СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Системы управления базами данных (СУБД) играют исключительную роль в организации современных промышленных, инструментальных и исследовательских информационных систем, а также в САПР. В реферате описываются наиболее интересные направления исследований и разработок проводимых в настоящее время.

1. Особенности СУБД, применяемых в САПР

Управление данными - одна из существенных частей систем автоматизации проектирования. В большинстве пакетов САПР управление данными реализуется в каждой системе по-своему, без какой-либо стандартизации. Но управление данными - не главное в САПР. В этих системах приходится решать массу специфических проблем и понятно желание разработчиков САПР пользоваться какими-нибудь стандартными средствами управления данными, лишь бы они удовлетворяли потребностям САПР: требуются эффективный доступ, возможности манипулирования объектами сложной структуры, возможности работы с различными версиями объектов, возможности откатов к предыдущим версиям и т.д.

Появление достаточно эффективных реляционных систем управления базами данных естественно обратило на них внимание разработчиков САПР, которые стали пытаться использовать эти системы как стандартные средства управления данными. Но тогда выяснилось, что возможности систем баз данных, вполне достаточные в традиционных приложениях, не полностью удовлетворяют потребности САПР. К таким особенностям реляционных систем управления базами данных можно отнести следующие:

1. Существует возможность описывать сложные объекты за счет включения в отношения атрибутов соединений. Более того, можно извлечь из базы данных сложный объект (или его представление) за один запрос, но этот запрос при традиционной организации базы данных потребует выполнения нескольких операций соединения и не сможет быть выполнен с требуемой для САПР эффективностью. Кроме того, плоская форма сложного объекта, которую можно получить за счет реляционного запроса, не очень-то удобна для последующей обработки в оперативной памяти.

2. Существует возможность взаимодействовать с базой данных в пределах транзакции, которая является атомарной единицей изменения базы данных и результаты работы которой либо полностью отображаются в состоянии базы данных, либо полностью отсутствуют. Система баз данных гарантирует нормальный конец транзакции только в том случае, если транзакция не нарушила логическую целостность базы данных, причем критерии логической целостности достаточно жестки. Такой механизм транзакций и соответствующей синхронизации непригоден для САПР, критерии логической целостности баз данных которых изменчивы, и даже в соответствии с этими критериями базы данных могут долго находиться в нецелостных состояниях.

3. Система управления базами данных поддерживает информацию о предшествующих состояниях баз данных в форме журналов изменений. Журнал используются системой при восстановлениях баз данных после сбоев процессора или поломок внешних носителей, но пользователи не имеют возможности явного доступа к накопленной в журнале информациии. САПР нуждаются в таких возможностях. Более точно, САПР требуются средства доступа к предыдущим состояниям объектов базы данных.

Резюмируя отмеченные несогласованности между возможностями реляционных систем управления базами данных и потребностями САПР, подчеркнем, что использованию реляционных систем в САПР мешают

нормализованность отношений, ограниченность понятия транзакций и отсутствие средств явной работы с предыдущими состояниями объектов. Вместе с тем, все традиционные возможности систем баз данных тоже полезны в САПР, и, следовательно, желательно появление систем управления базами данных, обладающих некоторыми новыми возможностями, но не утративших предыдущие. Работы над развитием традиционных СУБД в этом направлении интенсивно ведутся. В этом реферате мы рассмотрим развитие System R для обеспечения потребностей САПР и других нетрадиционных приложений.

2. Проблемы, связанные с применением СУБД в САПР

Вопрос о перспективах моделирования в системах баз данных, конечно, отнюдь не исчерпывается затронутыми проблемами. Серьезное обсуждение этой темы требует специального обсуждения, которое выходит за рамки данного реферата . Ограничимся здесь лишь несколькими примерами злободневных проблем. Например проблема падения напряжения на кеше 2-го уровня процессоров R3000, в результате которой данные из кеша второго уровня записываются на аварийный хранитель ниформации (например перфокарты), что может привести к утере значений определенных полей таблицы.

Одной из них является обеспечение интеграции неоднородных информационных ресурсов, в частности, структурированных и слабоструктурированных данных. Необходимость ее решения связана, в частности, со стремлением к полноценной интеграции систем баз данных в среду САПР-технологий. При этом уже недостаточно простого обеспечения доступа к базе данных традиционным теперь способом «из-под» САПР-форм. Нужна интеграция на модельном уровне. И не просто синтаксическая интеграция. Например, разработки цифровых библиотек (Digital Library) требуют обеспечения семантической интероперабельности информационных ресурсов.

Другая модельная проблема, которая на самом деле является подпроблемой предыдущей, состоит в разработке средств и технологий, предусматривающих явную спецификацию метаданных для ресурсов слабоструктурированных данных, которыми столь богата среда САПР, в поисках путей перенесения традиционных технологий моделирования из области баз данных в эту среду. (Напомним, что в области систем баз данных аналогичная ситуация была преодолена благодаря предложениям CODASYL о необходимости явной спецификации схемы базы данных, независимой от приложений, см. выше п.2). Именно на достижение этой цели направлены интенсивные разработки консорциумом W3C нового языка XML и его инфраструктуры (фактически, новой модели данных для этой среды), объектной модели документов и других компонентов нового комплекса средств, который, как можно ожидать, в близкое время станет основой новых технологий управления информационными ресурсами в СУБД.

Важнейшей проблемой организации крупных распределенных неоднородных информационных систем является разработка методов построения репозиториев метаданных, обладающих развитой функциональностью, в частности, обеспечивающих повторное использование ресурсов благодаря способности выявления ресурсов, удовлетворяющих заданным спецификациям, семантического отождествления ресурсов и т.д.

Можно упомянуть, наконец, «более прозаические» модельные проблемы, связанные с интеграцией систем баз данных и систем управления документами, решение которой в значительной мере продвинуто в настоящее время, но она все-таки еще далеко не исчерпана.

3. System R для обеспечения потребностей САПР

В этом разделе мы рассмотрим развитие System R для обеспечения потребностей САПР. Как мы неоднократно отмечали, System R получила развитие в нескольких направлениях. На базе этой системы возникли два коммерческих продукта фирмы IBM - SQL/DS и DB2, реляционные СУБД, оснащенные всеми необходимыми сервисными средствами. На протяжении ряда лет велись работы над распределенной СУБД System R* . Отдельным направлением является разработка прототипа СУБД для инженерных приложений XSQL, и именно на этой системе мы остановимся.

XSQL интересна своим комплексным подходом к решению проблем. Она предоставляет возможности работы со сложными объектами, в ней расширяется традиционное понятие транзакции и т.д. При этом разработчики стремятся в наибольшей степени использовать базовый подход System R, меняя в нем как можно меньше. В частности, базовым языком запросов XSQL остается известный язык SQL, в который добавлены конструкции для определения и манипулирования сложными объектами.

Мы рассмотрим следующие аспекты системы XSQL: подход к организации и управлению сложными объектами; управление транзакциями; управление версиями.

3.1. Организация сложных объектов в XSQL

Под сложным объектом базы данных, следуя идеологии разработчиков XSQL, мы понимаем совокупность кортежей одного или нескольких отношений базы данных, совместно содержащих информацию об одном реальном объекте предметной области.

В начале работ по развитию System R в направлении инженерных приложений в них активно участвовал известный специалист в области реляционных систем баз данных В.Ким (который в дальнейшем перешел в MCC и в настоящее время, видимо, отошел от этих работ). Мы подчеркиваем этот факт по той причине, что, пожалуй, лучшей публикацией о сложных объектах в System R является , список авторов которой он возглавляет.

