в интерактивном интерфейсе к СУБД Oracle SQL-Plus транзакция стартует неявно при первом обращении к базе и завершается явным выполнением оператора COMMIT или ROLLBACK. Во многих объектных базах данных необходимо явно указывать начало и конец транзакции. В последнем случае необходима поддержка вложенных транзакций, так как иначе невозможно объединить в одну транзакцию выполнение нескольких методов, содержащих операторы открытия и закрытия транзакции. Объемлющая транзакция фиксируется только после окончания всех вложенных подтранзакций.

Явное указание границы действия транзакции нарушает требование прозрачности работы с базой данной. Поэтому в системе GOODS была предложена альтернативная реализация аппарата транзакций. Вызов каждого метода рассматривается как вложенная подтранзакция. Транзакция автоматически фиксируется при завершении всех подтранзакций, то есть когда больше нет выполняющихся методов объектов GOODS. Все блокировки, наложенные на объекты за время транзакции, снимаются по завершению транзакции (это поведение может быть переопределено при создании пользователем собственных метаобъектов).

Иногда неявных средств управления транзакциями недостаточно. Для этого в GOODS предусмотрены методы для явного старта и завершения вложенной транзакции. Эта возможность позволяет объединять в одну транзакцию методы, позванные в разные моменты времени.

Все экземпляры объектов, измененные в процессе выполнения транзакции, остаются в кэш памяти процесса клиента до завершения транзакции, поэтому другие процессы не видят эти изменения до того момента, как транзакция будет зафиксирована. Подобный подход предъявляет повышенных требования к объему памяти клиента, однако существенно упрощает откатку и фиксирование транзакции на сервере. Так при откатке достаточно просто удалить все измененные объекты из кэш памяти клиента. Кроме того, так как все изменения осуществляются над локальными экземплярами объектов в процессе клиента, то серверу не надо изолировать объекты, модифицированные незавершившейся транзакцией, от других клиентов. Так как одной из основных задач при проектирование GOODS было перераспределение загрузки с сервера на клиентов, то для реализации транзакций была выбрана именно эта схема, хорошо сочетающаяся с общей схемой кэширования объектов в ООСУБД. Участвующие в транзакции объекты не могут быть замещены в кэш памяти до завершения транзакции.

Возможности отката транзакции в GOODS ограничены тем, что невозможно подменить объекты при активных методах, т.к. методы имеют прямой доступ к компонентам через this указатель. Поэтому в GOODS не поддерживаются откаты вложенных транзакций, а также точки отката. Возможен только глобальная откатка транзакции, при котором измененные экземпляры объекты базы данных будут удалены из кэш памяти клиента по мере завершения работающих с ними методов.

В GOODS поддерживаются распределенные транзакции, то есть транзакции затрагивающие объекты расположенные на нескольких серверах. Для фиксирования такой транзакции применяется двухфазная схеме фиксации транзакции. Управлением фиксацией распределенной транзакции осуществляет один из серверов участвующих в транзакции – координатор. В качестве координатора выбирается сервер с наименьшим идентификатором. Такой выбор координатора обусловлен тем, что при старте сервера (или при рестарте сервера после сбоя) он пытается связаться с серверами, имеющими меньший идентификатор, чем он сам (это сделана для того, чтобы предотвратить дублирование сообщений между серверами при старте). При рестарте сервера после сбоя происходит восстановление базы данных, при этом серверу необходимо связаться с координаторами всех глобальных транзакций, в которых данный сервер принимал участие. Если координатор имеет наименьший идентификатор среди всех процессов участвующих в транзакции, то любой сервер при рестарте всегда сначала установит соединение с координаторами глобальных транзакций.

Клиент не может выступать в качестве координатора, потому что сбой координатора в процессе фиксирования транзакции по двухфазной схеме может привести к блокированию всей базы данных. Действительно, суть двухфазной схемы фиксации транзакции состоит в том, что сначала все участники транзакции сообщают о готовности к фиксированию транзакции (то есть о готовности восстановить свою часть транзакции после сбоя). Если все участники прислали подтверждение о готовности, то транзакция считается глобально зафиксированной и всем участникам посылается сообщение о завершении транзакции. Однако, если после отправки сообщения координатору о готовности, сервер не получил подтверждения от координатора (в результате, например, выхода из строя последнего или сбоя в сети), то сервер не знает что делать дальше: он не может ни завершить локальную часть транзакции, ни откатить ее, так как не знает была ли транзакция зафиксирована или абортирована. Сервер должен ждать рестарта координатора для получения информации от координатора о статусе транзакции. Все это время объекты, участвующие в транзакции, остаются блокированными и нормальное продолжение работы с базой проблематично. Надежность сервера полагается гораздо выше надежности клиента и, к тому же, для сервера можно гарантировать рестарт после сбоя. Поэтому в качестве координатора выбирается один из серверов, а не клиент, хотя это и приводит к необходимости посылки дополнительного сообщения, как это будет показано в дальнейшем. Альтернативой могло бы служить выделение специального сервера для координации процесса фиксации распределенных транзакций, но такой подход был отвергнут, так как этот сервер может стать критическим местом для производительности и надежности системы.

Клиент начинает фиксацию транзакции с посылки координатору его части транзакции (измененных объектов с этого сервера) и вектора с идентификаторами других серверов, участвующих в этой транзакции. Координатор присваивает транзакции глобальный идентификатор, помечает в глобальном журнале транзакцию как не зафиксированную и заносит в локальный журнал свою часть транзакции. После этого координатор возвращает глобальный идентификатор транзакции клиенту и ждет сообщений о готовности от остальных серверов. По получению глобального идентификатора транзакции от координатора, клиент рассылает остальным серверам их части транзакции с указанием координатора транзакции и идентификатора транзакции, полученного от координатора. После этого клиент ждет подтверждения об окончании транзакции от координатора. По получению своей части транзакции от клиента, сервер заносит ее в локальный журнал транзакции и сообщает координатору о своей готовности к завершению транзакции. В этот момент данный сервер в состоянии восстановить свою часть транзакции. По получению сообщений о готовности к завершению от всех участвующих в транзакции серверов координатор помечает в глобальном журнале транзакцию как зафиксированную и рассылает серверам подтверждения на завершение транзакции. В ответ на это сообщение координатора серверы заносят изменения в свои хранилища и разблокируют участвовавшие в транзакции объекты. Одновременно с рассылкой разрешений на фиксацию транзакции серверам, координатор посылает инициировавшему транзакцию клиенту сообщение об успешном завершении трнзакции. Попытка клиента обратиться после этого к одному из участвоваших транзакции серверов будет блокирована до тех пор, пока данный сервер не завершит фиксацию своей части транзакции.

Для каждого участвующего в транзакции объекта клиент может определить набор операций, которые будут выполнены с данным объектом на сервере. Эти операции включают проверку версии объекта и автоматическое снятие блокировки с объекта по окончанию транзакции. Первая операция применяется при использовании оптимистического подхода для синхронизации транзакций. В этом случае блокировки на объект не накладываются, а при фиксации транзакции проверяется, что объекты не были за это время изменены другой транзакцией. Вторая операция используется для пессимистического подхода. При этом подходе объект перед использованием блокируется до конца транзакции. Для того, чтобы избежать явной посылки запросов на разблокирование объектов, в GOODS было решено включить возможность автоматического снятия блокировки объектов в обработчик транзакций.

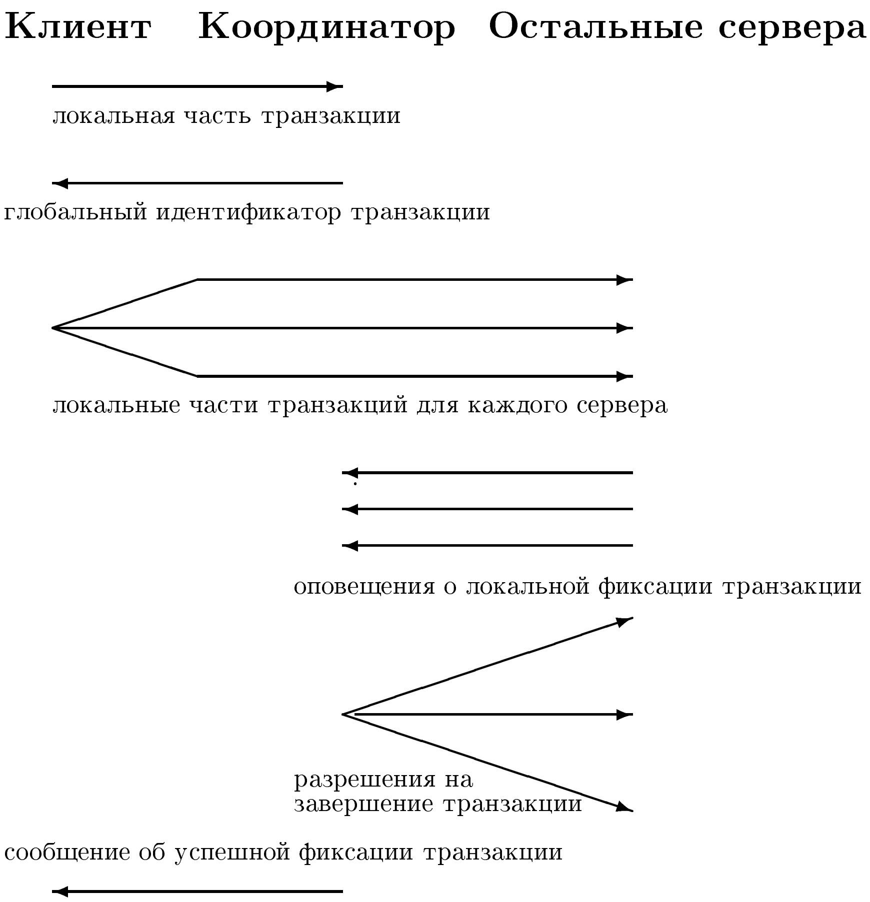

Рис. 4.2: Посылка сообщений между клиентом, координатором и остальными серверами

В транзакцию могут быть включены как модифицированные объекты, так и объекты доступ к которым осуществлялся только для чтения. Немодифицированные объекты могут быть включены в транзакцию для проверки того, что за время транзакции они не были изменены другим процессом (при оптимистическом подходе), или для снятия блокировок на чтение с этих объектов (при пессимистическом подходе).

Глава 5

Архитектура сервера GOODS

В основе проектирования сервера GOODS был заложен модульный принцип. Сервер был разбит на пять компонент, каждая из которых выполняет свои задачи и взаимодействует с другими компонентами посредством четко специфицированных интерфейсов. Для каждой компоненты был определен абстрактный класс, задающий интерфейс с этой компонентой. Сервер GOODS состоит из пяти различный управляющих модулей: менеджер доступа к объектам, менеджер распределения памяти, менеджер классов, менеджер файлового пула и менеджер транзакций. Возможность создания различных реализаций для каждой из компонент, использующих различные алгоритмы и стратегии, позволяет лучше подстраиваться под требования конкретных приложений и баз данных. Более того модульный принцип построения компонент образующих сервер, позволяет легко разрабатывать и исследовать новые алгоритмы для каждой из компонент.

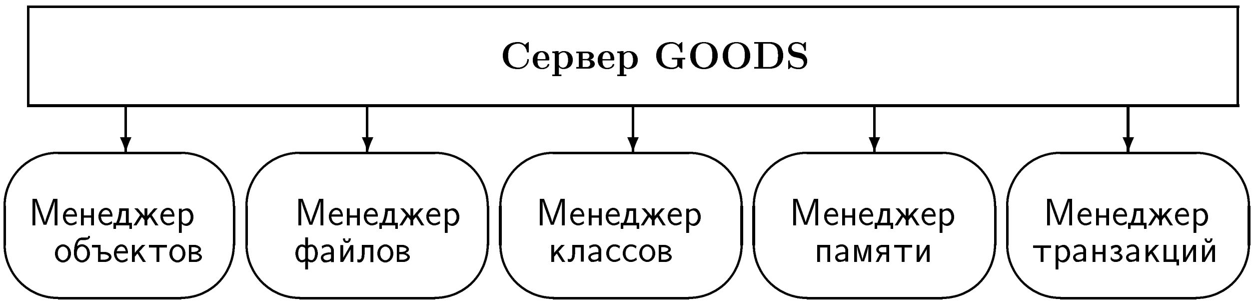

Рис. 5.1: структура сервера GOODS

Далее более подробно рассматриваются задачи и особенности реализации каждого из менеджеров.

5.1 Менеджер доступа к объектам

В задачи этого менеджера входит управление блокировками объектов и рассылка сообщений об изменении объекта всем клиентам, владеющим экземпляром данного объекта. Производительность этой компоненты особенно критична в работе сервера, так как запросы к менеджеру объектов выполняются очень часто.

Менеджер доступа к объектам использует хэш таблицу для доступа к информации об объекте. В хэш таблицу помещается информация обо всех блокированных или загруженных клиентами объектах. С каждым таким объектом связан список блокировок и список загруженных экземпляров объекта. Каждый клиент представлен структурой содержащей список наложенных процессом блокировок, список загруженных процессом экземпляров объектов и указателем на объект, блокировки которого ждет клиент. Таким образом мы имеем несколько отношение один-ко-многим:

процесс	$\longleftrightarrow\!\!\!\!\!\twoheadrightarrow$	блокированные объекты
процесс	$\longrightarrow$	объект, блокировки которого ждет процесс
процесс	$\longleftrightarrow\!\!\!\!\!\twoheadrightarrow$	загруженные объекты
объект	$\longleftrightarrow\!\!\!\!\!\twoheadrightarrow$	процессы наложившие на него блокировки
объект	$\longleftrightarrow\!\!\!\!\!\twoheadrightarrow$	процессы ждущие установления блокировки объекта
объект	$\longleftrightarrow\!\!\!\!\!\twoheadrightarrow$	процессы владеющие экземпляром данного объекта

Для представления этих отношений используются дополнительные структуры описывающие блокировки и экземпляры объектов и провязанные в списки. Общая схема организации структур данных менеджера по доступу к объектам представлена на рисунке 5.2.

Менеджер доступа к объектам играет свою роль в процессе освобождения памяти из под объектов, на которые нет ссылок (сборка мусора). Для проведения сборки мусора необходимо знать ссылки на какие объекты имеются есть у клиентов. Ситуация осложняется тем, что объекты могут меняться и у клиента может оказаться устаревшая копия объекта. Предположим, например, что в объекте A содержалась единственная

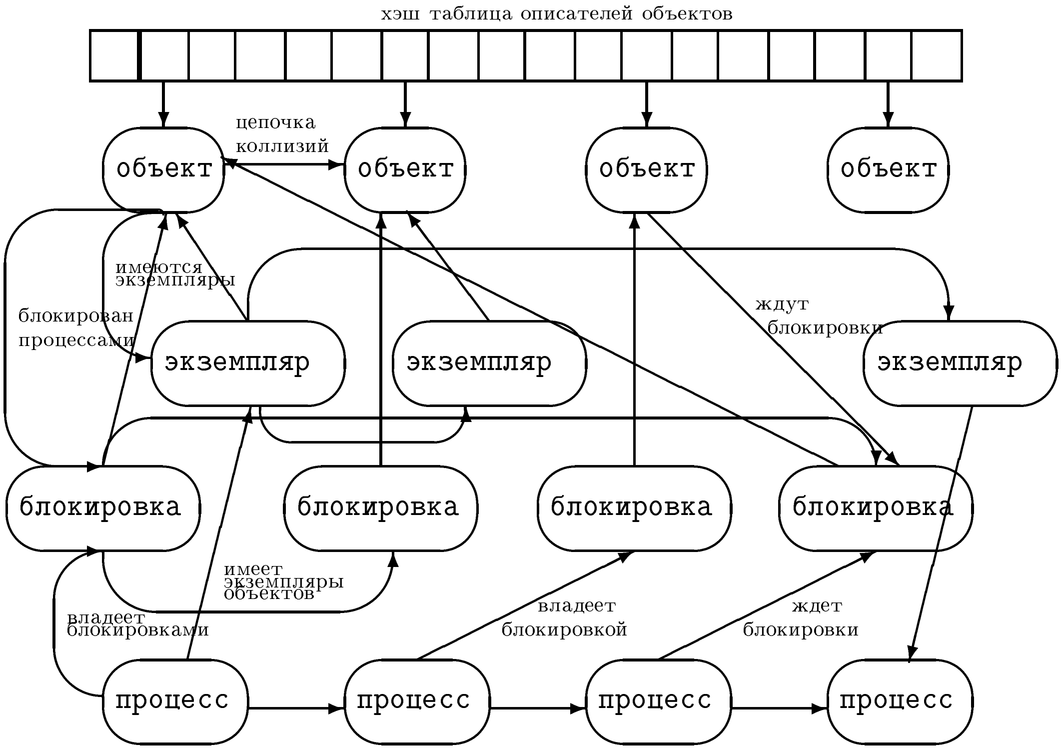

Рис. 5.2: Связь между объектами и процессами

ссылка на объект Б. Пусть экземпляры объекта А имелись у процессов П1 и П2. Пусть П1 заменил в А ссылку на Б ссылкой на какой-либо другой объект и зафиксировал транзакцию. Теперь в базе данных на объект Б никто не ссылается и он может быть вычищен сборщиком мусора. Но, допустим, у П2 сохранился устаревший экземпляр А, содержащий ссылку на Б (например сообщение об изменении объекта А еще не дошло до П2). Теперь следуя по ссылки из А процесс П2 может захотеть загрузить объект Б с сервера. Но такого объекта уже нет и на его месте может уже располагаться другой объект.