Напомним, что при организации памяти для хранения баз данных в System R был выбран подход, при котором эффективность доступа определялась в основном кластеризованностью одного или нескольких отношений по одному или нескольким полям. Под кластеризованностью понимается физическое размещение кортежей одного или нескольких отношений, обладающих одинаковым значением выделенных атрибутов, в одной или смежных физических страницах внешней памяти.

Кластеризация одного отношения позволяет наиболее эффективно осуществлять выборки в соответствии с указаниями значений полей кластеризации. Совместная кластеризация нескольких отношений (допустимая, но не используемая в последних версиях System R) обеспечивает потенциальную возможность эффективного выполнения соединения этих отношений по полям кластеризации. Эффективность выполнения операций

может быть обеспечена за счет существования индекса на полях кластеризации (при совместной кластеризации можно было бы использовать аппарат связей или мультииндексов).

В принципе, такая организация уже дает возможность эффективно управлять сложными объектами, поскольку необходимые соединения нескольких отношений становятся ненакладными. Но при этом семантика сложного объекта остается неизвестной системе. В частности, необходимо отдельно от описания структуры объекта указывать необходимые условия целостности, и невозможно автоматически проверить их достаточность. Второй деффект такого подхода мы уже отмечали - избыточность и неудобство работы на прикладном уровне с плоскими представлениями выбранных в оперативную память объектов.

Напрашивается решение ввести структуру сложного объекта на уровне хранения. Физические возможности для организации ссылок между кортежами обеспечиваются уровнем управления памятью System R - каждый кортеж получает на время своего существования уникальный идентификатор (tid), по которому возможен прямой доступ к кортежу. Но такое решение повлекло бы два следствия. Во-первых, нужно было бы переделывать подсистему управления памятью, перегружая ее информацией, по сути дела, логического уровня. Во-вторых, наличие между кортежами физических ссылок вызвало бы усложнение операции копирования сложного объекта как единого целого.

Подход, выбранный в XSQL, решает проблемы управления сложными объектами на логическом уровне СУБД с использованием средств кластеризации отношений, предоставляемых физическим уровнем (сам физический уровень, т.е. RSS, при этом не меняется по сравнению с System R).

Все отношения, кортежи которых входят в сложный объект, расширяются одним полем - суррогатом или логическим идентификатором кортежа. Ссылки между кортежами сложного объекта - это тоже явно указанные поля отношений на физическом уровне, и значения ссылок есть суррогаты кортежей. Структура сложного объекта (а тем самым и правила кластеризации составляющих его отношений и условия целостности сложного объекта) описывается на расширенном SQL. Конструкции расширенного SQL, предназначенные для определения структуры сложного объекта, довольно громоздки, и формальное описание синтаксиса языка в публикациях не приводится.

Отношение, кортежи которого являются корнями иерархий сложных объектов, расширяется еще одним большим по объему полем, которое содержит таблицу приписки суррогатов кортежей, составляющих данный сложный объект, к их физическим идентификаторам. Таким образом, для того, чтобы полностью выбрать сложный объект (или его указанный подобъект) достаточно найти каким-либо образом его корневой кортеж и пройти по ссылкам. Кластеризация кортежей сложного объекта делает эту операцию эффективной. Заметим по этому поводу, что хотя при реализации прототипа XSQL использовался тот же вариант RSS, что и в System R (точнее, в SQL/DS), в этой реализации используются возможности RSS по части совместной кластеризации кортежей нескольких отношений (в описанном в Разделе 2 интерфейсе RSS мы не упоминали эту возможность, поскольку она не использовалась в System R).

Возможны два пути указания подобъектов сложного объекта. Если наиболее важным фактором является эффективность выборки, то общие подобъекты нескольких сложных объектов физически копируются. Это не создает проблем с их физической согласованностью, потому что на уровне SQL модификация отношений происходит в соответствии с условием модификации и потому просто невозможно выполнить модификацию только одной копии кортежа (различающие их суррогаты не видны на уровне SQL). Но, конечно, при такой организации операция модификации может стать дорогостоящей.

Второй допустимый способ - физическое разделение общего подобъекта несколькими объектами. В этом случае ссылка становится несколько более сложной, но сохраняется логической. При таком способе организации, вообще говоря, теряется эффективность выборки, но модификации дешевы.

Наличие логических ссылок и таблицы приписки позволяет после выборки сложного объекта в основную память образовать обычную списковую структуру, заменив логические ссылки в кортежах на указатели по оперативной памяти. Такая структура сложного объекта в оперативной памяти очень удобна для последующей обработки на прикладном уровне. Наличие таблицы приписки, устанавливающей соответствие логических идентификаторов кортежей с их tid'ами, позволяет легко реализовать операцию копирования сложного объекта. При этом нужно модифицировать только одно поле (содержащее таблицу приписки) корневого кортежа объекта.

Появление новой управляющей структуры - таблицы приписки сложного объекта добавляет возможности при выборе способа пути доступа в оптимизаторе СУБД. Как подчеркивается в [61], бывает полезно использовать таблицу приписки (своего рода индекс) не только при выборке сложных объектов.

Диалект SQL, используемый в XSQL, содержит несколько расширений, относящихся в средствам выборки. Все они, естественно, отражают специфику работы со сложными объектами и составляющими их кортежами. Вообще говоря, язык в результате в некоторой степени утратил свою простоту, но зато достаточно сложные запросы можно формулировать лаконично. Соответствующим образом расширены и операторы занесения, удаления и модификации кортежей, входящих в сложные объекты.

3.2. Расширение модели транзакции в XSQL

Как мы отмечали выше, приложения САПР нуждаются, кроме средств организации сложных объектов, в расширении и пересмотре понятия транзакции. В.Ким с разными соавторами предлагал ввести целую иерархию понятий транзакций со своими правилами согласованности и протоколами на каждом уровне, предлагалась очень общая модель версий объектов в базах данных САПР. Вообще, последние работы этого автора отличаются слишком большим уровнем общности. Все эти работы содержат интересные предложения, но они настолько далеки от реализации (а гипотетическая реализация их настолько сложна), что практически работы становятся неинтересны. Видимо в этом и разошелся В.Ким с разработчиками XSQL, которые, начиная еще с System R, всегда отличались стремлением к простоте.

Для точности отметим, что В.Ким возвращается к практическим вопросам реализации систем управления базами данных для САПР. В этой статье утверждается, что основным отличием проводимой им (и другими разработчиками MCC) работы от XSQL является то, что разработчики из Сан-Хосе черезчур заботятся о сохранении неизменной подсистемы управления памятью RSS (на наш взгляд это вполне естественно: прежде, чем менять хорошо зарекомендовавший себя продукт, нужно по крайней мере убедиться в том, что его средства недостаточны).

По этому поводу интересно рассмотреть решение проблемы с длинными "инженерными" транзакциями в XSQL. Управление транзакциями основано на одновременой работе с общей базой данных и набором частных баз данных (последние могут, но не обязаны, располагаться на рабочих станциях). Перед доступом к объекту он должен быть явно перемещен в частную базу данных, и об этом производится запись в специальном управляющем отношении, хранящемся в общей базе данных (делается долговременный захват).

Работа с общей базой данных происходит в терминах традиционных "коротких" транзакций, но конец такой транзакции не приводит к автоматическому снятию долговременных захватов, появившихся при работе этой транзакции. Для обновления объекта в общей базе данных требуется наличие соответствующего долговременного захвата (что в общем случае может потребовать переговоров пректировщиков, одновременно работающих с данным объектом). Долговременные захваты снимаются либо по явной команде, либо при сбросе объекта в общую базу данных.

Заметим, что XSQL обеспечивает работу лишь с общей базой данных. Программное обеспечение частных баз данных может быть очень различно (например, тоже с использованием XSQL или на основе прикладных систем). Главное - это соблюдение правил работы с общей базой данных, а это XSQL гарантирует.

Имеется четкое разделение функций между логическим уровнем XSQL и RSS. Как и в System R, обеспечение синхронизации операций параллельно выполняемых коротких транзакций относится к числу функций RSS. Наличие длинных транзакций вообще неизвестно RSS. Управление ими (впрочем, как видно из изложенного выше, очень простое) осуществляется на логическом уровне RSS.

Трудно сказать, является ли достаточным для инженерных разработок механизм управления транзакциями, реализованный в XSQL. Видимо, об этом можно судить только на основе опыта. В доступных публикациях такой анализ пока не проводился. Но по крайней мере одно преимущество реализации механизма длинных транзакций в XSQL очевидно - это простота и ненакладность (о чем, кстати, можно судить, исходя из небольшого объема данного подраздела, в котором содержится вся существенная информация).

3.3. Управление версиями объектов в XSQL

Как мы отмечали в начале этого раздела, одним из требований САПР к системам управления базами данных является требование обеспечения доступа к данным, характеризующим предыдущее состояние проекта.

XSQL обеспечивает возможность образования версий объектов в общей базе данных. В соответствии с общим стремлением разработчиков к простоте и при решении этой проблемы они остановились на некотором относительно простом и не требующем переделок физического уровня системы варианте.

Рассмотрим общую модель управления версиями в XSQL. Прежде всего отметим, что в первой реализации механизма версий разработчики XSQL не стремились к экономии внешней памяти. Отмечается, что в дальнейшем, возможно, будет реализован более экономный механизм. Управление версиями производится на основе базового понятия сложного объекта. Для каждого объекта может существовать множество версий, составляющих *объект проектирования*. С объектом проектирования (ОП) связывается идентификатор.

Каждая версия идентифицируется номером версии внутри объекта проектирования, т.е. пара <идентификатор ОП, номер версии> уникально идентифицирует версию. Номера версий генерируются автоматически при создании версий. Внутри ОП номера версиям назначаются в порядке возрастания и никогда не используются повторно. Это отражает семантику версий: версия с большим номером является более новой; отсутствие версии с указанным номером (меньшим максимального) означает, что эта версия была уничтожена.

Одна и только одна версия ОП может быть помечена с помощью ключевого слова CURRENT (текущая). После связывания CURRENT с некоторой версией это ключевое слово является синонимом номера этой версии. Следовательно, пара <идентификатор ОП, CURRENT> однозначно идентифицирует версию.

Версию можно объявить *замороженной* (frozen). В общем случае в процессе проектирования создается несколько версий, пока не будет достигнут некоторый согласованный уровень, который можно отдать на использование другим проектировщикам. В этот момент версия замораживается, после чего не допускаются ее удаление и модификации до тех пор, пока ее явно не разморозит пользователь. Средства замораживания и размораживания версий доступны пользователям, и ими следует пользоваться осторожно. Замороженная версия ОП с максимальным номером идентифицируется парой <идентификатор ОП, LAST FROZEN> (последняя замороженная версия).

Диалект SQL, используемый в XSQL, включает следующие средства управления объектами проектирования и версиями:

- операторы создания и уничтожения объектов проектирования;

- операторы создания и уничтожения версий (допускается уничтожение указанной версии; текущей версии; всех версий, более "молодых", чем последняя замороженная; всех незамороженных версий; версий с номерами из указанного интервала);

- операторы замораживания и размораживания версий;

- операторы выборки объектов проектирования и их версий;

- операторы выборки мета-информации по поводу объектов проектирования и их версий (например, число существующих версий).

Следующий существенный аспект управления версиями связан с тем, что в одном объекте проектирования могут находиться ссылки на другие объекты проектирования. Пусть, например, существуют два объекта проектирования ОП1 и ОП2, и имеются ссылки из ОП1 на ОП2. Оказывается неудобным хранить в версиях ОП1 ссылки на версии ОП2. Это влечет накладные расходы и не позволяет разным пользователям ОП1 использовать в одной версии ОП1 разные версии ОП2.

Для решения этой проблемы предлагается техника, основанная на понятиях *родовых ссылок* и *среды*. Родовая ссылка - это ссылка на объект проектирования, в которой не уточняется, какая версия будет использвана.

Среда - это именованное бинарное отношение, состоящее из пар <идентификатор ОП, номер версии>. Каждая пара специфицирует номер версии, которая должна быть использована при разрешении родовой ссылки на соответствующий объект проектирования. Если для некоторого объекта проектирования среда не содержит такой пары, при разрешении родовой ссылки используются правила умолчания, которые также специфицируются (можно указать, что по умолчанию выбирается текущая версия или последняя замороженная версия объекта проектирования). Среда может быть создана в любой момент и хранится в базе данных. Однако, реальное уточнение родовых ссылок происходит только на основе *активной* среды. Активизация среды происходит по явной команде с указанием имени среды. В каждый момент времени только одна среда может быть активной, и активизация среды приводит к деактивизации предыдущей.

Различаются два представления среды - начальное, определяемое пользователем, и рабочее, вырабатываемое в ходе активизации среды или заранее. Начальное определение среды состоит из трех разделов. Первый раздел включает прямо указываемые соответствия объектов проектирования и их версий в данной среде. Номер версии может указываться явно, либо с употреблением идентификаторов LF (последняя замороженная версия) или C (текущая версия). Второй раздел содержит пары вида <идентификатор ОП, имя среды>. При этом понимается, что соответствие должно быть установлено на основе информации из именованной среды. Наконец, третий раздел содержит список включения ранее определенных сред с указанием их приоритетов, т.е. указывает на необходимость использования соответствий (если они не определены явно в первых двух разделах) из именованных сред в соответствии с их приоритетами.

Имеется средство преобразования начальной среды в рабочую, которая содержит только список пар, задающих явные соотствия объектов проектирования и их версий. При активизации среды можно использовать начальное представление (и тогда будет каждый раз происходить преобразование начальной среды в рабочую) или сформированную раньше и сохраненную в базе данных рабочую среду (тогда преобразования при активизации не потребуются).

При работе с версиями объекта проектирования бывает полезно уметь задавать структуру версий (например, некоторое подмножество версий составляет одно издание, несколько изданий составляют одну альтернативу

объекта проектирования и, наконец, объект проектирования состоит из набора альтернатив). Реализован общий механизм структуризации версий (необязательно иерархической). Введено понятие логического кластера версий или кластеров. Версия или кластер могут входить в несколько кластеров.

Существуют операции, позволяющие задать имена типов кластеров, участвующих в структуризации объекта проектирования, и их связи по вложенности; операции порождения и уничтожения экземпляров кластеров; операции выборки кластеров и версий. Имена вырожденных кластеров "объект проектирования" и "версия" предопределены. При определении типов кластеров контролируется доступность любого промежуточного кластера из корня - объекта проектирования и листа - версии из любого промежуточного кластера.

Использование кластеров допускается также при определении среды: пару <идентификатор ОП, номер версии> можно расширить до триплета <идентификатор ОП, идентификатор кластера, номер версии>. При этом номер версии может быть задан явно или с помощью идентификаторов LF и C, но понимается как номер версии в указанном кластере, а не в объекте проектирования в целом.

Отмечается, что механизм кластеров необязателен для использования, и если его не использовать, т.е. работать только на уровне версий объекта проектирования, то существование этого механизма не приводит к дополнительным накладным расходам.