Решение, предлагаемое в GOODS, состоит в том, чтобы запоминать при модификации объекта старые значения измененных ссылок. Если у изменяемого объекта имеются экземпляры, находящиеся у других клиентов (за исключением клиента, инициировавшего транзакцию), то происходит оповещение всех этих клиентов об изменении объекта. После чего осуществляется покомпонентное сравнение старой и новой версии объекта. Все ссылки из старого объекта, отсутствующие в новой версии, добавляются в список, ассоциированный с описателем объектом. Этот список хранится до тех пор, пока все клиенты не удалят из своей кэш памяти устаревшие экземпляры объекта. После чего список можно удалить. При проведении сборки мусора менеджер доступа к объектам информирует сборщик мусора о всех объектах, экземпляры которых имеются у клиентов, а также о всех ссылках, которых больше нет в новых версиях объектов, но которые присутствуют в экземплярах объектов, имеющихся у клиентов. Таким образом менеджер доступа к объектам осуществляет контроль за корректностью версий экземпляров объектов, имеющихся у клиентов, и обеспечивает правильное функционирование сборщика мусора.

Менеджер доступа к объекту добавляет новый элемент в хэш таблицу (если его там уже не было) при загрузке экземпляра объекта клиентом или при установке блокировки на объект. Если клиент выбрасывает экземпляр объекта из кэш памяти (в результате прихода сигнала об изменении объекта или в случае освобождения места в кэш памяти алгоритмом замещения), то серверу посылается сообщение об этом событии и сервер не будет больше посылать клиенту сообщение об изменении объекта. При удалении в клиенте всех ссылок на объект, серверу также посылается сообщение и сервер удаляет связь между этим объектом и процессом.

5.2 Менеджер распределения памяти

Управление памятью в любой системе является важной и сложной задачей. СУБД и особенно ООСУБД предъявляют к алгоритмам распределения памяти дополнительные требования: эффективное отведение памяти под объекты с сильно отличающимися размерами (от десятков байт до нескольких мегабайт), соблюдение принципа локальности при размещении последовательно создаваемых объектов, предотвращении фрагментации памяти в процессе длительной (годами) эксплуатации, восстановлении после сбоя корректного отображения распределенной памяти. Мы также считаем необходимым использование автоматического сборщика мусора вместо явного освобождения памяти. Сборка мусора особенно критична для распределенных приложений. Алгоритм используемый для сборки мусора во многом определяет способ распределения памяти.

При выборе сборщика мусора и алгоритма распределения памяти в GOODS рассматривались две альтернативы: использование битовой карты распределения памяти и копирующий сборщик мусора с выделением памяти из непрерывного сегмента. Использование других алгоритмов, например основанных на поддержании списка свободных участков, не рассматривались из-за сложности поддержания корректности структур данных, используемых этими алгоритмами, в случае сбоя. Так например, каждую операцию со списком свободных элементов необходимо было бы оформлять отдельной транзакции, иначе бы при сбое возможно было разрушение списка. Рассмотрим теперь более подробно стратегии распределения памяти с помощью битовой карты и при использовании копирующего коллектора.

5.2.1 Распределение памяти с помощью битовой карты

Алгоритм распределения памяти с помощью битовой карты выделяет память квантами. Размер кванта выбирается таким образом, чтобы минимизировать количество памяти теряемой в результате неполного использования кванта, и с другой стороны, чтобы уменьшить время поиска свободного фрагмента. Также использование слишком маленького кванта может привести к образованию большого числа групп из одного или нескольких свободных квантов не достаточных для размещения объекта. Каждому кванту памяти соответствует бит в битовой карте. Таким образом при среднем размере объекта S и размере кванта Q, количество памяти занимаемое N объектами можно оценить как:

$$N * (S + Q/2) * (1 + 1/Q/8)$$

В этой оценке полагается, это средние потери на фрагментацию в результате округления размера объекта до величины кванта выделения памяти составляют половину от размера кванта. Коэффициент $1/Q/8$ показывает, какой размер памяти необходим для хранения собственно битовой карты, по отношению к размеру отображаемого ею сегмента памяти.

Минимум этого выражения достигается при $Q = \sqrt{\frac{S}{4}}$. Средний объект в типичном приложении содержит около 10 ссылок, поэтому учитывая, что ссылка в GOODS пакуется в 6 байт, примем $S = 64$. Тогда оптимальным значением кванта с точки зрения уменьшения внутренней фрагментации является $Q = 4$. К сожалению это значение не подходит по другим критериям. Поэтому для достижения приемлемого уровня внешней фрагментации значение кванта стоит принимать равным не менее 32 байт. При этом потери на внутреннюю фрагментацию составят

$$Q/2/S * 100\% = 16/64 * 100\% = 25\%$$

Такие большие потери памяти на внутреннюю фрагментацию являются основным недостатком метода распределения памяти с помощью битовой карты. В остальных отношениях метод выглядит достаточно привлекательным. Размер битовой карты равен $N * S/Q/8$, и при $N = 1000000$ составляет 250 Кбайт, то есть битовая карта почти всегда будет помещаться в памяти и поэтому поиск свободного фрагмента будет происходить достаточно быстро. Битовая карта также не требует никаких специальных действий для гарантирования восстановления после сбоя. Достаточно просто при восстановление объекта из журнала транзакций, заново отметить в битовой карте кванты, занятые объектом. Если в результате сбоя остались не удаленными какие-то объекты, то эти "объекты-призраки" будут удалены при следующей сборке мусора. Также удобна битовая карта и для сборки мусора, так как освобождение памяти, занимаемой объектом, производится простой очисткой битов в карте. Использование битовой карты также обеспечивает достаточно хорошую локальность ссылок, так как последовательно создаваемые объекты обычно выделяется непрерывно из одного фрагмента. Однако отсутствие средств дефрагментации все таки приводит к образованию разрозненных свободных участков.

5.2.2 Непрерывное распределение памяти и копирующий сборщик мусора

При использовании копирующего сборщика мусора вся память разбивается на сегменты достаточно большого размера и место под объекты отводится непрерывным образом в пределах сегмента. Когда при сборки мусора выясняется, что внутри сегмента образовалось много свободных участков, может быть инициирована перезапись оставшихся объектов сегмента в новый сегмент и освобождение данного сегмента. Для очень больших объектов (сравнимых или превосходящих размер сегмента) должна применяться другая стратегия выделения памяти. Как следствие такого подхода, почти полностью снимается проблема фрагментации. При этом необходимо использование дополнительной таблицы адресов объектов в файле для доступа к объектам (вторая степень косвенности). В противном случае пришлось бы отслеживать и изменять все ссылки на перемещаемые объекты.

Однако у такого подхода есть и ряд недостатков. Во-первых копирование объектов достаточно дорогостоящая операция. К тому же при копировании надо принять меры для сохранения целостности базы данных в случае аварийного сбоя. Так, например, гарантированная корректность после сбоя, произошедшего в процессе перемещения объектов, обеспечивается выполнением операции копирования объектов синхронно (то есть по завершению операции гарантируется, что объекты действительно были записаны на диск) и синхронным изменением адресов объектов в индексе. Однако синхронная запись очень дорогостоящая операция.

Наиболее интересным представляется стратегия распределения памяти, объединяющая копирующий коллектор и использование теневых страниц при внесении модификаций. Такой подход позволяет обойтись вообще без журнала транзакции и без специальной процедуры восстановления базы данных после сбоя. Все измененные объекты заносятся в базу данных дописыванием непрерывным образом в конец текущего сегмента. Старые экземпляры объектов будут вычищены в последствии сборщиком мусора. При такой схеме необходимо отразить в индексе изменение адресов модифицированных объектов. Основная идея состоит в том, чтобы хранить индекс в виде N-арного дерева. Каждый узел дерева удобно реализовать, как страницу из N указателей. Страницы, соответствующие листьям дерева, содержат указатели на объекты в файле базы данных. Остальные страницы содержат указатели на страницы-потомки. Страницы дерева являются обычным, с точки зрения сервера базы данных, объектами и при изменении также записываются на новое место в текущем заполняемом сегменте.

Дерево может быть симметричным или асимметричным. В первом случае для доступа к любому объекту требуется обращение к фиксированному числу страниц дерева. Так в предположении, что на странице дерева размещается 1024 указателей и имея ограничение на потенциальное число возможных объектов в хранилище $2^{32} = 4294967296$, высота дерева составит $\lceil \log_{1024} 2^{32} \rceil = \lceil \frac{32}{10} \rceil = 4$. Асимметричное дерево может быть построено по такому же принципу, как и массив inode занятых файлом блоков в системе Unix. При этом предполагается, что в типичной базе данных объектов будет гораздо меньше чем 2^{32}. Допустим 999 указателей на корневой странице указывают на страницы дерева напрямую адресующие объекты. Таким образом до первых $999 * 1024 = 1022976$ можно добраться всего за два обращения к страницам дерева. Однако для доступа к остальным объектам потребуется обращение к 5 страницам дерева. Возможна также реализация дерева, высота которого растет по мере увеличения числа объектов в хранилище.

Итак при фиксации транзакции происходит синхронная запись участвующих в транзакции объектов и изменившихся в результате перемещения объектов страниц индекса в непрерывный сегмент памяти. При этом в базе данных остаются старые копии объектов и страниц индекса. Перемещение любого объекта ведет к изменению всех страниц дерева индекса лежащих на пути от корня дерева к странице листу дерева, содержащей указатель на данный объект. Старый (теневой) образ корневой страницы соответствует состоянию базы до момента начала данной транзакции (страницы индекса, адресуемые из теневого корня, указывают на старые версии объектов). Для обеспечения целостности базы в случае сбоя необходимо, чтобы операция смены корня дерева выполнялась атомарно. Для этого обновление корневой страницы должно происходить особым образом, после записи всех остальных изменений на диск. В хранилище по фиксированному смещения располагается две копии корневой страницы, которые используются по очереди. В то время как одна копия изменяется, вторая является теневой. У корневой страницы есть специальное поле, позволяющее отличать текущую версию от теневой. Например, это может быть просто счетчик, увеличивающийся в начале каждой транзакции. При открытии хранилища выбирается та страница, значение счетчика у которой больше. Таким образом не требуется предпринимать никаких специальных действий для приведения базы данных в корректное состояние после сбоя. Допустим, что во время записи транзакции произошел сбой. Если сбой случился до записи корневой страницы на диск, то у теневой страницы значение счетчика больше и все измененные незавершившейся транзакцией экземпляры объектов недоступны и будут вычищены сборщиком мусора. Если же сбой произошел после записи корневой страницы на диск, то значение ее счетчика больше, чем значение счетчика теневой страницы, и все изменения, сделанный транзакцией, останутся в силе.

К сожалению этот подход не работает для распределенных систем. При фиксации распределенной транзакции каждый участвующий в транзакции сервер должен приготовиться к завершению транзакции (то есть быть готовым восстановить транзакцию в случае сбоя), доложить координатору глобальной транзакции о готовности и ждать от него подтверждения. Если все сервера, участвующие в транзакции, доложили о своей готовности, то координатор рассылает всем участникам транзакции разрешение закончить транзакцию. В противном случае координатор рассылает участникам глобальной транзакции команду откатить транзакцию. При использовании теневых страниц мы вынуждены будем блокировать сервер до получения подтверждения от координатора. Действительно, для гарантирования восстановления транзакции в случае сбоя мы обязаны записать новую версию корневой страницы. При этом, так как транзакция еще глобально не зафиксирована, необходимо изолировать внесенные транзакцией изменения от других процессов. Так как индекс отображения объектов уже изменился, этого можно добиться только приостановив обработку запросов сервером до получения подтверждения от координатора. Такое поведение сервера является неприемлемым для большинства систем, так как цикл опроса координатором по сети всех серверов, участвующих в глобальной транзакции, может занять значительное время.

5.2.3 Распределение памяти в GOODS

В GOODS в качестве базовой реализации была выбрана стратегия распределения памяти с помощью битовой карты. Битовая карта располагается отдельно от данных в собственном файле. Этот файл отображается на память и в дальнейшим все обращения к нему осуществляются посредством обычных указателей.

В базе данных GOODS используется двойная степень косвенности при адресации объекта. Использование лишней степени косвенности необходимо для изменения формата базы (изменение определения классов объектов, хранящихся в базе) в онлайновом режиме (то есть без остановки работы). Кроме того, использование дополнительной степени косвенности позволяет эффективно реализовать методы типа become языка Smalltalk. У каждого объекта есть уникальный идентификатор, который фактически представляет из себя индекс в таблице заголовков объектов. В заголовке объекта определяется размер, класс объекта и смещение объекта в файле. Индекс помещается в отдельном файле и также отображается в память для эффективной работы с ним. Содержимое индекса в памяти записывается на диск только в точках синхронизации или при откачки страницы индекса из оперативной памяти. Для поддержания корректности индекса в случае сбоя, для каждого объекта транзакции в журнал заносится его идентификатор, тип, размер и смещение в файле. Таким образом при прокрутке вперед (redo) журнала транзакций после сбоя, заголовок объекта может быть восстановлен для каждого объекта, участвовавшего в транзакции. А информация о других объектах не менялись со времени последней точки синхронизации.

Использование двойной степени косвенности требует наличия алгоритма распределения памяти как для заголовков объектов, так и для самих объектов внутри хранилища. Для отведения памяти под объекты используется битовая карта, каждый бит в которой соответствует кванту памяти. Размер кванта памяти взят равным 32 байтам.

Так как все заголовки объектов имеют одинаковый размер, то для управления распределением заголовков удобно воспользоваться списком свободных. Однако тут возникает проблема поддержания корректности списка. Либо мы должны каждую операцию над списком оформлять как отдельную транзакцию, либо быть готовыми к тому, что список может быть разрушен в результате сбоя. Так как использование синхронной записи изменений на диск резко уменьшает производительность системы, то было решено пойти по второму пути. При извлечении элемента из списка поверяется, что это элемент действительно помечен как свободный. Если это не так, то список считается некорректным, и новые заголовки отводятся в конце индекса, за последним использованным заголовком. Список свободных заголовков обновляется каждый раз по завершении сборки мусора. Если была зафиксирована некорректность списка, то сборщик мусора строит список заново.

5.2.4 Сборка мусора в GOODS

Автоматическая сборка мусора является необходимой принадлежностью современных объектных систем. Ошибки при явном освобождении памяти (повторное удаление объекта, ссылки на несуществующий объект) могут привести к непредсказуемому поведению программы и нарушить целостность системы. Использование сборки мусора особенно критично в распределенных системах, так как на объект могут существовать ссылки из различных приложений. Сборка мусора в распределенной системе требует наличия специального протокола синхронизации между различными хранилищами, так как в распределенной системе не возможно получить мгновенный снимок глобального состояния системы. Кроме того, надо учитывать, что объем базы данных может быть очень велик и сборка мусора может потребовать значительного времени. Поэтому единственно возможной альтернативой является фоновый сборщик мусора (то есть работающий параллельно с основной программой).

В системе GOODS сборка мусора ведется автономно на каждом сервере, с использованием одного из серверов в качестве координатора для синхронизации работы. В качестве координатора в GOODS выбирается первый сервер в списке серверов хранилищ базы данных. Для сборки мусора на каждом сервере создается отдельная задача, приоритет которой понижен по сравнению с остальными задачами. В GOODS используется классическая схема сборки мусора "mark-and-sweep", при которой сначала рекурсивно отмечаются все доступные из некоторого корня объекты, а потом вычищаются все непомеченные объекты. Этот подход, в отличие от использования счетчиков ссылок, гарантирует освобождение циклических структур.

Определим теперь процесс сборки мусора более формально. Рассмотрим множество всех экземпляров объектов в базе данных $S = \bigcup^i S_i$, где S_i – множество экземпляров объектов в хранилище i с выделенным корневым объектом r_i. Также рассмотрим множество $C = \bigcup^j C_j$, где C_j – множество экземпляров объектов имеющихся по данным сервера у клиента j (версии объектов у клиента и сервера могут отличаться). Далее рассмотрим объединение этих двух множеств $O = C \bigcup S$, и отношение "указывает на": $P \subseteq O \times O$. Множество корневых объектов R определим следующим образом: $R = \left(\bigcup^i r_i\right) \cup \left(\bigcup^j C_j\right)$. Множество достижимых объектов $\hat{O}$ строится на основании следующих двух правил:

1. $R \subseteq \hat{O}$,
2. $\forall a, b : a \in \hat{O} \wedge \langle a, b\rangle \in P \Rightarrow b \in \hat{O}$.

Результатом корректного выполнения программы является расширение множества C за счет вновь созданных объектов, а также загрузки клиентами объектов, на которые имеются ссылки из C:

$$\forall a : a \in C' \Rightarrow a \in C \vee (\exists b : b \in C \wedge \langle b, a\rangle \in P)$$

Таким образом, если объект не был включен в множество достижимых объектов в момент времени t, то он уже никогда не будет входить в это множество. Процесс сборки мусора заключается в освобождении объектов, не входящих в множество доступных объектов.

Процесс сборки мусора можно разбить на три этапа: инициирование, маркировка и завершение. Далее рассмотрим более подробно действия выполняемые на каждом из этапов.

Старт сборки мусора

После отведения определенного количества памяти сервер хранилища выдает запрос на начало сборки мусора. Все сервера, за исключением координатора, посылают координатору запрос на начало сборки мусора. Для исключения возможности одновременного выполнения более чем одной сборки мусора, разрешение на начало сборки мусора выдает только координатор. Разрешение на начало сборки мусора выдается в том случае, если в данный момент сборка уже не ведется. В противном случае запрос игнорируется. Если координатор решает начать сборку мусора, он запрашивает подтверждение о готовности к началу сборки мусора у всех серверов в базе данных. Получив такой запрос, сервер переходит в режим ожидания разрешения начать фазу маркировки объектов. При этом запоминаются версии всех объектов, измененных за это время. Если все серверы доложили о готовности, координатор рассылает им уведомлении о начале стадии маркировки. Сборка мусора не возможна, если активны не все серверы, указанные в списке серверов базы данных. При аварийном сбое одного из серверов и его последующем восстановлении и установки соединения с остальными серверами, идущая в этот момент сборка мусора прерывается и начинается заново, так как состояние восстановившегося после сбоя сервера не согласовано с остальными серверами, участвующими в сборке мусора (он потерял информацию о внешних ссылках из других хранилищ).