Работа с версиями в текущей версии реализована над базовым уровнем XSQL. Как отмечается в, это позволило проверить правильность идей, после чего, если этого потребуют соображения эффективности, можно перенести некоторые части схемы ниже интерфейса XSQL. Из последнего замечания следует, что на момент публикации достаточного опыта XSQL с механизмом версии еще не было.

Завершая этот подраздел, заметим, что разработчиками XSQL предложен достаточно сложный механизм управления версиями (но все-таки гораздо более простой, чем описанный ранее). Мы не исключаем, что спустя некоторое время появятся публикации, описывающие более простую, эффективно реализуемую схему управления версиями в этой же системы (последователи System R неоднократно проходили подобный путь в своих разработках).

3.4. Синхронизация в System R

System R с самого начала задумывалась как многопользовалельская система, обеспечивающая режим мультидоступа к базам данных. Поэтому вопросам синхронизации доступа всегда уделялось очень большое внимание. Разработчики System R сформулировали и частично решили много проблем синхронизации, соответствующие публикации давно стали классикой, и на них ссылаются практически во всех работах, связанных с синхронизацией в системах управления базами данных. Тем не менее, мы считаем полезным рассмотреть этот вопрос еще раз и попытаемся привести историческую ретроспективу решений в области синхронизации в System R. Предварительно заметим, что вопросы синхронизации находятся в тесной связи с вопросами журнализации изменений и восстановления состояния базы данных. Поэтому этот и следующий подразделы также связаны, и нам снова придется временами

забегать вперед.

Начнем с рассмотрения целей, которыми руководствовались разработчики System R при выработке своего подхода к синхронизации. Дело в том, что начальной целью синхронизации операций было не обеспечение изолированности пользователей, а поддержка средств обеспечения логической целостности баз данных. Как мы отмечали во введении, логическая целостность баз данных System R поддерживается на основе наличия ранее сформулированных и запомненных в каталогах базы данных ограничений целостности. В конце каждой транзакции или при выполнении явного предложения SQL проверяется ненарушение ограничений целостности изменениями, произведенными в данной транзакции. Если обнаруживается нарушение ограничений целостности, с помощью операции RSS RESTORE производится откат транзакции, нарушившей ограничения.

Для того, чтобы можно было корректно выполнить такой откат, необходимо, чтобы до конца транзакции объекты базы данных, изменявшиеся транзакцией, не могли изменяться и другими транзакциями. В противном случае возникает так называемая проблема *потерянных изменений*. Действительно, пусть транзакция 1 изменяет некоторый объект базы данных A. Затем другая транзакция 2 также изменяет объект A, после чего производится откат транзакции 1 (по причине, например, нарушения ей ограничений целостности). Тогда при следующем чтении объекта A транзакция 2 увидет его состояние, отличное от того, в которое он перешел в результате его изменения транзакцией 2.

Исходным постулатом System R было то, что потерянные изменения допускать нельзя, и обеспечение этого являлось минимальным требованием к системе синхронизации. Соответствующий режим выполнения транзакций называется в System R первым уровнем совместимости транзакций. Это наиболее низкий уровень синхронизации, вызывающий минимальные накладные расходы в системе. Технически синхронизация на первом уровне совместимости предполагает долговременные (до конца транзакции) синхронизационные захваты изменяемых объектов базы данных и отсутствие каких-либо захватов для читаемых объектов.

С точки зрения обеспечения логической целостности баз данных первый уровень совместимости вполне удовлетворителен, но вызывает некоторые проблемы внутри транзакций. Основная проблема - возможность чтения *грязных данных*. Действительно, читающая транзакция абсолютно не синхронизуется с изменяющими транзакциями, и поэтому в ней может быть прочитано некоторое промежуточное с логической точки зрения состояние объекта (мы подчеркиваем, что "грязным" объект может быть только на логическом уровне; физическую согласованность в базе данных поддерживает другой, "физический" уровень синхронизации RSS, на котором мы остановимся позже). Следующий, второй уровень совместимости транзакций System R обеспечивает отсутствие грязных данных. Технически синхронизация на втором уровне совместимости предполагает долговременные синхронизационные захваты (до конца транзакции) изменяемых объектов и кратковременные (на время выполнения операции) захваты читаемых объектов.

Второй уровень совместимости транзакций System R гарантирует отсутствие грязных данных, но не свободен от еще одной проблемы -

неповторяющихся чтений. Если транзакция два раза (подряд или с некоторым временным промежутком) читает один и тот же объект базы данных, то состояние этого объекта может быть разным при разных чтениях, поскольку в промежутке между ними (а он может возникнуть, даже если чтения идут в транзакции подряд) другая транзакция может модифицировать объект. Эту проблему решает третий уровень совместимости транзакций System R, являющийся тем самым уровнем максимальной изолированности пользователей, о котором мы говорили раньше. Технически синхронизация на третьем уровне совместимости предполагает долговременные (до конца транзакции) синхронизационные захваты всех объектов, читаемых и изменяемых в данной транзакции.

В начальных версиях System R обеспечивались на самом деле все перечисленные уровни совместимости транзакций. Соответственно, существовали параметры операции RSS BEGIN TRANSACTION, определявшие уровень совместимости данной транзакции. Однако, уже первый опыт использования прототипа System R показал, что наиболее применительным является третий уровень совместимости. При этом существовали приложения, которые устраивал первый уровень (в основном, приложения, связанные со статистической обработкой, в которых ошибки "грязных" данных исправлялись за счет большого числа чтений). Второй уровень совместимости оказался практически неприменимым. В результате в последних версиях разработчики оставили только третий уровень совместимости, и далее мы будем иметь в виду только его.

Поскольку интерфейс RSS - покортежный, то логично производить синхронизацию в RSS именно на уровне кортежей или, вернее, их уникальных идентификаторов - tid'ов. Заметим, однако, что если две или более транзакций читают один кортеж, то с точки зрения синхронизации это вполне допустимо, а если хотя бы одна транзакция изменяет кортеж, то она должна блокировать все остальные транзакции, выполняющие операции чтения или изменения данного кортежа. Из этого следует потребность в двух разных режимах захватов кортежей - совместном режиме (для чтения) и монопольном (для изменений). Следуя терминологии System R, далее мы будем называть эти режимы режимом S (совместным) и режимом X (монопольным). Естественно формулируются *правила совместимости* захватов одного объекта в режимах S и X: захват режима S совместим с захватом режима S и несовместим с захватом режима X; захват режима X несовместим с захватом любого режима.

Соответственно, при чтении кортежа RSS прежде всего устанавливает захват этого кортежа в режиме S; при вставке, удалении или модификации кортежа - в режиме X. Если к этому моменту кортеж захвачен от имени другой транзакции в режиме, несовместимом с требуемым, транзакция, обратившаяся к RSS, блокируется, и требование захвата кортежа ставится в очередь до тех пор, пока не возникнут условия удовлетворения требуемого захвата, т.е. не будут сняты конфликтующие захваты с кортежа.

В принципе режим покортежных захватов достаточен для удовлетворения требований, ранее сформулированных в этом подразделе. Следует отметить, что большинство реляционных систем баз данных и ограничиваются покортежными (или даже более грубыми постраничными) захватами. Тем не менее в контексте System R синхронизация такого типа

недостаточна. Соответствующие проблемы и предложения по поводу их решения впервые были сформулированы в спецификации.