Маркировка объектов

Для маркировки объектов используется битовая карта, индексируемая идентификатором объекта (объекту с идентификатором N соответствует N-ый бит в битовой карте). Процесс маркировки можно представить как раскраску ориентированного графа объектов, вершинами в котором являются объекты базы данных, а ребра представляют ссылки между объектами. Объекты раскрашиваются в три цвета: белый, серый и черный. В начальный момент времени серым цветом отмечены корневые объекты и объекты, экземпляры которых загружены клиентами. Все остальные объекты имеют белый цвет. Обозначим как W множество объектов, помеченных белым цветом, G – множество объектов, помеченных серым цветом, и B – множество объектов, помеченных черным цветом. Алгоритм маркировки объектов можно описать следующим образом:

```
пока множество G не пусто
        для всех вершин o из G
                1. пометить серым цветом все вершины p белого цвета,
                   для которых ребро <o,p> принадлежит графу
                2. сменить цвет вершины o на черный.
```

По окончанию маркировки, множество достижымых объектов $\hat{O}$ соответствует множеству B, а множество подлежащих освобождению объектов – множеству W.

В каждом хранилище имеется корневой объект, который создается вместе с хранилищем и имеет фиксированных идентификатор. Используя метод `become` пользователь может изменить класс этого объекта на свой собственный. Множество G удобно реализовать в виде стека. В начале стадии маркировки в стек помещается ссылка на корневой объект и этот объект помечается в битовой карте.

Итерация цикла маркировки сводится к доставанию объекта из стека и сканированию всех содержащихся в нем ссылок. Формат хранения объектов в базе данных устроен таким образом, что все ссылки располагаются в начале объекта. Таким образом необходимо знать только количество ссылок в объекте. Так как в объекте допускаются массивы переменной длины, содержащие указатели, то общее количество ссылок можно определить просуммировав количество ссылок в фиксированной части объекта и количество ссылок в элементе массива переменной длины умноженное на размер этого массива. При сканировании ссылок проверяется, относится ли ссылка к этому хранилищу или это внешняя ссылка. В случае, если ссылка локальная, проверяется не отмечен ли уже адресуемый объект в битовой карте. Если объект не отмечен, то он отмечается и ссылка помешается в стек, в противном случае ссылка пропускается. Ссылки на объекты из других хранилищ помещаются в специальные буфера, отведенные для каждого хранилища базы данных. Перед занесением внешней ссылки в буфер проверяется, нет ли ее уже в буфере. Если ссылка уже есть, то повторного добавления не происходит. При полном заполнении буфера, все содержащиеся в нем ссылки отсылаются серверу соответствующего хранилища. Содержащиеся в буфере ссылки при этом не уничтожаются и участвуют в последующих проверках на наличие ссылки в буфере. При получении сообщения с внешними ссылками, сервер хранилища помещает эти ссылки в стек и отмечает в битовой карте в том случае, если адресуемые ими объекты не были уже отмечены.

Стадия маркировки продолжается до опустошения стека, то есть пока не останется не отслеженных ссылок. После этого сервер посылает сообщение координатору о локальном завершении сборки мусора и ожидает сообщения от сервера об окончании стадии маркировки и переходу к стадии освобождения непомеченных объектов. Если в процессе этого ожидания серверу приходят сообщения с внешними ссылками, маркировка продолжается и по ее окончанию координатору будет послано еще одно сообщение о локальном завершении маркировки.

Так как сборка мусора происходит параллельно с основным функционированием сервера, то требуются средства синхронизации их работы. Посмотрим, что могло бы произойти в отсутствии синхронизации. Пусть имеются два объекта A и B, таких что объект B содержит ссылку на объект C. Пусть эта ссылка является единственной ссылкой на этот объект. Теперь допустим, что сборщик мусора завершил обработку объекта A (пометил его черным цветом), но не успел отсканировать объект B. Пусть в этот момент произошла фиксация транзакции, которая занесла в объект A ссылку на объект C и удалила ссылку на C из объекта B. После продолжения сборки мусора и маркировки объекта B, сборщик мусора обнаруживает, что на объект C нет ссылок и вычищает его !

Для предотвращения вышеописанной ситуации, менеджер объектов сохраняет версии объектов, изменившихся за время сборки мусора (с момента получения запроса от координатора о готовностисервера к сборке мусора и до окончания стадии маркировки). Таким образом, сборщик мусора всегда работает с расширением множества $\hat{O}$, соответствовашему моменту начала сборки мусора.

При маркировки объектов необходимо учитывать экземпляры объектов, имеющиеся у клиентов. Дело в том, что у клиентов могут находится старые версии объектов. Пусть в объекте X содержится единственная ссылка на объект Y и экземпляр объекта X имеется у клиента C. Пусть теперь другой клиент C1 изменяет объект X, удаляя ссылку из него на объект Y. После фиксации транзакции клиента C1 в базе данных больше не остается ссылок на объект Y и он может быть вычищен сборщиком мусора. Однако, так как клиент C по прежнему обладает старой версией объекта X, содержащей ссылку на объект Y, то он может послать запрос серверу на загрузку объекта Y, в то время как тот уже был удален !

Предотвратить вышеописанную ситуацию можно несколькими способами:

- Запоминать ссылки, содержащиеся во всех объектах, посылаемых клиентам. Информацию об этих ссылках нужно хранить до тех пор, пока объект находится в кэш памяти клиента.

- При изменении объекта запоминать все ссылки, которые отсутствуют в новой версии объекта. Эту информацию нужно хранить до тех пор, пока все клиенты не обновят версии этого объекта в своей кэш памяти.

- Дождаться пока у клиента не останется устаревших экземпляров объектов.

- Запросить у клиентов, имеющих устаревшие объекты, информацию о ссылках из этих объектов.

Первый способ требует сканирования каждого объекта перед отправкой его клиенту, а также существенного увеличения объема информации хранящейся на сервере про каждый экземпляр объекта, загруженного клиентом.

Второй способ требует сканирования объекта гораздо реже, чем первый (обычно изменения объектов происходят реже, чем чтения), однако правильно определить измененные ссылки достаточно сложно, это требует применения квадратичного алгоритма. А при использовании простого покомпонентного сравнения, возможна ситуация когда, например, при вставке элемента в страницу Б дерева и соответственно сдвиге всех ссылок справа от вставляемого элемента, алгоритм определит, что поменялись почти все указатели.

Третий способ может привести к задержки процесса сборки мусора на неопределенное время, так как приложение клиента может не удалять объект из своей кэш памяти сколь угодно долго. При этом для определения момента, когда у всех клиентов не осталось устаревших объектов необходимо использовать достаточно сложный алгоритм.

И наконец четвертый способ плох тем, что требует посылки большого количества дополнительных сообщений по сети и также приводит к задержки сборщика мусора, связанной с ожиданием ответа от клиентов.

Так как второй способ требует меньших накладных расходов, то в GOODS было решено использовать именно его. Работа по оповещению клиентов об изменениях объектов и запоминание измененных ссылок выполняется менеджером доступа к объектам. Перед занесением измененного объекта в базу проверяется, имеются ли экземпляры этого объекта у других клиентов (кроме клиента инициировавшего транзакцию). Если таких процессов нет, то на этом работа менеджера доступа к объектам кончается. В противном случае осуществляется рассылка сообщений об устаревании копии объекта всем имеющим их клиентам. После этого производится покомпонентное сравнение новых и старых значений ссылок из объекта. При этом отслеживается случай вставки или удаления элемента из массива указателей. Предыдущие значения измененных ссылок, содержащихся в объекте, провязываются в список измененных ссылок объекта. Этот список поддерживается до того момента, пока все клиенты не обновят содержимое своих копий объектов, после чего список удаляется. Все объекты, экземпляры которых имеются у клиентов, заносятся в множество корневых объектов и помечаются серым цветом перед началом стадии маркировки.

Завершение сборки мусора

Для определения момента глобального завершения сборки мусора на всех серверах используется следующий алгоритм. Каждый сервер базы данных с идентификатором i поддерживает массив A с информацией о числе импортированных и экспортированных внешних ссылок. Для каждого сервера j в базе данных, компонента элемента этого массива $A[j].export$ содержит информацию о количестве ссылок на объекты в хранилище сервера j из объектов локального хранилища i, уже отосланных данному серверу (экспортируемые ссылки), а компонента $A[j].import$ содержит информацию о количестве ссылок на объекты в локальном хранилище сервера i из хранилища сервера j, полученных от этого сервера. Перед началом сборки в массив инициализируется нулями. При посылки координатору сообщения о локальном завершении маркировки сервером, в сообщение включается массив A.

Координатор, в свою очередь, поддерживает матрицу ссылок между серверами M. Пусть $i0$ – идентификатор координатора, тогда строка матрицы $M[i0]$ совпадает с массивом А координатора. Начальное состояние матрицы задается следующей формулой:

$$
\begin{aligned}
M[i,j].export &= \begin{cases} 0, & i \neq j \\ 1, & i = j \wedge i \neq i0 \end{cases} \\
M[i,j].import &= 0, \forall i, j
\end{aligned}
$$

При получении сообщения о локальном завершении сборки от сервера j, координатор заносит в строку j матрицы переданный в сообщении массив A сервера j. После этого производится проверка глобального окончания стадии маркировки на всех серверах. Для этого проверяется следующее условие:

$$
M[i,j].export = M[j,i].import, \forall i, j \tag{5.1}
$$

Покажем корректность этого алгоритма. Нам нужно дождаться ситуации, когда все сервера локально завершили сборку мусора, отослали сообщение об этом на сервер и не получали после этого сообщений с внешними ссылками от других серверов. Также требуется, чтобы все отправленные сообщения о внешних ссылках дошли до своих получателей. (Иначе может возникнуть ситуация, когда сервер А рапортовал о том, что он локально закончил сборку мусора, после чего сервер В послал А сообщение с внешними ссылками, и тоже рапортовал координатору об окончании маркировки. По получению сообщений от серверов А и В об окончании маркировки, координатор может решить, что маркировка завершена на всех серверах, хотя сервер А еще не обработал внешние ссылки, присланные сервером В.)

Первоначальные значения элементов матрицы гарантируют, что условие 5.1 не будет выполнено пока все сервера не пришлют сообщение о локальном завершении маркировки, так как до замещения строки j в матрице на вектор A сервера j, присланного в сообщении о завершении маркировки, $M[j,j].export \neq M[j,j].import$. После получения сообщений о завершении маркировки от всех серверов, выполнение условия 5.1

гарантирует, что все сообщения с внешними ссылками, посланные сервером i серверу j до рапорта координатору о локальном завершении маркировки на сервере i, были получены сервером j. Докажем от противного, что при выполнении условия 5.1 никакой сервер не мог бы получить сообщения с внешними ссылками от другого сервера после того, как посылал сообщения о завершении маркировки координатору.

Пусть k_1 – идентификатор сервера, получившего сообщение с внешними ссылками, уже после посылки координатору сообщения об локальном окончании маркировки. Тогда в силу выполнения 5.1 существует сервер k_2, пославший ему эти ссылки также после рапорта координатору об завершении маркировки. Но сервер может послать сообщение с внешними ссылками после локального окончания маркировки, только в том случае, если он сам получил сообщение от сервера k_2, содержащее внешние ссылки, и продолжил маркировку. К серверу k_2 применимы те же соображения, что и серверу k_1. таким образом мы получаем последовательность событий (посылка/принятие сообщений с внешними ссылками), связанных причинно-следственной связью. Так как количество элементов в этой последовательности конечно, должен существовать последний элемент в этой последовательности k_n, соответствующий серверу, получившему сообщение с внешними ссылками после локального завершения маркировки, но при отсутствии сервера, пославшего ему это сообщение после локального завершения маркировки (иначе бы последовательность была продолжена). Но это возможно только в том случае, если некоторый сервер x послал это сообщение до отправки сообщения координатору о завершении маркировки, и, соответственно, эти ссылки нашли отражение в векторе A сервера и в строке x матрицы M координатора. Но тогда не может быть выполнено условие 5.1. Противоречие.

Освобождение недоступных объектов

После определения координатором момента окончания всеми серверами стадии маркировки, всем серверам рассылается сообщение с разрешением перейти к стадии освобождения объектов. Для этого сервер сканирует битовую карту сборщика мусора и на основании ее строит новый список свободных объектов. Для каждого неотмеченного в битовой карте объекта по заголовку объекта выясняется был ли объект уже освобожден и, если нет, то освобождается участок занятой им памяти. После этого заголовок объекта включается в список свободных объектов.

Стадия освобождения объектов выполняется сборщиком мусора инкрементально, параллельно с выполнением менеджером распределения памяти запросов клиентов на отведение и освобождение памяти. Инкрементальность в данном случае обеспечивается тем, что сборщик мусора просматривает и освобождает объекты в пределах некоторой области. По окончании обработки этой области, осуществляющий сборку мусора процесс снимает блокировку с менеджера управления памяти, делая возможным обработку запросов клиентов. Для того, чтобы не освободить объекты, созданные за время проведения сборки мусора еще не зафиксированными транзакциями, все такие объекты при создании отмечаются в битовой карте маркировщика. Если в результате сбоя была обнаружена некорректность списка свободбых объектов, то список свободных элементов строится сборшиком мусора заново (в отличие от инкрементального зобавления элементом в обычном случае).

Сборщик мусора также используется для определения описателей классов, для которых в базе данных не существует экземпляров объектов. Подробнее об освобождении классов рассказано в следующем параграфе.

5.3 Менеджер классов

Менеджер классов управляет доступом к информации о классах хранимых в хранилище объектов. При записи объекта в базу поля переупорядочиваются таким образом, что ссылки всегда лежат в начале объекта, что упрощает сборку мусора и запоминание изменившихся ссылок. Сервер хранилища обращается к классу объекта только при проведении сборки мусора для определения числа ссылок из объекта. В остальном содержимое класса прозрачно для сервера, который работает с объектами только на нижнем уровне, как с последовательностью байт. Хранение классов в базе данных необходимо для поддержания целостности базы данных в процессе работы многочисленных клиентов с базой и изменением формата объектов при обновлении приложений.

В описателе класса хранится информация об имени классе, об именах и типах всех компонент класса и всех базовых классов. При описании типа компонент используются следующие атомарные типы: знаковое целое (1, 2, 4, 8 байт), беззнаковое целое (1, 2, 4, 8 байт), плавающее (4 или 8 байт), ссылка, а также структуры, массивы фиксированной и переменной длины, состоящие из компонент одного из вышеперечисленных атомарных типов. Под динамическим массивом понимается массив, расположенный в конце объекта, количество элементов которого определяется при создании объекта (таким образом размер самого объекта данного класса становится переменным).

В каждом объекте хранится идентификатор его класса длинной два байта. Это определяет ограничение на максимальное количество классов в хранилище – $2^{16} = 65536$. Классы адресуются в хранилище как обычные объекты, однако им соответствует фиксированный диапазон идентификаторов 1..65535 (0 не является допустимым идентификатором). Идентификатор класса в хранилище совпадает с идентификатором представляющего его объекта. Доступ к классам имеет только менеджер классов, который поддерживает список не использованных идентификаторов, заносит новые классы в хранилище и присваивает им идентификаторы, позволяет по идентификатору класса получить его дескриптор или изменить дескриптор существующего класса.

При сборке мусора менеджер классов отслеживает дескрипторы классов, на которые не ссылается ни один объект из хранилища и удаляет такие классы, таким образом предотвращая исчерпание пространства идентификаторов классов в результате интенсивной разработки новых версий классов. Однако, так как возможен сценарий при котором клиент экспортировал новый класс в базу данных, но еще не успел записать объекты этого класса, то для предотвращения удаления такого класса в результате сборки мусора, удалению подлежат только те классы, для которых не осталось использовавших их клиентов. Для этого каждому клиентскому процессу на сервере присваивается монотонно возрастающий идентификатор. При всех операциях с классами (создание, загрузка, модификация) указывается идентификатор процесса, производящего эту операцию. Для класса запоминается максимальный из идентификаторов использующих его процессов. Класс не может быть удален пока существует клиентский процесс с идентификатором меньшим или равным идентификатору, запомненном в классе.

При загрузке объекта из базы клиент смотрит на идентификатор класса загружаемого объекта. Идентификатор класса уникален для хранилища, которому принадлежит объект. Если клиент уже загружал объекты с таким идентификатором, то используя идентификатор класса как индекс в таблице отображения классов данного хранилища на локальные классы приложеня (будем для краткости называть ее таблицей БД→ЛОК), клиент просто получает указатель на локальный класс, соответствующий идентификатору класса хранилища. Если, однако, это первый объект с таким идентификатором класса с момента открытия сессии, то клиент посылает на сервер запрос с просьбой прислать описание класса, соответствующее данному идентификатору. По получению такого описания, клиент ищет среди локальных классов приложения класс с таким же именем. После этого происходит покомпонентное сравнение описания локального класса и класса хранилища. В случае, если описания не совпадают (есть по меньшей мере одна компонента имя или тип которой различаются в определении локального класса и класса хранилища), то считается что в приложении определена новая версия класса. По дескриптору локального класса и класса хранилища строится специальный дескриптор псевдокласса, задачей которого является преобразование объекта из старого формата (описываемого дескриптором класса хранилища) в новый формат (соответствующий локальному классу). Этот дескриптор записывается в таблицу БД→ЛОК по индексу равному идентификатору класса в хранилище. После этого описатель локального класса отсылается на сервер, фиксируется менеджером классов и присвоенный ему идентификатор возвращается клиенту. Клиент заменяет в загруженном объекте старый идентификатор класса на новый, таким образом, если объект будет изменен и отослан обратно на сервер, он будет записан уже в новом формате.

Рассмотрим более подробно процесс преобразования экземпляра объекта при изменении класса объекта. Как уже было сказано выше, при этом создается дескриптор псевдокласса, отвечающий за преобразование загружаемого объекта. Процесс конвертации объекта состоит из рекурсивного обхода компонент в соответствии с новым описанием класса и копирования с преобразованием компонент, для которых в старом описании класса найдена компонента с тем же именем. При этом допускаются любые виды преобразования скалярных компонент (целое в плавающее и наоборот, с потерей точности или с расширением), а также изменение размера фиксированных массивов, преобразованием фиксированного массива в динамический и наоборот. Единственным типом, который не может быть изменен при преобразовании, является ссылка. На рис. 5.3 приводится пример преобразования следующих классов:

```
class A { // old version
    int2  i;
    real8 r[3];
};
class B : public A {
    char  c;
    int4  j;
    struct {
        nat4 x;
        nat4 y;
    } coord;
};
```

```
class A { // new version
    real4 r[2];
    int4  i;
    int4  j;
};

class B : public A {
    struct {
        real8 x;
        real8 y;
    } coord[2];
};
```

При занесении нового объекта в базу данных клиент ищет в таблице отображения локальных классов на классы хранилища ЛОК→БД идентификатор класса сохраняемого объекта. Если объекты такого класса уже записывался на сервер, то в таблице будет указан идентификатор соответствующего класса в хранилище. Если объект такого класса пишется в базу данных впервые, то на сервер отсылается описатель локального класса сохраняемого объекта. Менеджер классов ищет класс с таким же описателем (с тем же именем и компонентами). Если такой класс найден, то клиенту возвращается его идентификатор. В противном случае, считается, что клиент определил новую версию класса, описатель

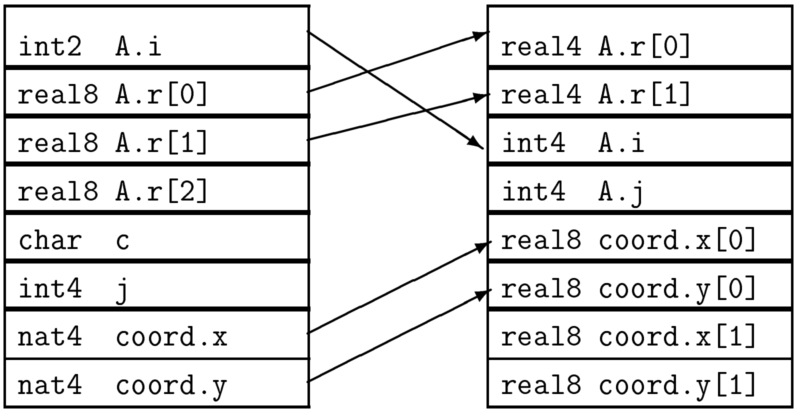

Рис. 5.3: Преобразование объектов

класса сохраняется в базе, ему присваивается идентификатор, который и возвращается клиенту. Полученный идентификатор клиент заносит в таблицу ЛОК→БД.

5.4 Менеджер файлового пула

Менеджер файлового пула управляет доступом к файлу, содержащему данные хранилища. Для абстрагирования от аспектов реализации файлов в конкретных операционных системах и связанных с этим ограничением (по размеру, положению на дисках…) в GOODS определен абстрактный класс файл с набором стандартных операций над файлами, учитывающим многозадачную специфику параллельного доступа к файлу. Для этого протокола разработано несколько системно-зависимых реализаций, а также реализация мультифайла – файла состоящего из нескольких физических сегментов на диске(ах). Мультифайл необходим для создания хранилища, размер которого превышает системные ограничения на размер файла, или при необходимости хранить различные сегменты хранилища в различных файловых системах (например из-за невозможности разместить файл целиком на одном диске). Кроме этого, протокол файла был расширен с целью поддержки отображаемого на память файла, для которого также существует несколько системно-зависимых реализаций.

Текущая реализация менеджера файлового пула в GOODS осуществляет доступ к файлу через страничный пул. Страничный пул представляет из себя кэш страниц, использующий дисциплину "наиболее давно использованный" (last recently used) для замещения страниц в пуле. Размер страницы пула и количество страниц в пуле задаются при создании данного менеджера. Для поиска страницы в кэш памяти используется хэш таблица, ключом в которой служит смещение страницы в файле. Менеджер файлового пула также отвечает за синхронизацию содержимого пула с содержимым диска.

Страничный доступ к файлам и кэширование страниц позволяет существенно повысить производительность при работе с диском. Однако, следует заметить, что при этом дублируется аналогичные функции файловой системы, что может привести к недостаточно эффективному использованию памяти сервера и лишнему копированию данных (с диска в системную кэш, из системной кэш памяти в файловый пул). Поэтому можно порекомендовать работать напрямую с устройствами (дисками) в обход файловой системы.

Так как чтение данных с диска требует обычно достаточно много времени, то для эффективной работы системы необходимо, чтобы одновременно с чтением данных с диска сервер мог выполнять запросы других клиентов. Поэтому особое внимание при реализации менеджера файлового пула было уделено обеспечению возможности паралельного выполнения операций с файлом различными клиентами. При этом менеджер блокируется только на момент работы с внутренними структурами данных, а перед выдачей запроса на чтение объекта с диска блокировка снимается, позволяя другим задачи также осуществлять операции ввода/вывода.

Для предотвращения замещения алгоритмом управления кэш памятью страницы, на которую производится чтение или запись, страница на время выполнения запроса исключается из списка страниц, доступных алгоритму замещения. При этом возможна ситуация, когда все страницы в пуле заняты под операции ввода/вывода и алгоритм управления кэш памятью не может найти и освободить страницу для нового запроса. В этом случае (надо признаться, что вероятность такого события очень мала), менеджер ждет на специальном семафоре завершения хотя бы одной операции ввода/вывода.

Если одновременно несколько задач обращаются к одной и той же странице, то для их синхронизации в описателе страницы имеется семафор и счетчик числа задач, работающих со страницей. Алгоритм доступа к странице включает в себя следующие шаги:

1. При обращении к странице, происходит поиск ее в кэш памяти с использованием хэш таблицы.

2. Если страница отсутствует в кэш памяти, то находится свободная или замещается наиболее давно использованная страница (если она была изменена, то происходит ее запись в файл). Если выбор страницы для замещения не возможен, то задача засыпает на семафоре и будет разбужена при появлении страниц, доступных для замещения. У страницы устанавливается флажок занятости, снимается блокировка менеджера файлового пула и инициируется чтение страницы с диска.

3. Если страница найдена в кэш памяти, то проверяется ее состояние. В том случае, если установлен флажок занятости, то задача устанавливает флажок ожидания и засыпает на семафоре, ассоциированном со страницей, и будет разбужена по завершению операции чтения на страницу другой задачей.

4. Если счетчик задач, использующих данную страницу равен нулю, то описатель страницы вывязывается из LRU списка для предотвращения замещения страницы до окончания доступа. После этого происходит увеличение счетчика и выход менеджера файлового пула из критической секции. С этого момента доступ к странице может осуществляется задачами параллельно.

5. По завершению операции со страницей, менеджер файлового пула снова входит в критическую секцию, уменьшает на единицу счетчик числа задач и, в случае если его значение становится равным нулю, ввязывает страницу в LRU список. После этого проверяется установлен ли флаг ожидания, и если он установлен, то посылается сигнал всем ожидающим завершения операции чтения задачам.

5.5 Менеджер транзакций

Менеджер транзакций управляет фиксацией транзакций на сервере. Транзакции в системе GOODS различаются на локальные, то есть затрагивающие объекты только в одном хранилище, и глобальные - затрагивающие объекты в нескольких хранилищах и требующих специального протокола для синхронизации фиксации транзакции серверами этих хранилищ. Для глобальной транзакции назначается координатор из числа участвующих в этой транзакции серверов. Координатор присваивает транзакции глобальный идентификатор, уникальный для данного сервера. Таким образом глобальная транзакция уникальным образом идентифицируется парой: (идентификатор сервера-координатора, присвоенный координатором идентификатор транзакции). Организация выполнения транзакций и протокол синхронизации между клиентом инициатором транзакции и серверами, участвующими в транзакции, описаны в параграфе 4.4. Здесь мы более подробно остановимся на аспектах реализации механизма фиксации транзакции на сервере.

Для обеспечения целостности базы данных в случае сбоя, сервер ведет два журнала транзакций: локальный и глобальный. В локальный журнал заносятся непрерывным образом локальные транзакции и части глобальных транзакций, относящиеся к данному серверу. При записи транзакции в журнал вначале указывается заголовок транзакции, содержащий сведения о размере тела транзакции, идентификаторе глобальной транзакции (0 в случае локальной транзакции), координаторе глобальной транзакции, участвующих в транзакции серверах и контрольной сумме. После заголовка записываются участвующие в транзакции объекты. Перед каждым объектом записывается заголовок, содержащий информацию об его идентификаторе, типе, размере и позиции в файле.

В СУБД используются журналы транзакций организованные по трем различным принципам: только восстановление (redo-only), только откатка (undo-only) и откатка+восстановление (undo-redo). Для обеспечения целостности базы в случае сбоя при использовании первого подхода необходимо гарантировать, что никакие изменения не попадут в хранилище до записи в журнал трансакций (write-ahead log). Таким образом в случае сбоя в хранилище не могут оказаться изменения, сделанные не завершившимися транзакциями. Следовательно, при восстановлении после сбоя необходимо только восстановить все транзакции, которые были занесены в журнал с момента последней точки синхронизации (точки синхронизации обсуждаются ниже). Второй способ ведения журнала возможен, если перед фиксацией транзакции все внесенные ей изменения были записаны на диск. В этом случае необходимо только откатить изменения, сделанные незавершившимися транзакциями. При использовании третьего подхода сделанные транзакцией изменения могут быть записаны в хранилище до фиксации транзакции, но при этом в журнале должна быть сделана запись, позволяющая откатить сделанные изменения. При восстановлении после сбоя с использованием журнала такого типа необходимо откатить изменения, сделанные всеми незавершенными транзакциями, и восстановить завершенные.

Так как, в GOODS изменения, сделанные транзакцией, остаются в контексте выполняющего данную транзакцию клиента до момента фиксации транзакции, то не возникает потребность в откатке внесенных изменений при восстановлении после сбоя. Поэтому используется журнал типа "только восстановление". В локальный журнал заносятся только новые значения измененных объектов, необходимые для восстановления транзакции после сбоя. Перенос изменений в хранилище происходит только после записи транзакции в локальный журнал транзакций. Управление возвращается клиенту, инициировавшему транзакцию, только после завершения транзакции всеми серверами в ней участвовавшими. Завершение транзакции предполагает копирование всех измененных объектов в хранилище и снятие с этих объектов блокировок.

При записи определенного количества данных в локальный журнал, происходит процедура определения так называемой *контрольной точки* (check point). При этом содержимое файлового пула сервера записывается на диск, происходит запись в файл содержимого индекса объектов и карты распределения памяти. После того, как содержимое оперативной памяти синхронизировано с хранящимися на диске данными, можно удалить все записи из локального журнала транзакций, так как все изменения, сделанные зафиксированными к данному моменту транзакциями, записаны на диск. Удаление записей в журнале осуществляется изменением размера файла с журналом транзакций до нуля и увеличении на единицу номера версии журнала. Номер версии журнала заносится в начало журнала, а также указывается в заголовках всех транзакций. При восстановлении транзакций из журнала проверяется, что номер текущей версии журнала, записанный в начале файла, совпадает с номером, указанным в заголовке транзакции. Восстановление транзакции осуществляется только при совпадении этих номеров. Такая проверка номера версии необходима для предоставления возможности работы напрямую с разделами диска в обход операционной системы. При этом нет возможности изменить размер файла с журналом транзакций и факт прохождения сервером контрольной точки и удаления всех записей из журнала отмечается только увеличением номера журнала и установкой указателя текучей позиции в начало файла. Точно такая же последовательность действий выполняется и при закрытии сервера базы данных. Использование контрольных точек позволяет ограничить размер локального журнала транзакций и ускорить восстановления базы данных после сбоя.

В глобальный журнал заносится статус глобальных транзакций. Результат выполнения глобальной транзакции фиксируется только координатором этой транзакции. Если остальным участвующим в транзакции серверам необходимо узнать статус глобальной транзакции (например при восстановлении после сбоя), они посылают запрос координатору.

Основной проблемой при ведении глобального журнала статуса транзакций является определение момента, когда запись о статусе транзакции больше не нужна и ее место в файле может быть использовано для новых записей. Каждый из участвующих в транзакции серверов сохраняет свою часть глобальной транзакции в локальном журнале и готов восстановить ее в случае сбоя при получении подтверждения от координатора о том, что транзакция глобально зафиксирована. Таким образом, координатору необходимо хранить информацию о статусе транзакции до тех пор, пока все участвовавшие в транзакции сервера не пройдут через точку синхронизации (и таким образом удалят информацию из своих локальных журналов транзакций).

Для корректной работы алгоритма определения статуса глобальных транзакций необходимо гарантировать уникальность идентификаторов, присваиваемых координатором глобальным транзакциям. Действительно, предположим что координатор присвоил транзакции T1 идентификатор Tid и некоторые сервера занесли в свои локальные журналы информацию об этой транзакции (вместе с идентификатором транзакции). Пусть теперь на координаторе произошел сбой и после восстановления он использовал идентификатор Tid для другой глобальной транзакции T2 (так как транзакция T1 не была глобально зафиксирована, то информация об ее статусе и идентификаторе не попала в глобальных журнал координатора и, следовательно, координатор ничего не знает о существовании транзакции с таким идентификатором). Допустим транзакция T2 успешно завершилась и была зафиксирована. Пусть теперь один из серверов, записавших в свой журнал часть транзакции T1, пытается узнать у координатора, что ему делать с этой транзакцией. Так как в глобальном журнале координатора записано, что транзакция с идентификатором Tid успешно завершилась, то координатор сообщает серверу, что транзакция T1 зафиксирована!

Гарантировать уникальность идентификатора можно несколькими способами: например делать запись в глобальном журнале при каждом присвоении идентификатора. Однако это ведет к лишней синхронной записи на диск, которой в данном случае можно избежать. В GOODS запись текущего идентификатора в глобальный журнал осуществляется не при каждой транзакции, а только после выполнения определенного их числа DNTRANS (в данной реализации 1000). Если произошел сбой, то при восстановлении после сбоя к идентификатору последней записанной в глобальном журнале транзакции прибавляется константа DNTRANS. Идентификаторы всем последующим транзакциям назначаются начиная с этого числа. Таким образом гарантируется уникальность идентификатора при минимальных накладных расходах (одна запись на 1000 транзакций) и сохранении небольшого размера (4 байта) идентификатора транзакции, что важно для минимизации размера передаваемых по сети сообщений.

При восстановлении после аварийного сбоя, менеджер транзакций последовательно читает из локального журнала все корректно записанные транзакции. После чтения тела транзакции происходит вычисление контрольной суммы и сравнение ее с записанной в заголовке транзакции. При обнаружении ошибки дальнейшее чтение журнала прекращается (последняя операция записи в журнал могла не успеть завершиться из-за произошедшего сбоя). Если тело транзакции почиталось корректно, выясняется является ли данная транзакция глобальной. Для глобальной транзакции посылается запрос координатору транзакции о статусе этой транзакции. Если транзакция не была глобально зафиксирована, то данная транзакция не восстанавливается и сканирование журнала продолжается дальше. При восстановление транзакции происходит копирование содержимого всех участвовавших в транзакции объектов в хранилище, а также изменение индекса объектов и карты распределения памяти для корректного отображения восстановленных объектов.

Использование двухуровневой организации памяти в хранилище и сборка мусора приводят к следующей проблеме при восстановлении объектов из журнала транзакций. Допустим объект A был изменен одной из транзакций и поэтому помещен в журнал транзакций. Пусть после этого были удалены все ссылки на этот объект и сборщик мусора освободил его, т.е. вернул заголовок объекта в список свободных заголовков и освободил память из под объекта, очистив в битовой карте распределения памяти соответствующие биты. Освобожденная память могла быть использована для создаваемых объектов. Далее пусть произошел сбой. В процессе восстановления из журнала транзакций, удаленный объект А будет заново занесен в хранилище. Теперь возможна ситуация, когда отведенная под этот объект память пересекается с областью памяти, занимаемой объетом Б, созданным после удаления объекта А. Пока в этом нет ничего критического, так как более новый объект был занесен в журнал после объекта А, то и восстанавлен он будет позже объекта А, затерев ту часть объекта А, с которой он пересекается. Однако теперь в хранилище два заголовка объектов указывают на перекрывающиеся сегменты памяти. Так как на объект А не ссылок (раз он был удален сборщиком мусора), то при первой же сборке мусора после сбоя объект А будет повторно удален и, заодно, будет освобождена часть памяти, занимаемой объектом Б, на который имеются ссылки из других объектов. Мы получили нарушение целостности хранилища. Для предотвращения описаной выше проблемы, в GOODS при восстановление из журанла транзакций осуществляется проверка на пересечение занимаемой объектом области с восстановленными ранее объектами. Если такие пересечения обнаружены, то все более старые объекты удаляются. После завершения восстановления заново строится битовая карта распределения памяти на основании информации, хранящейся в заголовках объектов.