Основной проблемой является возможность появления корежей-*фантомов* при повторных сканированиях отношений в одной транзакции (даже на третьем уровне совместимости). Действительно, после первого сканирования все прочитанные кортежи будут захвачены в режиме S, из-за чего никакая другая транзакция не сможет удалить или модифицировать такие кортежи. Но ничто не мешает вставить в другой транзакции в то же отношение новый кортеж, значения полей которого удовлетворяют условиям сканирования первой транзакции. Тогда при повторном сканировании того же отношения с теми же условиями сканирования будет прочитан кортеж, которого не было при первом сканировании, т.е. появится кортеж-фантом.

В спецификации был предложен подход к решению проблемы фантомов на основе так называемых *предикатных захватов*. Идея подхода состоит в том, что при сканировании следует на самом деле захватывать не индивидуальные прочитанные кортежи, а все виртуальное множество кортежей, которые могли бы быть прочитаны при данном сканировании. Для этого достаточно "захватить" условие (предикат), которому должны удовлетворять все кортежи при данном сканировании. Например, если должны быть прочитаны кортежи отношения со значением поля a > 5, то захватывается предикат a > 5. Если некоторая другая транзакция выполняет операцию, например, вставки в то же отношение кортежа со значением поля a > 5, то предикатный захват, предшествующий реальному выполнению этой операции, должен конфликтовать с захватом первой транзакции и т.д.

Естественно, что чем более точно формулируется предикат, тем больше реальная параллельность выполнения транзакций в системе с предикатными захватами. С этой точки зрения наибольшего уровня асинхронности можно было бы достичь, если допустить использование в качестве синхронизационных предикатов условий выборки, удаления или модификации языка SQL (или другого языка запросов высокого уровня). Но с другой стороны, чем более общий вид предикатов допускается, тем сложнее становится проблема обнаружения совместимости соответствующих захватов. Как отмечается в спецификации, достаточно легко реализуется система предикатных захватов, в которой предикаты представляют собой логические выражения, состоящие из простых предикатов сравнения полей отношения с константными значениями. Разработчики System R не пошли и на такую систему, но, как мы покажем ниже, некоторые идеи предикатных захватов использовали.

При наличии только покортежных захватов неясным остается вопрос синхронизации в RSS покортежных операций и операций создания и уничтожения отношений и индексов и добавления полей к существующему отношению. Непонятно, как сочетать покортежные захваты с возможность явного захвата отношения и т.д. Очевидно, что общая система предикатных захватов такие проблемы снимает, но как мы уже отмечали, разработчики System R не пошли на ее реализацию. Вместо этого, как описывается в спецификации к базовой версии, был разработан протокол иерархических *гранулированных* захватов (с некоторыми элементами предикатных захватов).

Основная идея состоит в том, что имеется некоторая иерархия памяти хранения кортежей: сегмент-отношение-кортеж. Объекты каждого уровня

иерархии могут быть захвачены. Кроме того, можно захватывать диапазон значений любого индекса. Набор возможных режимов захватов расширяется так называемыми *целевыми* (intented) захватами. Семантически целевой захват "сложного" объекта (сегмента или отношения) означает намерение данной транзакции устанавливать целевые или обычные захваты на более низком уровне иерархии. Введены следующие типы целевых захватов: IX (intented to X), IS (intented to S) и SIX (shared, intented to X).

Захват объекта в режиме IX соответствует намерению транзакции производить захваты объектов ниже по иерархии в режимах IX или X. Захват объекта в режиме IS соответствует намерению транзакции производить захваты объектов ниже по иерархии в режимах IS или S. Наконец, захват объекта в режиме SIX соответствует захвату объекта в режиме S и намерению транзакции захватывать объекты ниже по иерархии в режиме X. Из этого следуют правила совместимости захватов разных режимов.

Протокол синхронизации с использованием перечисленных режимов захватов следующий: Чтобы захватить объект (например, кортеж отношения) в режиме S (X), предварительно установить захваты в режиме IS (IX) соответствующие объекты выше по иерархии (в случае захвата кортежа - сегмент и отношение). При этом захваты должны устанавливаться, начиная от корня иерархии (в нашем случае - сначала для сегмента, затем для отношения и только потом для кортежа).

Очевидно, что протокол иерархических захватов решает проблему совместимости глобальных захватов сложного объекта (например, захватов отношения в режиме S) с захватами подобъектов этого объекта (кортежей). Но также очевидно, что протокол, вообще говоря, не решает проблему фантомов. Если отношение сканируется без использования индекса, то отсутствие фантомов можно гарантировать, если предварительно захватить все отношение в режиме S. Тогда в соответствии с иерархическим протоколом никакая другая транзакция не сможет занести в это отношение новый кортеж, потому что будет блокирована при попытке захватить отношение в режиме IX (захваты отношения в режимах S и IX несовместимы). Можно, конечно, захватывать все отношение и при сканировании с использованием индекса. Таким образом можно решить проблему фантомов, но это очень неэффективное решение, потому что резко ограничивает возможности параллельного выполнения транзакций. Любая только читающая отношение транзакция конфликтует с любой транзакций, изменяющей это отношение. С другой стороны, в RSS при сканировании отношения по индексу имеется дополнительная информация (диапазон сканирования), которая ограничивает множество кортежей, среди которых не должны возникать фантомы.

Исходя из этих соображений было предложено ввести в систему синхронизации элементы предикатных захватов. Заметим сначала, что технически захваты сегментов, отношений и кортежей трактуются единообразно, как захваты tid'ов. При захвате кортежа на самом деле захватывается его tid. При захвате сегмента или отношения на самом деле захватывается tid описателя соответствующего объекта во внутренних отношениях описателей таких объектов (сегментов и отношений). Предлагалось расширить систему синхронизации, разрешив применять захваты к паре идентификатор индекса - интервал значений ключа этого индекса. К такой

паре разрешено применять захваты в любом из допустимых режимов, причем два захвата совместимы в том и только в том случае, если они совместимы в соответствии с таблицей на Рис.5 или указанные диапазоны значений ключей не пересекаются.

При наличии такой возможности, если открывается сканирование отношения по индексу, отношение захватывается в режиме IS, и в этом же режиме захватывается пара идентификатор индекса - диапазон сканирования. При занесении (удалении) кортежа отношение захватывается в режиме IX, и в этом же режиме для каждого индекса, определенного на данном отношении захватывается пара идентификатор индекса - значение ключа из затрагиваемого операцией кортежа. Тогда читающие транзакции реально конфликтуют только с теми изменяющими транзакциями, которые затрагивают диапазон сканирования. При этом решается проблема фантомов и асинхронность транзакций ограничивается "по существу", т.е. только тогда, когда их параллельное выполнение влечет проблемы.

Заметим сразу, что описанное решение проблем синхронизации далеко от идеального. Во-первых, по-прежнему при сканировании отношения без использования индексов отсутствие фантомов можно гарантировать только при полном захвате всего отношения в режиме S. Во-вторых, даже при сканировании по индексу условие реальной выборки кортежа часто может быть строже простого указания диапазона сканирования, а это значит, что требуемый захват слишком сильный, т.е. охватывает более широкое множество кортежей, чем то, которое будет реальным результатом сканирования.

Видимо, по этим причинам, а также по причинам требуемого усложнения системы синхронизации описанные средства борьбы с фантомами не были реализованы в System R (по крайней мере, это следует из заключительных публикаций). Более того, в силу половинчатости этого решения и слишком большого ограничения степени асинхронности разработчики отказались и от неявных захватов отношения в режиме S при сканировании без использования индексов. (Напомним, что возможность явного захвата отношения целиком осталась). Тем самым, System R не гарантирует отсутствие фантомов при повторном сканировании отношения.