Выполнение фиксацией транзакции осуществляется агентом инициировавшего транзакцию клиента, при этом возможна параллельная фиксация транзакций (как локальных, так и глобальных) нескольких клиентов. Основное время при фиксации транзакции тратится на синхронную запись транзакции в журнал. Так как по скорости выполнения позиционирования и записи данных диск значительно уступают быстродействию процессора и оперативной памяти, то эффективная работа с диском возможна только при использовании буферизации и кэширования данных. Обычно операционная система не производит запись на диск немедленно по получению запроса. Вместо этого данные копируются в буфера операционной системы и управление возвращается программе. Программа может продолжать выполнятся, а данные будут записаны на диск позже, параллельно с работой программы. Однако в случае записи в журнал транзакций, сервер не имеет права продолжать работать с модифицированными транзакцией объектами до того момента, как транзакция будет записана на диск.

В GOODS было уделено особое внимание повышению эффективности выполнения синхронных операций записи в журнал транзакций. Многие операционные системы (например все современные версии Unix) предоставляют средства для записи сразу нескольких сегментов памяти одним запросом (readv и writev в Unix). Идея оптимизации синхронной записи в файл состоит в том, что пока выполняется один запрос на запись, запросы других задач (агентов клиентов), поступившие за это время, могут быть объединены в единый запрос, время выполнения которого будет значительно меньше, чем суммарное время выполнения каждого запроса по отдельности. Это объясняется тем, чти при записи на диск значительная часть времени уходит на подвод головки к нужному сектору и цилиндру. При последовательной записи на диск нескольких сегментов, мы устраняем эти расходы для всех сегментов кроме первого. Кроме того, экономия получается за счет уменьшения числа системных вызовов.

Для работы этого алгоритма необходимо иметь циклический буфер е из двух синхронизирующих объектов типа "событие". Пусть i – указатель текущей позиции в этом буфере. В начальном состоянии первое событие в буфере находится в сигнальном состоянии (signaled). При получении запроса на запись проверяется, является ли этот запрос первым в списке отложенных запросов (то есть до него не было отложенных запросов). Если это так, то происходит смещение текущей позиции в буфере `i = i + 1`, сброс события `e[i]` и ожидание наступления события `e[(i - 1) % 2]` (то есть завершения предыдущей операции записи). После завершения ожидания выполняется операция записи сегментов всех отложенных к данному моменту операций записи. После чего событие `e[i]` переводится в сигнальное состояние, пробуждая все ждущие завершения операции записи задачи. Если запрос не является первым отложенным запросом, то его сегмент добавляется к списку сегментов для комбинированного запроса, после чего задача ожидает наступления события `e[i]`, соответствующего завершению операции записи запроса, содержащего данный сегмент.

Этот алгоритм не требует специальной настройки и автоматически адаптируется к интенсивности потока фиксируемых транзакций. Чем больше выдается параллельных запросов на фиксацию транзакций, тем больше степень векторизации запросов, достигаемая данным алгоритмом (то есть количество запросов объединяемых в одну операцию записи). Произведенные измерения показали, что при использовании данного алгоритма достигается более чем двукратное увеличение производительности системы в случае интенсивного обновления объектов базы данных несколькими клиентами.

Еще один вид оптимизации операций синхронной записи на диск заключается в осуществлении записи квантами кратными размеру дискового блока. При этом драйверу не нужно читать модифицируемый блок с диска, что приводит к увеличению скорости выполнения запроса почти в два раза. Для обеспечения такой возможности в файле заведены два буфера размером с дисковый блок. Алгоритм буферизации объединен с алгоритмом накопления запросов и позволяет преобразовать несколько последовательных запросов на запись любого количества данных в файл в единую операцию, осуществляющую запись целого числа блоков. Буфера используются циклически для запоминания записываемых данных, занимающих часть от дискового блока.

5.6 Оптимизация пересылки объектов между сервером и клиентом

Основным запросом к серверу объектно-ориентированной СУБД является запрос на загрузку объекта. В отличие от реляционных СУБД, где результатом выполнения запроса является множество (таблица) строк, в ООСУБД единицей передачи информации между клиентом и сервером служит объект. Это ведет к значительному увеличению числа посылаемых сообщений между клиентом и сервером по сравнению с реляционными СУБД. Так как время передачи сообщения по сети достаточно велико, то посылка отдельного запроса для каждого объекта может привести к тому, что производительность системы станет неудовлетворительной. Более того, следует учитывать, что при посылки сообщения по сети значительную часть от времени передачи сообщения составляют накладные расходы, поэтому время передачи сообщения длиной 10 байт будет слабо отличаться от времени передачи сообщения длиной 100 байт.

Исходя из вышесказанного, ООСУБД для достижения высокой производительности должна обладать средствами для передачи в одном сообщении нескольких объектов (то есть осуществлять кластеризацию объектов). Большинство существующих ООСУБД позволяют программисту самому определять кластера объектов. В этом случае задачей системы является только расположение объектов, определенных в один кластер, рядом друг с другом на диске (обычно система пытается разместить их на одной странице) и пересылка всех объектов, входящих в кластер, в виде одного сообщения. К сожалению перекладывание на программиста задачи по группировке объектов в кластера приводит к нарушению принципа прозрачности для сохраняемых объектов, требует написания большого количества дополнительного кода и не всегда приводит к оптимальным результатам, так как стратегия кластеризации может зависеть от многих факторов, таких например, как загрузка сети, взаимное расположение объектов на диске, загрузка сервера...

Поэтому в системе GOODS было решено остановиться на автоматической кластеризации объектов. При этом вся ответственность за разбиение объектов по кластерам ложится на сервер. Для решения этой задачи возможно использование нескольких подходов:

- одновременная пересылка всех объектов на странице;
- включение в кластер всех объектов, на которые из запрошенного объекта имеются ссылки;
- включение в кластер некоторого подмножества множества объектов, полученного замыканием по ссылкам всех объектов, доступных из запрошенного, суммарный размер которых не превосходит некоторого значения;
- проведение статистического анализа последовательности запросов объектов и включение объектов в кластер на основании накопленной информации;
- объединение в один кластер всех объектов, записанных одной транзакцией.

Сравнение эффективности этих и других подходов представляет из себя отдельную интересную проблему, которая лежит за рамками данной работы. Так как основной целью при проектировании GOODS было предоставить базовый средства с возможностью их расширения, то в качестве стратегии кластеризации в GOODS был выбран простейший подход с включением в сообщение всех отсутствующих у клиента объектов, напрямую доступных из запрошенного объекта, суммарный размер которых не превосходит наперед заданной величины. Управление кластеризацией не входит в обязанности ни одной из компонент, на которые разбит сервер, и осуществляется агентом клиента – специальным классом, ответственным за получение запросов от клиента, инициирование выполнения запроса и отсылку результата выполнения запроса клиенту. Для отправки объекта(ов) клиенту в этом классе предусмотрен специальный виртуальный метод, который может быть переопределен для реализации собственной политики кластеризации. Таким образом, предоставляется возможность использования различных алгоритмов кластеризации для разных клиентов. Это особенно удобно в случае использования статистических алгоритмов, так как у каждого клиента может быть свой профиль запросов.

Глава 6

Интерфейс приложений с базой данных

6.1 Интерфейс с хранилищем

Система GOODS предполагает использование различных языков программирования для доступа к объектам базы. Объекты в хранятся на серверах в системно и языково-независимом формате. Интерфейс с приложениями определяется на уровне хранилищ. Ниже приводится описание методов абстрактного хранилища базы данных:

```
class dbs_storage {
  protected:
    //
    // Identifier of storage within database.
    //
    const sid_t    id;
    //
    // Interface with application
    //
    dbs_application* application;

  public:
    virtual opid_t allocate(cpid_t cpid, size_t size, bool aligned) = 0;
    virtual void   deallocate(opid_t opid) = 0;

    virtual bool   lock(opid_t opid, lck_t lck, int attr) = 0;
    virtual void   unlock(opid_t opid, lck_t lck) = 0;
```

```
//
// Get class descriptor by class identifier.
// Class descriptor is placed in the buffer supplied by application.
// If there is no such class at server buf.size() is set to 0.
//
virtual void   get_class(cpid_t cpid, dnm_buffer& buf) = 0;
//
// Store class descriptor at server receive it's identifier
//
virtual cpid_t put_class(dbs_class_descriptor* desc) = 0;
//
// Change existed class descriptor
//
virtual void   change_class(cpid_t cpid, dbs_class_descriptor* desc) = 0;

//
// Load object from server into the buffer supplied by application.
// Before each object dbs_object_header structure is placed.
// If there is no such object at server then "cpid" field of
// dbs_object_header is set to 0.
//
virtual void   load(opid_t* opid, int n_objects,
                    int flags, dnm_buffer& buf) = 0;

virtual void   load(opid_t opid, int flags, dnm_buffer& buf) = 0;


//
// Inform server that client no more has reference to specified object
//
virtual void   forget_object(opid_t opid) = 0;
//
// Inform server that client no more has instance of specified object
//
virtual void   throw_object(opid_t opid) = 0;

//
// Initiate transaction at server. Allocate place for transaction header
// in buffer. All objects involved in transaction should be appended
// to this buffer.
//
virtual void   begin_transaction(dnm_buffer& buf) = 0;

//
// Commit local transaction or part of global transaction
```

```
    // at coordinator server. In latter case coordinator will return
    // transaction indentifier which should be
    // passed to all other servers participated in transaction.
    // If transaction is aborted by server "false" is returned.
    //
    virtual bool   commit_coordinator_transaction(int n_trans_servers,
                                                   sid_t* trans_servers,
                                                   dnm_buffer& buf,
                                                   tid_t& tid) = 0;
    //
    // Commit local part of global transaction at server.
    //
    virtual void   commit_transaction(sid_t coordinator,
                                      int n_trans_servers,
                                      sid_t* trans_servers,
                                      dnm_buffer& buf,
                                      tid_t tid) = 0;
    //
    // Wait completion of global transaction (request to coordinator)
    // If transaction is aborted by server "false" is returned.
    //
    virtual bool   wait_global_transaction_completion() = 0;

    //
    // Establish connection with server, returns 'false' if connection failed
    //
    virtual bool   open(const char* server_connection_name) = 0;
    virtual void   close() = 0;

    dbs_storage(sid_t sid, dbs_application* app)
    : id(sid), application(app) {}
};
```

Класс dbs_application представляет из себя интерфейс к приложению для сервера хранилища. Так как система GOODS представляет из себя конфигурацию с пассивным сервером и активными клиентами, то набор операций, которые инициируются сервером в приложении клиента, очень ограничен. В класс dbs_application входит метод invalidate, посредством которого сервер сообщает клиенту об изменении в хранилище объекта, экземпляр которого имеется у клиента. В этом интерфейсе также определяется метод для обработки ситуации прерывания сессии сервером. Интерфейс dbs_application реализуется конкретным классом, определяющим специфичные для данной системы программирования способы взаимодействия с хранилищем. Этот класс отвечает за синхронизацию доступа к хранилищу, преобразование объектов при посылки на сервер и при получении с сервера, обработку асинхронных сообщений от сервера, таких как оповещение об изменении объекта. В данный момент в GOODS разработан и реализован интерфейс только дла языка C++. Ниже приводится краткое описание методов абстрактного хранилища базы данных:

`virtual opid_t allocate(cpid_t cpid, size_t size, bool aligned)`

Запрос на создание объекта на сервере с указанным классом и размером. Возвращается идентификатор этого объекта, присвоенный ему на сервере.

`void deallocate(opid_t opid)`

Производит удаление объекта на сервере. В системе GOODS отсутствуют явные средства удаления объекта, вместо этого используется автоматическое освобождение недоступных объектов на сервера (сборка мусора). При сборке мусора на стадии маркировки определяются все объекты, которых нельзя достигнуть следуя по ссылкам из постоянных корневых объектов. Однако в некоторых случаях и без маркировки объектов можно утверждать, что на объект не может быть ссылок. Так например, при откатке транзакции должны быть удалены из базы данных все объекты, которые стали сохраняемыми в процессе выполнения данной транзакции. Именно для этого в интерфейсе предусмотрен метод явного удаления объекта.

`bool lock(opit_t opid, lck_t lck, int attr)`

Запрос на блокировку объекта. Тип блокировки может быть *разделяемый* (shared) или *исключительный* (exclusive). *Разделяемая* блокировка допускает установку *разделяемых* блокировок другими процессами, в том время как *исключительная* блокировка может быть установлена только одним процессом. Блокировка объекта в разделяемом режиме может быть повышена до исключительной блокировки. Параметр attr задает различные атрибуты блокировки. В данный момент поддерживается только атрибут `lckattr_nowait`, который определяет, будет ли запрос выполняться с ожиданием или без ожидания. В случае установки блокировки с ожиданием, запрос блокируется до того момента, когда появится возможность его выполнить. При использовании аттрибута `lckattr_nowait`, если запрос на блокировку не может быть удовлетворен немедленно (по причине блокирования объекта другими процессами), то сразу же возвращается значение `false`.

```
void unlock(opid_t opid, lck_t lck)
```

Снять блокировку с объекта или понизить уровень блокировки с исключительной до разделяемой. Параметр `lck` определяет тип оставшейся блокировки объекта. Так, чтобы понизить режим блокировки с эксклюзивного на разделяемый необходимо выполнить запрос: `unlock(obj, lck_shared)`.

```
void   get_class(cpid_t cpid, dnm_buffer& buf) = 0;
```

Загрузить описание класса с сервера по заданному идентификатору класса. Описатель класса помещается в предоставляемый приложением буффер.

```
cpid_t put_class(dbs_class_descriptor* desc)
```

Послать описание класса на сервер и получить присвоенный ему идентификатор. В случае, если описание класса совпадает с уже имеющимся в базе данных классом, возвращается идентификатор этого уже зарегистрированного класса. В противном случае описание нового класса заносится в базу данных и ему присваивается новый идентификатор, который и возвращается клиенту.

```
void change_class(cpid_t cpid, dbs_class_descriptor* desc)
```

Изменить описание класса на сервере. При этом идентификатор класса не меняется, а переписывается только его описание. Этот метод может быть использован для переименования полей класса (стандартный механизм автоматического отслеживания изменения схемы базы данных основан на отождествлении имен полей и поэтому не может быть использован для их переименования).

```
void load(opid_t opid, int flags, dnm_buffer& buf)
```

Загрузить объект или кластер объектов с сервера. Параметр flags определяет параметры загрузки объекта(ов). Поддерживаются два флага:

lof_cluster: Разрешается посылка кластера объектов. В некоторых ситуациях, например при перезагрузке объекта, клиент может не захотеть получать объекты, отличные от интересующего его объекта.

lof_copy: Клиент загружает новую копию экземпляра объекта, сохраняя старую копию. Эта информация необходима серверу для правильного отслеживания версий имеющихся у клиентов экземпляров объектов.

Объект(ы) загружаются в предоставляемый приложением буфер. Количество переданных объектов определяется приложением исходя из размера использованной части буфера.

```
void load(opid_t* opid, int n_objects, int flags, dnm_buffer& buf)
```

Этот метод во многом аналогичен предыдущему, кроме того, что позволяет запрашивать с сервера сразу несколько объектов. Этот метод может быть использован для формирования более эффективных запросов серверу.

```
void forget_object(opid_t opid)
```

Проинформировать сервер о том, что у клиента больше нет ссылок на данный объект. Сервер может больше не хранить информацию о наличие у клиента ссылки на данный объект. Этот метод вызывается клиентом не для всех ссылок, содержащихся в полученных от сервера объектах, а только для тех ссылок, по которым клиент осуществлял обращения и, следовательно, загружал адресуемые этими ссылками объекты. Эти объекты могли быть в последствии удалены из кэш памяти в результате прихода сообщения об устаревании объекта, но метод `forget_object` все равно будет вызван только после удаления ссылки. Такая стратегия используется для отслеживания сервером ссылок, скопированных клиентом в локальные переменные. При копировании ссылки происходит загрузка объекта и сервер запоминает информацию о наличие у клиента данной ссылки. Адресуемый этой ссылкой объект не будет освобожден сборщиком мусора до тех пор, пока у клиента имеется ссылка. При этом можно не посылать сообщения об удалении ссылок, по которым не было обращений, что значительно сокращает количество передаваемой по сети информации. При изменении объекта одним процессом, может оказаться, что у другого клиента в кэш памяти осталась старая копия объекта, содержащая ссылки, которых больше нет в базе данных. Сервер "знает" об этих ссылках, запоминая при записи модифицированного объекта старые значения измененных ссылок. Эта информация хранится сервером до тех пор, пока у клиентов имеются старые версии объекта.

```
void throw_object(opid_t opid)
```

Проинформировать сервера о том, что клиент выбросил экземпляр данного объекта из кэш памяти. Обладая такой информацией сервер может больше не посылать этому клиенту сообщения об изменении данного объекта. Если все клиенты удалили устаревшие экземпляры объекта из своей кэш памяти, то сервер может удалить информацию об измененных ссылках для данного объекта.

```
void begin_transaction(dnm_buffer& buf);
```

Проинформировать сервер хранилища об участие данного сервера в транзакции. При этом инициализируется предоставляемый приложением буфер для записи участвующих в транзакции объектов. Дописывание объектов в буфер осуществляется приложением без участия сервера хранилища. Перед каждым объектом должен располагаться заголовок определенного типа, в котором указывается идентификатор, класс, размер объекта, а также флажки, определяющие операции, выполняемые с данным объектом на сервере. При занесении в буфер объекта, который не был изменен, указывается только заголовок без содержимого полей объекта.

```
bool   commit_coordinator_transaction(int         n_trans_servers,
                                      sid_t*      trans_servers,
                                      dnm_buffer& buf,
                                      tid_t&      tid);
```

Отослать локальную транзакцию или часть глобальной транзакции серверу-координатору. В случае глобальной транзакции координатору сообщается информация об участвующих в транзакции серверах, а в ответ координатор возвращает присвоенный транзакции глобальный идентификатор. Метод возвращает `false` если транзакция была абортирована сервером. Возврат `true` для локальной транзакции означает, что транзакция была успешно зафиксирована на сервере.

```
void commit_transaction(sid_t       coordinator,
                        int         n_trans_servers,
                        sid_t*      trans_servers,
                        dnm_buffer& buf,
                        tid_t       tid);
```

Отослать серверу, отличному от координатора, его часть глобальной транзакции (объекты принадлежащие хранилищу этого сервера). Указывается идентификатор транзакции, координатор транзакции, а такцеччже идентификаторы всех остальных серверов, участвующих в транзакции.

```
bool wait_global_transaction_completion()
```

Дождаться сообщения от координатора о завершения глобальной транзакции. Возврат `true` означает, что транзакция была успешно глобально зафиксирована, а `false` – что транзакция была абортирована координатором.