Опыт System R в области синхронизации оказал очень большое влияние на разработчиков реляционных СУБД во всем мире. Особенно это касается предложений по части предикатных захватов. В ряде существующих или пректируемых СУБД предикатные захваты составляют основу системы синхронизации, которая, конечно, при этом становится существенно более сложной, чем в System R.

В заключение данного подраздела кратко упомянем о еще одном уровне синхронизации, присутствующем в RSS, - уровне *физической* синхронизации. Мы уже отмечали, что после выполнения любой операции RSS оставляет базу данных в физически согласованном виде. Это означает, в частности, корректность всех межстраничных ссылок. Примерами таких ссылок могут быть ссылки между станицами B-деревьев индексов и т.д. Во время выполнения операций изменения (занесения, модификации или удаления кортежа) может возникать временная некорректность состояния страниц данных. Для того, чтобы каждая операция при начале своего выполнения имела корректную информацию, необходима дополнительная кратковременная синхронизация

уровня страниц. На время выполнения операции все необходимые страницы захватываются в режиме чтения или изменения. Захваты снимаются при оканчивании выполнения операции.

И последнее замечание. При синхронизации транзакций могут возникнуть тупиковые ситуации, когда две или более транзакции не могут продолжать свое выполнение по причине взаимных блокировок. RSS не предпринимает каких-либо действий по предотвращению тупиков. Вместо этого периодически проверяется состояние системы захватов на предмет обнаружения тупика, и если тупик обнаруживается, выбираются одна или несколько транзакций жертв, для которых инициируется откат к началу или к ближайшей точке сохранения транзакции, гарантирующий разрушение тупика (при откате транзакции ее синхрозахваты снимаются). Выбор жертвы производится в соответствии с критериями минимальной стоимости проделанной транзакцией работы, которую придется повторить после отката. Мы не будем более подробно рассматривать схемы обнаружения и разрушения тупиков. Заметим лишь, что при обнаружении тупика применяется широко распространенная техника редукции графа ожидания с целью обнаружения в нем циклов, наличие которых и свидетельствует о наличии тупика.

3.5. Журнализация и восстановление в System R

Одно из основных требований к любой системе управления базами данных состоит в том, что СУБД должна *надежно* хранить базы данных. Это означает, что СУБД должна поддерживать средства восстановления состояния баз данных после любых возможных сбоев. К таким сбоям относятся индивидуальные сбои транзакций (например, деление на ноль в прикладной программе, инициировавшей выполнение транзакции); сбой процессора при работе СУБД (так называемые *мягкие* сбои) и сбои (поломки) внешних носителей, на которых расположены базы данных (*жесткие* сбои).

Ситуации, возникающие при сбоях каждого из отмеченных классов, различны и, вообще говоря, требуют разных подходов к организации восстановления баз данных. Индивидуальные сбои транзакций означают, что все изменения, произведенные в базе данных некоторой транзакцией, незаконны, и их необходимо устранить. Для этого необходимо выполнить индивидуальный откат транзакции такого же типа, как при выполнении RSS явной операции RESTORE.

При возникновении мягкого сбоя системы утрачивается содержимое оперативной памяти. Восстановление состояния базы данных состоит в том, что после завершения этого процесса база данных должна содержать все изменения, произведенные транзакциями, закончившимися к моменту сбоя, и не должна содержать ни одного изменения, произведенного транзакциями, которые к моменту сбоя не закончились. Существенным аспектом ситуации является то, что состояние базы данных на внешней памяти не разрушено, что позволяет сделать процесс восстановления не слишком длительным.

Жесткие сбои приводят к полной или частичной потере содержимого баз данных на внешней памяти. Тем не менее цель процесса восстановления та же, что и в случае мягкого сбоя: после завершения этого процесса база данных должна содержать все изменения, произведенные транзакциями,

закончившимися к моменту сбоя, и не должна содержать ни одного изменения, произведенного транзакциями, не закончившимися к моменту сбоя. В случае жесткого сбоя единственно возможный подход к восстановлению состояния базы данных может быть основан на использовании ранее произведенной копии базы данных. В общем случае процесс восстановления после жесткого сбоя существенно более накладен, чем после мягкого сбоя.

Как отмечает Дейт, любое восстановление состояния базы данных должно основываться на наличии некоторой дополнительной, избыточной информации. Из общих соображений очевидно, что для восстановления базы данных необходима информация, которая не теряется при сбоях, вызывающих потребность в восстановлении. Тем самым, задача любой СУБД и System R, в частности, состоит в выборе некоторого базового механизма поддержки такой избыточной информации, который удовлетворял бы потребностям всех используемых типов восстановлений.

Алгоритмы восстановления System R основаны на двух базовых средствах - ведении журнала и поддержке теневых состояний сегментов. Рассмотрим сначала механизм журнализации. Мы уже упоминали о наличии журнала в предыдущих подразделах. Журнал это отдельный файл внешней памяти, для которого для надежности обычно поддерживаются две копии и в который помещается информация по поводу всех операций изменения состояния базы данных. В предыдущем подразделе мы упоминали об использовании журнала для отката транзакции по явной операции RESTORE или при неявных откатах при разрушении тупиков. Та же схема употребляется и при откатах индивидуальных транзакций при сбоях.

Механизм индивидуального отката основан на обратном выполнении всех изменений, произведенных данной транзакции (undo). При этом из журнала в обратном хронологическому порядке выбираются все записи об изменении базы данных, произведенные от имени данной транзакции. Для этого необходима идентификация всех записей в журнале. В System R все записи одной транзакции связываются в один список в порядке обратном хронологическому. Ссылка в списке представляет собой адрес записи в файле-журнале. Поскольку схема индивидуального отката едина для всех ситуаций индивидуальных сбоев, в частности для ситуации разрушения тупиков, то обратное выполнение операций сопровождается снятием установленных при прямой работе транзакции синхронизационных захватов с объектов базы данных. Следовательно, после выполнения индивидуального отката транзакции ситуация в системе такова, как если бы транзакция никогда и не начиналась.

Специфика мягкого сбоя системы состоит в утрате состояния оперативной памяти. В оперативной памяти находятся буфера базы данных. Поддерживаются буфера двух сортов: буфера журнала и буфера собственно базы данных. Буфера журнала содержат последние записи в журнал. Имеются два буфера журнала. Как только один буфер полностью заполняется, производится его запись в файл-журнал и продолжается заполнение второго буфера. Таким образом при обычной работе системы обмены с файлом журнала не приводят к приостановке работы. Буфера базы данных содержат копии страниц базы данных, которые использовались в последнее время. В силу обычных в программировании принципов локализации ссылок после помещения копии страницы базы данных в буфер достаточно вероятно, что

эта страница потребуется в ближайшем будущем. Поэтому наличие копии страницы в буфере позволит избежать потребности в обмене с устройством внешней памяти, когда эта страница понадобится в следующий раз.

Заметим, что размер буферного пула СУБД во многом определяет ее производительность. Более того, как отмечает Дейт, обычная реляционная СУБД такая, как System R, при наличии достаточного размера буферного пула вполне конкурентноспособна по отношению к системам, основанным на специализированной аппаратуре машин баз данных.

Задача System R по обеспечению надежного завершения транзакций, т.е. гарантированию наличия произведенных ими изменений в базе данных, требует наличия на внешней памяти информации об этих изменениях. Реально для этого при оканчивании любой транзакции поддерживается гарантированное присутствие в файле-журнале все записей об изменениях, произведенных этой транзакцией. При использовании буферизации для записи в журнал для этого достаточно насильственно вытолкнуть на внешнюю память недозаполненный буфер журнала. Под насильственным выталкиванием понимается запись буфера на внешнюю память в соответствии не с логикой ведения журнала, а с логикой оканчивания транзакции. Только после произведения такого насильственного выталкивания буфера журнала транзакция считается закончившейся. Заметим, что последней записью в журнале от любой изменяющей базу данных транзакции является запись о конце транзакции. Эти записи используются при восстановлении. Рассмотрим теперь (пока не совсем точно) как осуществляется в System R восстановление базы данных после мягкого сбоя.

Основой алгоритма восстановления является то, что система придерживается правила *упреждающей записи в журнал* (WAL Write Ahead Log). Это правило означает, что при выталкивании любой страницы из буфера страниц сначала гарантируется наличие в файле журнала записи, относящейся к изменениям этой страницы после момента ее вталкивания в буфер. Поскольку записи в журнал блокируются, то для соблюдения правила WAL перед выталкиванием страницы данных необходимо вытолкнуть недозаполненный буфер журнала, если он содержит запись, относящуюся к изменению страницы. Применение правила WAL гарантирует, что если на внешней памяти находится страница базы данных, то в файле журнала находятся все записи об операциях, вызвавших изменение этой страницы. Обратное неверно: в файле журнала могут содержаться записи об изменении некоторых страниц базы данных, а сами эти изменения могут быть не отражены в состояниях страниц на внешней памяти.

При окончании любой транзакции (т.е. выполнении операции RSS END TRANSACTION) производится выталкивание недозаполнnного буфера журнала и тем самым гарантируется наличие в журнале полной информации обо всех изменениях, произведенной данной транзакцией. Насильственное выталкивание страниц буфера базы данных не производится (слишком накладно было бы производить такие выталкивания при окончании любой транзакции). Тем самым после мягкого сбоя состояние базы данных на внешней памяти может не соответствовать тому, которое должно было бы быть после оканчивания транзакций. Следовательно, после мягкого сбоя некоторые страницы на внешней памяти могут не содержать информации, помещенной в них уже закончившимися транзакциями, а другие страницы

могут содержать информацию, помещенную транзакциями, которые к моменту сбоя не закончились. При восстановлении необходимо добавить информацию в страницах первого типа и удалить информацию в страницах второго типа.

System R периодически устанавливает *системные контрольные точки*. Более подробно мы остановимся на этом ниже. Пока заметим лишь, что при установлении такой контрольной точки производится насильственное выталкивание на внешнюю память буфера журнала и всех буферов страниц. Это дорогостоящая операция, и выполняется она достаточно редко. При каждой системной контрольной точке в журнал помещается специальная запись.

Предположим, что последняя системная контрольная точка устанавливалась в момент времени tc, я мягкий сбой произошел в некоторый более поздний момент времени tf. Тогда все транзакции системы можно разбить на пять категорий. Транзакции категории T1 начались и кончились до момента tc. Следовательно, все произведенные ими изменения базы данных надежно находятся на внешней памяти, и по отношению к ним никаких действий при восстановлении производить не нужно. Транзакции категории T2 начались до момента tc, но успели кончиться к моменту мягкого сбоя tf. Изменения, произведенные такими транзакциями после момента tc, могли не попасть на внешнюю память, и при восстановлении должны быть повторно произведены. Транзакции категории T3 начались до момента tc, но не кончились к моменту сбоя. Все их изменения, произведенные до момента tc, и, возможно, некоторые изменения, произведенные после момента tc, содержатся на внешней памяти. При восстановлении их необходимо удалить. Транзакции категории T4 начались после момета установки системной контрольной точки и успели закончиться до момента сбоя. Их изменения могли не отобразиться на внешнюю память; при восстановлении их необходимо выполнить повторно. Наконец, транзакции категории T5 начались после момента tc и не закончились к моменту сбоя. Их изменения должны быть удалены из страниц на внешней памяти.

В принципе можно было бы выполнить все необходимы восстановительные действия после мягкого сбоя, основываясь только на информации из журнала. Соответствующий алгоритм описан в спецификации. Однако, в System R ситуация несколько упрощается за счет применения техники теневых страниц. Принцип теневых страниц давно использовался в файловых системах), поддерживающих файлы со страничной организацией. В соответствии с этим принципом после открытия файла на изменение модифицированные страницы записываются на новое место внешней памяти (т.е. под них выделяются свободные блоки внешней памяти). При этом на внешней памяти сохраняется старая (теневая) таблица отображения страниц файла на внешнюю память, а в оперативной памяти по ходу изменения файла формируется новая таблица. При закрытии файла заново сформированная таблица записывается на внешнюю память, образуя новую теневую таблицу, а блоки внешней памяти, содержащие предыдущие образы страниц файла, освобождаются. При сбое процессора тем самым автоматически сохраняется состояние файла, в котором он находился перед последним открытием (конечно, с возможной потерей некоторых блоков внешней памяти, которые затем собираются с помощью специальной утилиты). Допускаются операции явной фиксации текущего состояния файла и явного отката состояния файла к точке последней фиксации.

В System R применяется развитие идей теневого механизма в контексте мультидоступных баз данных. Как мы уже отмечали, сегменты баз данных System R представляют собой файлы со страничной организацией. Соответственно, существуют и таблицы приписки этих файлов на блоки внешней памяти. При выполнении операции установки системной контрольной точки после выталкивания буферов страниц на внешнюю память таблицы отбражения всех сегментов также фиксируются на внешней памяти, т.е. становятся теневыми. Далее до следующей контрольной точки доступ к страницам сегментов производится через таблицы отображения, располагаемые в оперативной памяти, и каждая изменяемая страница любого сегмента записывается на новое место внешней памяти с коррекцией соответствующей текущей таблицы отображения.

Тогда, если происходит мягкий сбой, все сегменты автоматически переходят в состояние, соответствующее последней системной контрольной точке, т.е. изменения, произведенные позже момента установления этой контрольной точки, в них просто не содержатся.

Это достаточно сильно упрощает процедуру восстановления после мягкого сбоя. Система вообще не должна предпринимать никаких действий по отношению к изменениям транзакций типа T5: этих изменений нет на внешней памяти. При восстановлении достаточно выполнить обратные изменения транзакций типа T3 (undo в терминологии System R), повторно выполнить изменения транзакций типа T2 (redo в терминологии System R; заметим, кстати, что эти изменения можно теперь выполнять безусловно, не заботясь о том, что они, возможно, и так содержатся на внешней памяти). Кроме того, нужно просто повторить изменения транзакций типа T4. Естественно, что начинать действия по журналу следует с записи о последней конрольной точке.

Справедливости ради, отметим, что на самом деле теневой механизм используется в System R главным образом не для упрощения процедуры восстановления после мягкого сбоя. Как мы уже отмечали, без этого можно обойтись. Главная причина в другом, а именно, в том, что восстановление базы данных можно начинать только от ее физически согласованного состояния. Дело в том, что в журнал помещается информация об изменении объектов базы данных, а не страниц. Например, в журнале может находиться информация о модификации кортежа в виде триплета <tid, старое состояние кортежа, новое состояние кортежа>. Реально же при выполнении операции модификации изменяются несколько страниц: исходная страница; возможно, страница замены, если кортеж не поместился в исходную страницу; страницы индексов. И так происходит при выполнении любой операции изменения базы данных. Поскольку буфера страниц выталкиваются на внешнюю память по отдельности, то к моменту мягкого сбоя на внешней памяти может возникнуть набор физически рассогласованных страниц, не соответствующий никакой журнализуемой операции. При таком состоянии внешней памяти восстановление по журналу невозможно.

Когда выполняется операция установки системной контрольной точки, то до насильственного выталкивания буферов страниц система дожидается завершения всех операций всех транзакции и до окончания выталкивания не допускает выполнения новых операций. Поэтому теневое состояние всех сегментов базы данных физически согласовано и может служить основой

восстановления по журналу.