6.2 Использование языка C++ для доступа к базе данных

В текущей реализации GOODS в качестве языка взаимодействия с базой был выбран C++. Это выбор был обусловлен тем, что C++ в настоящее время является наиболее распространенным объектно-ориентированным языком, позволяющим разрабатывать эффективные программы для практически любых разделов программирования: от системного программирования до систем искусственного интеллекта.

При проектирование интерфейса к языку C++ за основу были выбраны следующие положения:

- Интерфейс должен использовать только стандартные средства языка C++. Было решено отказаться от использования конвертеров (препроцессоров) или специализированных компиляторов, использующих синтаксические расширения языка для работы с сохраняемыми объектами. Такой подход нарушает принцип ортогональности свойства сохраняемости объектов, а также делает неудобной разработку программы. Так использование специализированного препроцессора ведет к тому, что пользователь редактирует одни исходники, а компилятор или отладчик работают с другими. Написание специализированного компилятора представляет из себя достаточно сложную задачу и ограничивает переносимость системы. Кроме того желательно предоставить программисту возможность работать в той среде программирования, к которой он привык. По этой же причине при проектировании интерфейса не использовались средства специфичные для конкретной среды (такие например, как извлечение информации о формате объектов из отладочной информации, генерируемой компилятором) или последние добавления к стандарту C++ (обработка исключительных ситуаций и получение информация о типе объекта во время выполнения программы), которые поддерживаются не всеми компиляторами.

- Интерфейс должен быть максимально защищен от ошибок программиста. Так как в результате отказа от использования специализированных средств, большая часть контроля за доступом к данным ложится на программиста, то необходимо минимизировать вероятность внесения ошибок и позволять обнаруживать большинство из них на стадии компиляции (статический контроль). Одной из основных проблем при отсутствии в языке метауровня (то есть средств получения информации о структуре данных и изменения правил доступа к ним) является получение информации о формате объектов и переопределение операций доступа к данным с отслеживание момента внесения изменений. Для решения первой проблемы в GOODS был разработан собственный протокол описания формата класса, который сделан максимально удобным для программиста. Вторая проблема решается с использованием "умных указателей", реализованных посредством шаблонов. Места внесения изменений явно указываются программистом с использованием для контроля механизма константных (`const`) спецификаторов языка C++. Такой контроль позволяет выявить все ошибки, связанные с неправильным доступом к данным, на этапе компиляции.

- Интерфейс должен быть системно-независимым и, в тоже время, предоставлять средства для доступа к информации о структуре классов, для организации многопоточных вычислений и синхронизации выполнения различных потоков. Для этого в интерфейс GOODS к C++ был включен ряд библиотек, инкапсулирующих системно-зависимые аспекты реализации.

К сожалению язык C++ плохо подходит для написания высокоуровневых приложений, так как страдает отсутствием стандартных библиотек и не обладает защищенностью от ошибок. Применительно к роли языка доступа к базе данным, C++ не хватает рефлективности (способности в терминах языка отражать свою структуру), автоматического управления памятью и встроенной поддержки многопоточности. Поэтому в GOODS были включены классы для поддержки многозадачности, рефлективности, сборки мусора и прозрачного доступа к объектам базы. Обсудим основные принципы реализации этих классов.

Для поддержки многозадачности разработаны классы *задача, семафор, событие, критическая секция*, обеспечивающие интерфейс управления задачами, независимый от используемой операционной системы. Для абстрактного интерфейса многозадачности создано три различный реализации:

1. Кооперативная многозадачность, использующая только стандартные функции библиотеки языка C setjmp и longjmp для переключения контекстов задач. Переключение контекста происходит либо в результате блокирования активной задачи или путем явного вызова перепланировщика.

2. Реализация, построенная на базе интерфейсов потоков (threads) стандарта Posix. Данная реализация будет работать на большинстве современных Unix платформ и предоставляет все преимущества выталкивающей (preemptive) многозадачности с возможностью задействовать несколько процессоров в многопроцессорной архитектуре.

3. Реализация, использующая функции интерфейса Win32 и работающая на платформах Windows-95 и Windows-NT.

При наличии нескольких параллельно выполняющихся процессов возникает задача синхронизации их работы. В объектно-ориентированном программировании обычно используется один из двух подходов к организации параллельных вычислений: неявная модель параллелизма, при которой выполнение каждого метода объекта трактуется как запуск параллельного процесса, и явное указание потоков выполнения. Модель с явным созданием процессов обладает большей гибкостью и способствует более эффективной реализации. Поэтому в работе рассматривается именно эта модель. Наиболее простым и универсальным способом синхронизации работы потоков в данной модели является представление объекта как критического ресурса, для которого в любой момент времени может выполняться не более одного метода. Данную стратегию легко реализовать посредством метаобъектного протокола. Однако возникает проблема "тупиковых ситуаций" — взаимной блокировки несколькими процессами друг друга.

Назовем программу состоящую из нескольких потоков управления *корректной*, если при ее работе невозможно возникновение тупиковых ситуаций. Пусть $C = \{C_1, C_2, C_3, \ldots C_{n_c}\}$ – множество всех классов объектов в программе, $O_i = \{O_i^1, O_i^2, \ldots O_i^{n_o}\}$ – множество экземпляров объектов класса C_i и $M_i = \{M_i^1, M_i^2, \ldots M_i^{n_m}\}$ – множество методов класса C_i. Предположим, что для каждого метода у нас имеется информация о всех объектах, к которым осуществляются обращения из этого метода и, что метод объекта может быть позван из любого потока управления. Пусть AO_i^j – множество всех объектов, которым посылаются сообщения из метода M_i^j. Рассмотрим направленный граф вершинами в котором являются объекты из множеств O_i. Для каждой вершины графа представляющей объект O_i^j проведем ребра $V_k = \langle O_i^j, O_{i_k}^{j_k} \rangle : O_{i_k}^{j_k} \in AO_i^j$. Тогда можно показать, что достаточным условием для корректности программы является отсутствие циклов в этом графе.

Однако далеко не всегда можно определить к каким конкретным экземплярам объектов возможно обращение из кода метода. В таких случаях единственной информацией, которой мы располагаем на основании статического анализа кода программы, является множество классов AC_i^j, которым (или выведенным из которых) могут принадлежать объекты, являющиеся адресатами сообщений, посылаемых при выполнении метода M_i^j (это множество классов получаются путем статического анализа типов всех переменных и выражений, используемых в данном методе для определения получателя сообщения). Рассмотрим два отношения частичного порядка над множеством классов. Первое отношение $<$ соответствует иерархии классов при наследовании. Если класс C_i выведен из классов $C_{i_1}, C_{i_2}, C_{i_3}, \ldots C_{i_n}$, то определим $C_i < C_{i_k}, \forall k \in [1..n]$. Теперь транзитивно замкнем это отношение, т.е. положим $C_i < C_j, \forall i, j : \exists k, C_i < C_k \wedge C_k < C_j$. Второе отношение $\rightarrow$ определим следующим образом: $C_i \rightarrow C_k$, если для некоторого метода j класса C_i выполняется одно из двух условий:

$$\begin{cases} C_k \in AC_i^j \\ \exists C_l : C_l < C_k \wedge C_l \in AC_i^j \end{cases}$$

Построим транзитивное замыкание отношения $\rightarrow$ положив $C_i \rightarrow C_j, \forall i, j : \exists k, C_i \rightarrow C_k \wedge C_k \rightarrow C_j$. Тогда необходимое условие корректности программы можно записать следующим образом: $\not\exists i, j : C_i \rightarrow C_j \wedge C_j \rightarrow C_i$.

Поскольку средства поддержки рефлективности компиляторами языка C++ сильно ограничены, то для обеспечения данной возможности нужна поддержка программиста. В GOODS был добавлен класс “описатель класса” `class_descriptor`, который содержит информацию о всех полях класса. Для этого требуется, чтобы каждый объект GOODS (т.е. объект, который может быть сохранен в базе) имел метод `describe_components`, который бы возвращал список описателей полей класса. Каждая структура используемая в качестве компоненты класса должна иметь метод

```
friend field_descriptor& describe_field(T)
```

где `T` – тип структуры.

Такие методы уже определены для всех базовых типов. Кроме того имеется ряд макроопределений для генерирования описателей компонент: FIELD, ARRAY, MATRIX и VARYING. Последняя макро нуждается в пояснении. Макро VARYING предназначена для определения массивов переменной длины, задаваемой при создании объекта. Для этого массив переменной длины должен располагаться в конце объекта. При выведении из класса, содержащего массив переменной длины, нельзя определить новые компоненты экземпляра класса. Для обычных компонент компилятор сам генерирует вызов соответствующих деструкторов при удалении экземпляра объекта. Однако в случае с массивами переменной длинны компилятор не знает границ массива, поэтому работа по удалению ссылок, содержащихся в компонентах массива переменной длинны, должна явно выполняться методом описателя этого класса, зовущимся из деструктора объекта.

Для инициализации памяти под объект при загрузке его из базы данных (прописывание указателей на виртуальные таблицы, ...) в момент регистрации класса указывается функция, ответственная за отведение памяти под объект. Обычно эта функция просто зовет специальный конструктор, передавая ему идентификатор объекта и описатель класса, а также размер создаваемого объекта. Возможно определение собственных методов занятия и освобождения памяти для конкретного класса.

Для получения информацию о статическом типе ссылки на объект GOODS, каждый класс GOODS должен определять функцию `classof`, единственный параметр которой позволяет выбрать необходимую функцию среди семейства совместно используемых `classof` функций. Результатом вызова этой функции является ссылка на описатель класса, соответствующего статистическому типу указателя аргумента функции.

Для упрощения регистрации класса в базе данных в GOODS было создано два макроопределения, определяющих прототипы и стандартную реализацию методов, предоставляющих метаинформацию о классе:

```
#define METACLASS_DECLARATIONS(CLASS, BASE_CLASS)                 \
    static class_descriptor self_class;                           \
    static object* constructor(hnd_t hnd, class_descriptor& desc, \
                               size_t varying_size);              \
    CLASS(hnd_t hnd, class_descriptor& desc, size_t varying_size) \
    : BASE_CLASS(hnd, desc, varying_size) {}                      \
    friend class_descriptor& classof(CLASS const*);               \
    virtual field_descriptor& describe_components()


#define REGISTER(CLASS, BASE, MOP)                             \
class_descriptor& classof(CLASS const*)                        \
{ return CLASS::self_class; }                                  \
object* CLASS::constructor(hnd_t hnd,                          \
               class_descriptor& desc, size_t var_len)         \
{ return new (desc, var_len) CLASS(hnd, desc, var_len); }      \
class_descriptor CLASS::self_class(#CLASS,                     \
                                   sizeof(CLASS),              \
                                   &MOP,                       \
                                   CLASS::constructor,         \
                                   &BASE::self_class)
```

Пояснить использование этих методов лучше всего на примере:

```
class Money {
  public:
    nat1 currency;
    nat8 ammount;

    field_descriptor& describe_components() {
        return FIELD(currency), FIELD(ammount);
    }
    inline friend field_descriptor& describe_field(Money& m) {
        return m.describe_components();
    }
};

class Account : public object {
  protected:
    Money total;
    Data  history[ACCOUNT_HISTORY_SIZE];

    char  name[1]; // name of the account

  public:
    METACLASS_DECLARATIONS(Account, object);

    static Account* create(char* name, Money initSum);

};

field_descriptor& Account::describe_components()
{
    return FIELD(total), ARRAY(history), VARYING(name);
}

REGISTER(Account, object, repeatable_read_pessimistic_scheme);
```

Макро `REGISTER(имя-класса, базовый-класс, метаобъект)` регистрирует класс с данным именем. При этом производится обход дерева описателей компонент класса, вычисляются некоторые атрибуты класс, такие например, как размер фиксированной и переменной части класса в базе данных и в памяти, количество ссылок в фиксированной и переменной частях класса, смещения компонент... После этого строится описатель класса для базы данных `dbs_class_descriptor`, который используется при загрузке и записи объектов этого класса в хранилище.

При создании объекта его конструктору передается указатель на свой описатель класса и возможно метаобъект. Таким образом объект всегда имеет ссылки на свой класс и метаобъект.

Для доступа к объектам вместо обычных указателей C в GOODS применяются так называемые "умные указатели" (smart pointers). Эти указатели реализованы как параметрические классы (templates) и обеспечивают прозрачную загрузку объектов из базы и автоматическое освобождение недоступных объектов. Для сборки мусора применяется алгоритм подсчета ссылок. В заголовке каждого объекта хранится счетчик ссылок, который увеличивается каждый раз, когда на объект заводится новая ссылка и уменьшается при уничтожении ссылки. При обращении по ссылки к объекту производится проверка наличия объекта в кэш памяти клиента, загрузка при необходимости объекта из хранилища и вызов методов метаобъекта для управления доступом к объекту.

К основным недостаткам сборки мусора с использованием счетчиков ссылок относится то, что такой сборщик не вычищает циклические структуры, то есть группы объектов ссылающихся только на самих себя и на которые нет ссылок извне. Это происходит потому, что счетчики ссылок всех этих объектов отличны от нуля. Однако в GOODS эта проблема не так остра, так как для загружаемых из базы данных объектов используется алгоритм замещения, выбрасывающий из памяти объекты, к которым давно не было обращений. Таким образом выбрасывание из памяти элементов циклических структур ведет к разрушению цикла и позволяет сборщику мусора удалить остальные объекты. В приложении A приводится определение классов реализующих ссылки к объектам GOODS.

К сожалению, использование шаблонов для реализации "умных указателей" не позволяет организовать статический контроль за совместимостью типов при присвоении указателей друг другу. Так как каждая конкретизация параметризованного класса образует свой класс, то шаблон, в качестве параметра которого указан класс B, выведенный из класса A, будет не совместим с шаблоном конкретизированным классом A:

```
class A : public object {
    ...
};
class B : public A {
    ...
};
```

```
foo(ref<A> ra, ref<B> rb) {
    ra = rb;
}
```

Для того, чтобы разрешить подобные преобразования, для "умных указателей" определен оператор присваивания, позволяющий присваивать указателю любой другой указатель, не зависимо от типа параметра. Контроль за корректностью преобразования осуществляется на стадии выполнения. В операторе присвоения проверяется, что тип присваиваемого объекта совпадет или выведен из типа, которым параметризован изменяемый указатель. Для получения описателя класса, соответствующего типу параметра указателя, используется полиморфная функция classof.

Так как некоторые компиляторы C++ компилируют тела встроенных функций параметризованного класса при первой подстановке шаблона, то для предоставления возможности написания взаимно рекурсивных структур (то есть классов, содержащих ссылки друг на друга) все встроенные методы "умных указателей" реализованы таким образом, чтобы сделать возможной компиляцию при неполном определении класса параметра, если от него имеется только предописание (forward declaration).

Для извлечения объектов из базы данных необходимо получить корневой объект. Доступ к остальным объектам в хранилище осуществляется точно также, как и к обычным объектам - путем обращения по ссылкам. Однако загрузка корневого объекта должна осуществляться особым образом (так как на него нет доступных ссылок). Каждое хранилище в базе данных GOODS имеет свой корневой объект, который может быть переопределен пользователем с использованием оператора `become`. Этот объект создается при инициализации хранилища и имеет предопределенный тип `storage_root`. Для получения ссылки на этот объект в базе данных имеется специальный метод:

```
void get_root(object_reference& ref, sid_t sid = 0)
```

Этот метод инициализирует ссылку идентификатором корневого объекта в хранилище. При этом немедленная загрузка объекта не производится, а откладывается до момента первого обращения к объекту. Так как корневой объект не может быть удален из хранилища, то серверу в данном случае не обязательно знать о том, это у клиента имеется ссылка на этот объект. При инициализации хранилища приложение обычно заменяет класс корневого объекта с `storage_root` на определенный пользователем класс. По умолчанию для корневого объекта используется метаобъект, реализующий монопольный доступ к объекту (то есть на объект будет наложена эксклюзивная блокировка при первом доступе к нему для модификации или только для чтения). Такой подход позволяет избежать одновременной замены корневого объекта различными процессами. Для определения того, что корневой объект уже был заменен, удобно воспользоваться виртуальным методом `is_abstract_root` класса object, который возвращает значение `true` только в том случае, если объект принадлежит классу `storage_root`. Таким образом, типичный сценарий начала работы приложения с базой данных может выглядеть следующим образом:

```
class appliction_root : object {
  public:
    ref<B_tree>    employees;
    ref<set_owner> departments;
...
    applicatoin_root() : object(self_object) {}
    void initialize() const;

  protected:
    METACLASS_DECLARATIONS(appliction_root, object);
};

void application_root::initialize() const
{
    if (is_abstract_root()) {
        modify(this)->become(new application_root);
    }
}


database db;

main()
```

```
{
    if (!db.open("enterprise.odb")) {
        console::error("Failed to open database\n");
    }
    ref<application_root> root;
    db.get_root(root);
    root->initialize();
    ...
    db.close();
}
```

Глава 7

Заключение

Обзор существующих коммерческих СУБД показывает необходимость использования более гибких средств для взаимодействия приложения с СУБД. В данной работе была предпринята попытка на основе анализа существующих подходов и алгоритмов управления доступа к данным создать такую метамодель, которая позволит реализовывать любую из известных стратегий управления транзакциями и параллельным доступом к объектам, а также создавать специфичные для конкретного приложения модели. Таким образом, разработанная система GOODS обладает базовыми возможностями ООСУБД (концепция минимального ядра), позволяющими с использованием метаобъектного протокола расширять функциональные возможности системы до необходимого пользователю уровня.

Для эффективного использования метаобъектного протокола необходимо тщательно разработать интерфейс и таким образом определить базовые классы, чтобы можно было легко создавать специфичные для приложения модели, наследуя основную функциональность из этих базовых классов и внося минимальные дополнения. В GOODS созданы стандартные метаклассы для реализации оптимистической и пессимистической модели синхронизации. Так же разработан гибкий протокол взаимодействия сервера с клиентом в процессе фиксации транзакции, обладающий высокой степенью общности и минимизирующий количество передаваемых сообщений между клиентом и сервером.

Система GOODS была построена по модульному принципу, с разбиением всех компонент системы на логически обособленные модули, взаимодействующие друг с другом посредством четко определенных интерфейсов. Это позволяет использовать различные реализации для каждого модуля с целью настройки системы под конкретные конфигурации для повышения производительности системы и удовлетворения специфических требований приложений. Экспериментируя с различными реализациями модулей, отвечающими за распределение памяти, обработку транзакций, и.т.д., можно оценить эффективность различных алгоритмов и их пригодность для различных типов приложений.

Работы над системой GOODS будут продолжены в следующих направлениях:

- интеграции в систему модели удаленного вызова объектов
- повышение степени масштабируемости системы
- повышение надежности и отказоустойчивости системы
- реализация интерфейсов к други языкам программирования

Приложение A

Определение объектных ссылок в GOODS

```
#ifndef __REFS_H__
#define __REFS_H__

#include "object.h"

class object_reference {
    friend class database;
  public:
    hnd_t get_handle() const { return hnd; }

  protected:
    hnd_t hnd;

    void unlink() {
        object_handle::remove_reference(hnd);
    }
    void link() {
        if (hnd != 0) {
            hnd->access += 1;
            if (!IS_VALID_OBJECT(hnd->obj)) {
                internal_assert(hnd->storage != NULL);
                hnd->storage->load(hnd);
                internal_assert(IS_VALID_OBJECT(hnd->obj));
            }
        }
    }
    object_reference(hnd_t hnd) {
        object_monitor::lock_global();
```

```
        this->hnd = hnd;
        link();
    }
    object_reference() {
        hnd = 0;
    }
    ~object_reference() {
        unlink();
        object_monitor::unlock_global();
    }
};


template<class T>
class read_access : public object_reference {
  private:
    object* obj;
  public:
    T const* operator ->() {
        return (T const*)obj;
    }
    read_access(hnd_t hnd) : object_reference(hnd) {
        hnd->obj->mop->begin_read(hnd);
        // object can be reloaded by begin_read
        obj = hnd->obj;
        object_monitor::unlock_global();
    }
    read_access(read_access const& ra) : object_reference(ra.hnd) {
        obj = ra.obj;
        obj->mop->begin_read(hnd);
        object_monitor::unlock_global();
    }
    ~read_access() {
        object_monitor::lock_global();
        // object can be changed by become operator
        hnd->obj->mop->end_read(hnd);
    }
};


template<class T>
class write_access : public object_reference {
  private:
    object* obj;
  public:
```

```
    T* operator ->() {
        return (T*)obj;
    }
    write_access(hnd_t hnd) : object_reference(hnd) {
        hnd->obj->mop->begin_write(hnd);
        // object can be reloaded by begin_write
        obj = hnd->obj;
        object_monitor::unlock_global();
    }
    write_access(write_access const& wa) : object_reference(wa.hnd) {
        obj = wa.obj;
        obj->mop->begin_write(hnd);
        object_monitor::unlock_global();
    }
    ~write_access() {
        object_monitor::lock_global();
        // object can be changed by become operator
        hnd->obj->mop->end_write(hnd);
    }
};


template<class T>
class ref : public object_reference {
  public:
    inline read_access<T> operator->() const {
        assert(hnd != 0);
        return read_access<T>(hnd);
    }
    inline friend write_access<T> modify(ref const& r) {
        assert(r.hnd != 0);
        return write_access<T>(r.hnd);
    }
    inline friend write_access<T> modify(T const* p) {
        assert(p != NULL);
        return write_access<T>(((object*)p)->get_handle());
    }

    inline bool operator==(object_reference const& r) const {
        return hnd == r.get_handle();
    }

    inline bool operator!=(object_reference const& r) const {
        return hnd != r.get_handle();
    }
```

```
inline bool operator==(T const* p) const {
    return p ? (hnd == ((object*)p)->get_handle()) : (hnd == 0);
}

inline bool operator!=(T const* p) const {
    return p ? (hnd != ((object*)p)->get_handle()) : (hnd != 0);
}

inline bool is_nil() const { return hnd == 0; }

inline void operator=(ref const& r) {
    object_monitor::lock_global();
    if (r.hnd != hnd) {
        unlink();
        hnd = r.hnd;
        link();
    }
    object_monitor::unlock_global();
}

friend class_descriptor& classof(T const*);

inline void operator=(object_reference const& r) {
    object_monitor::lock_global();
    if (r.get_handle() != hnd) {
        unlink();
        hnd = r.get_handle();
        link();
        assert(hnd == 0 || CLASSOF(T).is_superclass_for(hnd));
    }
    object_monitor::unlock_global();
}

inline void operator=(T const* p) {
    object_monitor::lock_global();
    unlink();
    if (p == NULL) {
        hnd = 0;
    } else {
        hnd = ((object*)p)->get_handle();
        link();
    }
    object_monitor::unlock_global();
}
```

```
    ref(ref const& r) : object_reference(r.hnd) {
        object_monitor::unlock_global();
    }

    ref(object_reference const& r) : object_reference(r.get_handle()) {
        assert(hnd == 0 || CLASSOF(T).is_superclass_for(hnd));
        object_monitor::unlock_global();
    }

    ref(T const* p) {
        if (p != NULL) {
            object_monitor::lock_global();
            hnd = ((object*)p)->get_handle();
            link();
            object_monitor::unlock_global();
        }
    }

    ref() {}

    ~ref() { object_monitor::lock_global(); }
};


#endif
```

Приложение B

Стандартные классы GOODS

```
//-< DBSCLS.H >-----------------------------------------------------*--------*
// GOODS                     Version 1.0         (c) 1997  GARRET    *     ?  *
// (Garret Object Oriented Database System)                          *   /\|  *
//                                                                   *  /  \  *
//                          Created:     28-Mar-97    K.A. Knizhnik  * / [] \ *
//                          Last update: 17-Apr-97    K.A. Knizhnik  * GARRET *
//------------------------------------------------------------------*--------*
// Some common database classes
//------------------------------------------------------------------*--------*

#ifndef __DBSCLS_H__
#define __DBSCLS_H__

#include "goods.h"

//
// Dynamicly reallocatable array template. Array is automatically
// expanded when new element is stored at position beyond array bound.
// This template can't be used directly because it is impossible
// to register template classes in database.
// Instead of this concrete instantiations of this template can be used.
//
template<class T>
class array_template : public object {
  protected:
    nat4 n_elems;
    T    buf[1];

    array_template* expand(nat4 n) {
        nat4 allocated = (size - offsetof(array_template, buf)) / sizeof(T);
```

```
        if (n > allocated) {
            size_t new_size = n < allocated*2 ? allocated*2 : n;
            array_template* new_arr = new (new_size*sizeof(T)) array_template
                                          (0, cls, new_size*sizeof(T));
            memcpy(new_arr->buf, buf, n_elems*sizeof(T));
            new_arr->n_elems = n;
            new_arr->mop = mop;
            become(new_arr);
            return new_arr;
        }
        n_elems = n;
        return this;
    }
  public:
    static ref<array_template<T> > create(size_t init_size = 0) {
        return new (init_size*sizeof(T)) array_template(self_class, init_size);
    }

    const size_t length() const { return n_elems; }

    //
    // Array elements accessors
    //
    void putat(nat4 index, T elem) {
        if (index >= n_elems) {
            expand(index+1)->buf[index] = elem;
        } else {
            buf[index] = elem;
        }
    }
    T getat(nat4 index) const {
        assert(index < n_elems);
        return buf[index];
    }
    T operator[](nat4 index) const {
        assert(index < n_elems);
        return buf[index];
    }

    //
    // Stack protocol
    //
    void push(T elem) {
        int n = n_elems;
        expand(n+1)->buf[n] = elem;
```

```
  }
  T    pop() {
      assert(n_elems > 0);
      n_elems -= 1;
      return buf[n_elems];
  }
  T    top() const {
      assert(n_elems > 0);
      return buf[n_elems-1];
  }

  //
  // Massive operation
  //
  void  insert(nat4 pos, T elem) {
      nat4 n = n_elems;
      assert(pos <= n);
      array_template* arr = expand(n+1);
      memmove(&arr->buf[pos+1], &arr->buf[pos], (n - pos)*sizeof(T));
      memset(&arr->buf[pos], 0, sizeof(T));
      arr->buf[pos] = elem;
  }

  void  remove(nat4 pos) {
      assert(pos < n_elems);
      T zero;
      buf[pos] = zero;
      memcpy(&buf[pos], &buf[pos+1], n_elems - pos - 1);
  }

  void  copy(T* buf, int from, int len) const {
      assert(unsigned(from + len) <= n_elems && from >= 0 && len >= 0);
      int n = len;
      while (--n >= 0) {
          buf[n] = this->buf[n + from];
      }
  }

  METACLASS_DECLARATIONS(array_template, object);

protected:
  array_template(class_descriptor& desc, size_t init_size)
  : object(desc, init_size*sizeof(T))
  {
      n_elems = init_size;
```

```
    }
};

template<class T>
inline field_descriptor& array_template<T>::describe_components() {
    return FIELD(n_elems), VARYING(buf);
}

typedef array_template<char>          arrayOfChar;
typedef array_template<int4>          arrayOfInt;
typedef array_template<real8>         arrayOfReal;
typedef array_template< ref<object> > arrayOfObject;

class string : public arrayOfChar {
  public:
    int  compare(const char* str) const { return strcmp(buf, str); }
    int  compare(ref<string> const& str) const { return -str->compare(buf); }

    int  index(const char* str) const {
        char* p = strstr(buf, str);
        return p ? p - buf : -1;
    }

    static ref<string> create(size_t init_size = 0) {
        return new (init_size) string(self_class, init_size);
    }

    METACLASS_DECLARATIONS(string, arrayOfChar);

  protected:
    string(class_descriptor& desc, size_t init_size)
    : arrayOfChar(desc, init_size) {}
};


//
// Representation of one-to-many relationship: set of objects.
// Set conssist of set owner and a number of set members.
// Each member contains references to previouse and next member,
// reference to set owner, reference to associated object
// and full key of this object. First (last) member has next (prev) field
// equal to NULL. Member of a set has virtual function to compare
// two members. Also default function for converting complete key
// to short integer form
//
```

```
typedef nat4 skey_t;

class set_member : public object {
    friend class set_owner;
  public:
    ref<set_member> next;
    ref<set_member> prev;
    ref<set_owner>  owner;
    ref<object>     obj;

    char            key[1];

    static ref<set_member> create(ref<object> obj, const char* key) {
        size_t key_size = strlen(key) + 1;
        return new (key_size) set_member(obj, key, key_size);
    }
    static ref<set_member> create(ref<object> obj, const char* key,
                                  size_t key_size) {
        return new (key_size) set_member(obj, key, key_size);
    }

    ref<set_member> first() const;
    ref<set_member> last() const;

    inline  int     compare(ref<set_member> mbr) const {
        return -mbr->compare(key, size - offsetof(set_member, key));
    }
    virtual int     compare(const char* key, size_t size) const;
    virtual skey_t  get_key() const;

    static skey_t   str2key(const char* s, size_t len);

    METACLASS_DECLARATIONS(set_member, object);

  protected:

    set_member(ref<object> const& obj, const char* key, size_t key_size)
    : object(self_class, key_size)
    {
        next = NULL;
        prev = NULL;
        owner = NULL;
        this->obj = obj;
        memcpy(this->key, key, key_size);
```

```
    }
};


class set_owner : public object {
  public:
    ref<set_member> first;
    ref<set_member> last;
    ref<object>     obj;
    nat4            n_members;

    bool empty() const { return first == last; }
    ref<set_member> find(const char* key) const;

    void put_first(ref<set_member> mbr);
    void put_last(ref<set_member> mbr);
    void put_after(ref<set_member> after, ref<set_member> mbr);
    void put_before(ref<set_member> before, ref<set_member> mbr);

    ref<set_member> get_first();
    ref<set_member> get_last();
    void remove(ref<set_member> mbr);

    static ref<set_owner> create(ref<object> obj) {
        return new set_owner(self_class, obj);
    }

    METACLASS_DECLARATIONS(set_owner, object);

  protected:

    set_owner(class_descriptor& desc, ref<object> const& obj)
    : object(desc)
    {
        this->obj = obj;
        n_members = 0;
    }
};

//
// B tree - multibranch balanced tree, provides fast access for objects
// in database. B_tree class is derived from set_owner and use set members
// to reference objects.
//
class B_tree : public set_owner {
```

```
  public:
    ref<set_member> find(skey_t key) const;

    ref<set_member> find(const char* str, size_t len, skey_t key) const;
    ref<set_member> find(const char* str) const;

    void            insert(ref<set_member> mbr);
    void            remove(ref<set_member> mbr);

    static ref<B_tree> create(ref<object> obj) {
        return new B_tree(self_class, obj);
    }

    METACLASS_DECLARATIONS(B_tree, set_owner);

  protected:
    friend class B_page;

    ref<B_page> root;   // root page
    nat4        height; // height of tree

    B_tree(class_descriptor& desc, ref<object> const& obj)
    : set_owner(desc, obj)
    {
        root = NULL;
        height = 0;
    }
};

class B_page : public object {
    friend class B_tree;
  protected:
    enum { n = 1024 }; // Minimal number of used items at not root B tree page
    int4 m;            // Index of first used item

    struct item {
        ref<object> p; // for leaf page - pointer to 'set_member' object
                       // for all other pages - pointer to child page
        skey_t      key;

        field_descriptor& describe_components() {
            return FIELD(p), FIELD(key);
        }
        inline friend field_descriptor& describe_field(item& s) {
            return s.describe_components();
```

```
        }
    } e[n*2];
    //
    // Class invariants:
    //
    //   0 <= m <= 2*n-2, for root page
    //   0 <= m <= n, for other pages
    //
    // For not leaf page:
    // ( for all i : m < i < 2*n : e[i-1].key < e[i].key ) &&
    // ( for all i, j : m <= i < 2*n, e[i].p->m <= j < 2*n-1 :
    //   e[i].key > e[i].p->e[j].key &&
    //   e[i].key == e[i].p->e[2*n-1].key )
    //

    ref<set_member> find(int height, skey_t key) const;

    enum operation_effect {
        done,
        overflow,
        underflow,
        propagation
    };

    operation_effect insert(int r, item& ins);
    operation_effect remove(int r, item& rem);
    operation_effect handle_underflow(int r, skey_t key, item& rem);

    operation_effect find_insert_position(int level,
                                          ref<B_tree> const& tree,
                                          item& ins) const;
    operation_effect find_remove_position(int level, item& rem) const;

    B_page() : object(self_class) {}
    B_page(ref<B_page> const& root, item& ins); // create new B tree root

  public:
    METACLASS_DECLARATIONS(B_page, object);
};

#endif
```

```
//-< DBSCLS.CXX >---------------------------------------------------*--------*
// GOODS                     Version 1.0         (c) 1997  GARRET    *     ?  *
// (Garret Object Oriented Database System)                          *   /\|  *
//                                                                   *  /  \  *
//                          Created:     28-Mar-97    K.A. Knizhnik  * / [] \ *
//                          Last update: 10-Apr-97    K.A. Knizhnik  * GARRET *
//-------------------------------------------------------------------*--------*
// Some common database classes
//-------------------------------------------------------------------*--------*

#include "dbscls.h"

//
// Registration of array classes
//

REGISTER(arrayOfChar, object, pessimistic_scheme);
REGISTER(arrayOfInt, object, pessimistic_scheme);
REGISTER(arrayOfReal, object, pessimistic_scheme);
REGISTER(arrayOfObject, object, pessimistic_scheme);
REGISTER(string, arrayOfChar, pessimistic_scheme);

//
// Set methods
//

ref<set_member> set_member::first() const
{
    return owner->first;
}

ref<set_member> set_member::last() const
{
    return owner->last;
}

int set_member::compare(const char* key, size_t size) const
{
    size_t len = size < this->size - offsetof(set_member, key)
        ? size : this->size - offsetof(set_member, key);
    int diff = memcmp(this->key, key, len);
    return diff == 0 ? len - size : diff;
}

skey_t set_member::get_key() const
```

```
{
    return str2key(key, size - offsetof(set_member, key));
}

skey_t set_member::str2key(const char* s, size_t len)
{
    switch (len) {
      case 0:
        return 0;
      case 1:
        return *(nat1*)s << 24;
      case 2:
        return ((*(nat1*)s << 8) | nat1(s[1])) << 16;
      case 3:
        return ((((*(nat1*)s << 8) | nat1(s[1])) << 8) | nat1(s[2])) << 8;
      default:
        return ((((*(nat1*)s << 8) | nat1(s[1])) << 8)
                 | nat1(s[2])) << 8) | nat1(s[3]);
    }
}

field_descriptor& set_member::describe_components()
{
    return FIELD(next), FIELD(prev), FIELD(owner), FIELD(obj), VARYING(key);
}

ref<set_member> set_owner::find(const char* key) const
{
    size_t key_len = strlen(key);
    for (ref<set_member> mbr = first; mbr != NULL; mbr = mbr->next) {
        if (mbr->compare(key, key_len) == 0) {
            return mbr;
        }
    }
    return NULL;
}

void set_owner::put_first(ref<set_member> mbr)
{
    modify(mbr)->owner = this;
    modify(mbr)->next = first;
    modify(mbr)->prev = NULL;
    if (!first.is_nil()) {
        modify(first)->prev = mbr;
    } else {
```

```
        last = mbr;
    }
    first = mbr;
    n_members += 1;
}

void set_owner::put_last(ref<set_member> mbr)
{
    modify(mbr)->owner = this;
        modify(mbr)->next = NULL;
        modify(mbr)->prev = last;
    if (!last.is_nil()) {
        modify(last)->next = mbr;
    } else {
        first = mbr;
    }
    last = mbr;
    n_members += 1;
}

void set_owner::put_before(ref<set_member> before, ref<set_member> mbr)
{
    if (first == before) {
        modify(this)->put_first(mbr);
    } else {
        modify(mbr)->owner = this;
        modify(mbr)->next = before;
        modify(mbr)->prev = before->prev;
        modify(before)->prev = mbr;
        modify(mbr->prev)->next = mbr;
        n_members += 1;
    }
}


void set_owner::put_after(ref<set_member> after, ref<set_member> mbr)
{
    if (last == after) {
        modify(this)->put_last(mbr);
    } else {
        modify(mbr)->owner = this;
        modify(mbr)->next = after->next;
        modify(mbr)->prev = after;
        modify(after)->next = mbr;
        modify(mbr->next)->prev = mbr;
```

```
        n_members += 1;
    }
}

void set_owner::remove(ref<set_member> mbr)
{
    if (mbr == first) {
        (void)get_first();
    } else if (mbr == last) {
        (void)get_last();
    } else {
        assert(mbr->owner == this);
        modify(mbr->prev)->next = mbr->next;
        modify(mbr->next)->prev = mbr->prev;
        modify(mbr)->next = NULL;
        modify(mbr)->prev = NULL;
        modify(mbr)->owner = NULL;
        n_members -= 1;
    }
}

ref<set_member> set_owner::get_first()
{
    ref<set_member> mbr = first;
    if (!mbr.is_nil()) {
        first = first->next;
        if (first.is_nil()) {
            last = NULL;
        } else {
            modify(first)->prev = NULL;
        }
        modify(mbr)->next = NULL;
        modify(mbr)->prev = NULL;
        modify(mbr)->owner = NULL;
        n_members -= 1;
    }
    return mbr;
}

ref<set_member> set_owner::get_last()
{
    ref<set_member> mbr = last;
    if (!mbr.is_nil()) {
        last = last->prev;
        if (last.is_nil()) {
```

```
            first = NULL;
        } else {
            modify(last)->next = NULL;
        }
        modify(mbr)->next = NULL;
        modify(mbr)->prev = NULL;
        modify(mbr)->owner = NULL;
        n_members -= 1;
    }
    return mbr;
}


field_descriptor& set_owner::describe_components()
{
    return FIELD(first), FIELD(last), FIELD(obj), FIELD(n_members);
}

REGISTER(set_member, object, optimistic_scheme);
REGISTER(set_owner, object, pessimistic_repeatable_read_scheme);

//
// B tree
//

ref<set_member> B_tree::find(skey_t key) const
{
    if (root.is_nil()) {
        return NULL;
    } else {
        return root->find(height, key);
    }
}

ref<set_member> B_tree::find(const char* str, size_t len, skey_t key) const
{
    if (root != NULL) {
        ref<set_member> mbr = find(key);
        while (mbr != NULL) {
            int diff = mbr->compare(str, len);
            if (diff == 0) {
                return mbr;
            } else if (diff > 0) {
                return NULL;
            }
```

```
        }
    }
    return NULL;
}

ref<set_member> B_tree::find(const char* str) const
{
    size_t len = strlen(str);
    return find(str, len, set_member::str2key(str, len));
}

#define MAX_KEY 0xFFFFFFFF

void B_tree::insert(ref<set_member> mbr)
{
    B_page::item ins;
    ins.p = mbr;
    ins.key = mbr->get_key();

    if (root == NULL) {
        put_last(mbr);
        root = new B_page(NULL, ins);
        height = 1;
    } else if (root->find_insert_position(height, this, ins)
               == B_page::overflow)
    {
        root = new B_page(root, ins);
        height += 1;
    }
}

void B_tree::remove(ref<set_member> mbr)
{
    B_page::item rem;
    rem.key = mbr->get_key();
    rem.p = mbr;
    if (modify(root)->find_remove_position(height, rem) == B_page::underflow
        && root->m == items(root->e)-1)
    {
        root = root->e[items(root->e)-1].p;
        height -= 1;
    }
    set_owner::remove(mbr);
}
```

```
field_descriptor& B_tree::describe_components()
{
    return FIELD(root), FIELD(height);
}


B_page::B_page(ref<B_page> const& root, item& ins) : object(self_class)
{
    if (ins.key == MAX_KEY) {
        m = 2*n-1;
        e[2*n-1] = ins;
    } else {
        m = 2*n-2;
        e[2*n-2] = ins;
        e[2*n-1].key = MAX_KEY;
        e[2*n-1].p = root;
    }
}

ref<set_member> B_page::find(int height, skey_t key) const
{
    int l = m, r = 2*n-1;
    while (l < r)  {
        int i = (l+r) >> 1;
        if (key > e[i].key) l = i+1; else r = i;
    }
    internal_assert(r == l && e[r].key >= key);
    if (--height != 0) {
        ref<B_page> child = e[r].p;
        return child->find(height, key);
    } else {
        if (key == e[r].key) {
            return e[r].p;
        } else {
            return NULL;
        }
    }
}

B_page::operation_effect B_page::find_insert_position(int level,
                                                      ref<B_tree> const& tree,
                                                      item& ins) const
{
    int  i, l = m, r = 2*n-1;
    skey_t key = ins.key;
```

```
    while (l < r)  {
        i = (l+r) >> 1;
        if (key > e[i].key) l = i+1; else r = i;
    }
    internal_assert(r == l && e[r].key >= key);
    if (--level == 0)  {
        if (e[r].key == key) {
            ref<set_member> mbr;

            for (mbr = e[r].p;
                 mbr != NULL && mbr->compare(ins.p) < 0;
                 mbr = mbr->next);
            if (mbr == NULL) {
                modify(tree)->put_last(ins.p);
            } else {
                modify(tree)->put_before(mbr, ins.p);
                if (mbr == e[r].p) {
                    modify(this)->e[r].p = ins.p;
                }
            }
            return done;
        } else {
            if (e[r].p == NULL) {
                modify(tree)->put_last(ins.p);
            } else {
                modify(tree)->put_before(e[r].p, ins.p);
            }
        }
    } else {
        ref<B_page> child = e[r].p;
        if (child->find_insert_position(level, tree, ins) == done) {
            return done;
        }
    }
    // insert before e[r]
    return modify(this)->insert(r, ins);
}

B_page::operation_effect B_page::insert(int r, item& ins)
{
    // insert before e[r]
    if (m > 0) {
        memcpy(&e[m-1], &e[m], (r - m)*sizeof(item));
        memset(&e[r-1], 0, sizeof(item));
        m -= 1;
```

```
        e[r-1] = ins;
        return done;
    } else { // page is full then devide page
        B_page* b = new B_page;
        if (r < n) {
            memcpy(&modify(b)->e[n], e, r*sizeof(item));
            memset(&modify(b)->e[n+r], 0, sizeof(item));
            modify(b)->e[n+r] = ins;
            memcpy(&modify(b)->e[n+r+1], &e[r], (n-r-1)*sizeof(item));
        } else {
            memcpy(&modify(b)->e[n], e, n*sizeof(item));
            memset(&e[r-1], 0, sizeof(item));
            e[r-1] = ins;
            memcpy(&e[n-1], &e[n], (r-n)*sizeof(item));
        }
        ins.key = b->e[2*n-1].key;
        ins.p = b;
        m = n-1;
        modify(b)->m = n;
        return overflow;
    }
}

B_page::operation_effect B_page::find_remove_position(int level,
                                                      B_page::item& rem) const
{
    int  i, l = m, r = 2*n-1;
    nat4 key = rem.key;
    while (l < r) {
        i = (l+r) >> 1;
        if (key > e[i].key) l = i+1; else r = i;
    }
    internal_assert(r == l && e[r].key >= key);
    if (--level == 0) {
        assert(e[r].key == key);
        ref<set_member> mbr = rem.p;
        if (mbr == e[r].p) {
            if (mbr->next != NULL && mbr->next->get_key() == key) {
                modify(this)->e[r].p = mbr->next;
                return done;
            } else {
                return modify(this)->remove(r, rem);
            }
        }
    } else {
```

```
        ref<B_page> child = e[r].p;
        switch (child->find_remove_position(level, rem)) {
          case underflow:
            return modify(this)->handle_underflow(r, key, rem);
          case propagation:
            if (key == e[r].key) {
                modify(this)->e[r].key = rem.key;
                return propagation;
            }
          default:;
        }
    }
    return done;
}

B_page::operation_effect B_page::remove(int r, item& rem)
{
    if (e[r].key == MAX_KEY) {
        e[r].p = NULL;
        return done;
    } else {
        e[r].p = NULL;
        memmove(&e[m+1], &e[m], (r - m)*sizeof(item));
        rem.key = e[2*n-1].key;
        return (++m > n) ? underflow : propagation;
    }
}

B_page::operation_effect B_page::handle_underflow(int r, skey_t key, item& rem)
{
    ref<B_page> a = e[r].p;
    internal_assert(a->m == n+1);
    if (r < 2*n-1) { // exists greater page
        ref<B_page> b = e[r+1].p;
        int bm = b->m;
        if (bm < n) {  // reallocation of nodes between pages a and b
            int i = (n - bm + 1) >> 1;
            memcpy(&modify(a)->e[n+1-i], &a->e[n+1], (n-1)*sizeof(item) );
            memcpy(&modify(a)->e[2*n-i], &b->e[bm], i*sizeof(item) );
            modify(b)->m += i;
            modify(a)->m -= i;
            e[r].key = a->e[2*n-1].key;
            return done;
        } else { // merge page a to b
            internal_assert(bm == n);
```

```
            memcpy(&modify(b)->e[1], &a->e[n+1], (n-1)*sizeof(item));
            e[r].p = NULL;
            memmove(&e[m+1], &e[m], (r - m)*sizeof(item));
            modify(b)->m = 1;
            // dismiss page 'a'
            return ++m > n ? underflow : done;
        }
    } else { // page b is before a
        ref<B_page> b = e[r-1].p;
        int bm = b->m;
        if (key == e[r].key) {
            e[r].key = rem.key;
        }
        if (bm < n) { // reallocation
            int i = (n - bm + 1) >> 1;
            memcpy(&modify(a)->e[n+1-i], &b->e[2*n-i], i*sizeof(item));
            memmove(&modify(b)->e[bm+i], &b->e[bm], 2*n - bm - i );
            e[r-1].key = b->e[2*n-1].key;
            modify(b)->m += i;
            modify(a)->m -= i;
            return propagation;
        } else { // merge page b to a
            internal_assert(bm == n);
            memcpy(&modify(a)->e[1], &b->e[n], n*sizeof(item));
            e[r-1].p = NULL;
            memmove(&e[m+1], &e[m], (r-1-m)*sizeof(item));
            modify(b)->m = 1;
            // dismiss page 'b'
            return ++m > n ? underflow : propagation;
        }
    }
}

field_descriptor& B_page::describe_components()
{
    return FIELD(m), ARRAY(e);
}


REGISTER(B_tree, set_owner, pessimistic_repeatable_read_scheme);
REGISTER(B_page, object, optimistic_scheme);
```

Список литературы

[1] J. W. Schmidt, "Some High Level Language Constructs for Data of Type Relation", ACMP Trans. Database Systems, 2(3), Sept. 1977.

[2] J. M. Smidth, S. A. Fox and T. A. Landers, "ADAPLEX: Rationale and Reference Manual", Technical Report CCA-83-08, Computer Corporation of America, May 1983.

[3] P. O'Brien, B, Bullis, and C. Schaffert, "Persistent and Shared Objects in Trellis/Owl"

[4] A. Albano, G. Ghelli, M. E. Occhiuto, and R. Orsini "A Strongly Typed, Interactive Object-Oriented Database Programming Language"

[5] H.-J. Schek and M. H. Scholl, "The Relational Model with Relation-Value Attributes" Information Systems, 11,4, 1986.

[6] P. Fischer and S. Thomas, "Operators for Non-Foirst-Normal-Form Relations", Proc. COMPSAC83, November, 1983

[7] F. Bancilhon and S. Khoshafian, "A Calculus of Complex Objects", Proc. ACM SIGACT-SIGMOD Sypm. on Principles of Database Systems, 1986.

[8] E. F. Codd, "Extending the database Relational Model to Capture More Meaning", ACM Trans. database Systems, 4, No.4 (December 1979).

[9] P. Checn, "The Entity-Relationship Model – Toward a Unified View of Data", ACMP Trans. database Systems, 1, No.1 (March 1976).

[10] J. Mylopoulos, P. Bernstein, and H.K.T. Wong, "A Language Facility for Designing Database-Intensive Applications", ACM Trans. Database Systems, 5, 2 (June 1980).

[11] M. Hammer and D. McLeod, "The Semantic Data Model", Proc. ATM-SIGMOD Intl. Conf. on Management of Data, 1978.

[12] D. Shipman, "The Functional Data Model and the Data Language DAPLEX"

[13] Adele Goldberg and D. Robson, *Smalltalk-80: The language and Its Implementation*, Addison-Wesley, 1983.

[14] O. J. Dahl and K. Nygaard, "SIMULA – An Algol-based Simulation Language", Comm. ACM 9, 671-678.

[15] Elisa Bertino and Lorenzo Martino, /it Object-Oriented Database Systems: Concepts and Arcitectures, Addison-Wesley, Reading, MA, 1993.

[16] Alan W. Brown, *Object-Oriented Databases: Applications in Software Engineering*, McGraw-Hill, London, 1991.

[17] Malcolm P. Atkinson, Francois Bancilhon, David DeWitt, Klaus Dittrich, David Maier, Stanley Zdonik, "The Object-Oriented Database System Manifesto"

[18] Davied Maier and Stanley Zdonik, "Fundamentals of Object-Oriented Databases",

[19] The Committee for Advanced DBMS Function, "Third-Generation Data Base System Manifesto", Memorandum UCB/ERL M90/28, University of California, Berkley, 9 April 1990.

[20] Shamim Ahmed, "A Comparison of Object-Oriented Database management Systems for Engineering Applications", Research Report R91-12, Inteligent Engineering Systems Laboratory, Mass. Inst. of Technology, May 1991.

[21] Robert Bretl, David maier, Allan Otis, Jason Penney, Bruce Schuchardt, Jacob Stein, E. Harold Williams, Monty Williams, "The GemStone Data Management System".

[22] Paul Butterwoworth, Allem Otis, jacob Stein, "The GemStone Object database Management System", Comm. ACM 34(10), Oct. 1991, 64-77.

[23] Usenet newsgroup comp.object "Frequently Asked Questions (FAQ) List and Availbale Systems for Object-Oriented Technology"

[24] Mike Ricciuti, "The Easy Way to OOPS", Datamation, February 1, 1992.

[25] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb, "The ObjectStore Database System", Comm. ACM 34(10), 50-63 (October 1991).

[26] Object Design, Ins., `ObjectStore Tutorial`, Release 2.0, October 1992.

[27] Object Design, Ins., `ObjectStore User Guide`, Release 2.0, October 1992.

[28] Object Design, Ins., `ObjectStore Technical Overview`, Release 3.0, March 1994.

[29] P. A. Bernstein, V. Hadzilacos, N. Goodman, *Concurrency Control and Recovery in Database Syastens*, Addison-Wesley, 1987.

[30] C. Papadimitriou, *The Theory of Database Concurrency Control*, Computer Science Press. 1986.

[31] O. O'Neil, *DatabaseL Principles, Programming, Performance*, Morgan Kaufmann, 1994, Section 9.5.

[32] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Corrected Second Printing, Morgan Kaufmann, 1993. Section 7.6 and following.

[33] G. Kiczales, J. des Rivieres, and D. G. Bobrow, *The Art of te Metaobject Protocol*, MIT Press, Cambrifge, MA, 1991.

[34] G. Kiczales, "Aspect-Oriented Programming", 1997: Xerox PARC.

[35] Дейт К. *Введение в системы баз данных*, М.: Наука, 1980 .

[36] Замулин А. В. *Системы программирования баз данных и знаний*, М.: Наука, 1990 .

[37] Замулин А. В. *Типы данных в языках программирования и базах данных*, Новосибирск: Наука, Сиб. отд-ние, 1987

[38] Мартин Дж. *Организация баз данных в вычислительных системах*, М.: Наука, 1980.

[39] Тыугу Э. Х., Мацкин М. Б., Пеньям Я. Э. и др. “Объектно-ориентированный язык НУТ”, Прикладная информатика, М.: Финансы и статистика, 1983.