При жестких сбоях утрачивается содержимое всех или части сегментов базы данных. Для восстановления базы данных используются журнал и ранее произведенная копия базы данных. В System R допускается посегментное восстановление. Для этого копия сегмента переписывается с архивного носителя на заново выделенный рабочий носитель, а затем по журналу повторяются все изменения, производившиеся в объектах этого сегмента после момента копирования. Поскольку в момент жесткого сбоя содержимое оперативной памяти не утрачивается, то возможно продолжение выполнения транзакций после завершения восстановления. Более того, если авария коснулась только части сегментов базы данных, то транзакции могут продолжать работу на фоне процесса восстановления с объектами базы данных, расположенными в неповрежденных сегментах.

Единственным требованием к архивной копии сегмента является то, что страницы в ней должны находиться в физически согласованном состоянии (поскольку восстановление ведется в терминах записей журнала). Поэтому для создания архивной копии сегмента достаточно лишь дождаться конца выполнения операций над объектами данного сегмента и запретить начало новых операций до конца копирования. Тем самым, выполнение архивной копии не требует перевода системы в какой-либо особый режим работы и только незначительно тормозит нормальную работу транзакций.

В заключение данного подраздела заметим, что в первых версиях System R в качестве архивного носителя использовались магнитные ленты. Однако, как отмечается, со временем стало ясно, что во-первых, надежность магнитных лент существенно меньше надежности магнитных дисков, а во-вторых, они стали уступать и в емкости. Поэтому в последних версиях системы использовалась только дисковая память.

И последнее замечание. Журнал System R располагается в файле большого, но постоянного размера. Он используется в циклическом режиме. Когда записи журнала достигают конца файла, они начинают помещаться в его начало. Поскольку переход на начало файла можно считать утратой предыдущего журнала, этот переход сопровождается копированием сегментов базы данных. В некоторых других системах, как отмечается, используется подход с архивизацией самого журнала.

4. Informix в качестве хранилищ данных САПР

Всплеск активности по поводу объектно-ориентированного подхода (ООП), возникший в первой половине 80-х, в начале 90-х докатился и до систем управления базами САПР. Одна за другой стали возникать небольшие фирмы, предлагавшие свои решения, свое видение интеграции ООП и САПР. Затем стали проявлять активность и фирмы-гиганты, которые доминировали на рынке САПР. В результате на сегодняшний день уже очень многие фирмы-производители САПР или поставляют сервера баз данных с поддержкой ООП, или объявили о своей готовности сделать это.

Фирма Informix Software объявила о своем намерении выпустить сервер СУБД с поддержкой ООП еще в 1991 году, когда была начата разработка новой архитектуры Informix OnLine Dynamic Server. В декабре 1996 года появился Informix Universal Server, который был сделан на основе архитектуры Informix Dynamic Scalable Architecture и обеспечивал поддержку объектно-ориентированного подхода. В 1999 году появились идеи использования СУБД Informix в качестве хранилищ данных САПР. В последствии СУБД Informix разрабатываласьи дополнялась с учетом специфики САПР.

4.1. Зачем нужна поддержка объектов в серверах БД?

Самая распространенная модель данных в настоящее время - это реляционная модель. Практически все фирмы-производители систем управления базами данных ориентируются или, хотя бы, поддерживают язык SQL как стандартный язык доступа к реляционным базам данных. Реляционная модель очень хорошо подходит для решения большинства задач. Однако не все задачи могут быть одинаково хорошо и изящно решены с помощью реляционных баз данных вообще и языка SQL в частности. Рассмотрим те ограничения и неудобства, которые заложены в реляционные СУБД.

Понятие отношения предполагает, что у каждого реального объекта или у каждой реальной сущности, описываемой записью в отношении, есть некоторое количество атрибутов. С точки зрения реляционной модели все эти атрибуты равноправны, находятся на одном уровне абстрации. Это не означает, конечно, что между атрибутами совсем нет никакой разницы. Например, бывают ключевые и не ключевые атрибуты. Суть в том, что атрибуты не могут быть сгруппированы, они не могут быть собраны в иерархию. Все атрибуты в отношении находятся «на одном уровне».

На самом деле, в реальных задачах некоторые атрибуты более важны, нежели другие. С другой стороны, некоторые атрибуты должны быть сгруппированы, структурированы. Проблема группирования атрибутов может быть решена на понятийном уровне до тех пор, пока этих атрибутов не так много. Когда число атрибутов в таблице переваливает за несколько десятков, в аттрибутах уже легко запутаться.

Рассмотрим пример. Предположим, нам надо реализовать в реляционном отношении (в таблице) список сотрудников предприятия для местного отдела кадров. В такой таблице должны быть следующие атрибуты - фамилия, имя, отчество, табельный номер, отдел, должность, дата приема на работу и т.д:

```
CREATE TABLE persons (
        tabel_num       INTEGER,
{табельный номер}
        last_name       CHAR(40),
```

```
{фамилия}
        first_name      CHAR(40), {имя}
        second_name     CHAR(40),
{отчество}
        department      INTEGER,
{отдел}
        dolgnost        CHAR(20), {должность}
        emp_from        DATE,
{работает с...}
. . . . . .
)
```

При такой структуре данной таблицы все атрибуты «свалены» в кучу. Более правильно было бы сгруппировать фамилию, имя и отчество в новый составной тип под названием «ФИО», а отдел и должность - в тип «позиция»:

```
CREATE ROW TYPE fio_t (
        last_name       CHAR(40),
{фамилия}
        first_name      CHAR(40), {имя}
        second_name     CHAR(40)
{отчество}
)
CREATE ROW TYPE position_t (
        department      INTEGER,
{отдел}
        dolgnost        CHAR(20)  {должность}
)
```

После чего определить таблицу уже с учетом структурированных, упорядоченных атрибутов:

```
CREATE TABLE persons (
        tabel_num       INTEGER,
{табельный номер}
        fio             fio_t,    {ФИО}
        position        position_t, {позиция}
        emp_from        DATE,
{работает с...}
        . . . . . .
)
```

Такое группирование, структурирование атрибутов снижает вероятность ошибки, делает и схему базы данных, и программы, работающие с этой базой, более читаемыми и, следовательно, более надежными.

В реляционных СУБД набор типов данных, которые можно использовать для атрибутов, фиксирован. Этот набор включает в себя целые числа, числа с плавающей и фиксированной запятой, типы данных для указания времени и даты, а также символьные строки. Подобный набор характерен для финансовых приложений, приложений складского характера и т.д. Современная тенденция такова, что растет число приложений, где необходимо совместить сервер баз данных и мультимедийные типы данных (видео, звук, форматированные тексты). С другой стороны, появляется необходимость обрабатывать новые типы данных, которые не представлены в SQL, а определяются конкретной предметной областью (отпечатки пальцев, фотографии, схемы и т.д.).

Нельзя сказать, что современные реляционные СУБД такой возможности не предоставляют совсем. Например, такие типы данных в Informix Dynamic Server, как BYTE и TEXT предназначены для хранения больших (до 2 ГБт)

двоичных и текстовых объектов соответственно. Сервер обеспечивает только хранение таких объектов, он не может сам обрабатывать данные этих типов. Вся обработка должна проводиться на программе-клиенте. То есть для того, чтобы, например, просмотреть все имеющиеся в базе данных отпечатки пальцев и отобрать те, которые похожи на заданный образец, программа-клиент должна последовательно просмотреть все имеющиеся отпечатки. Проведение такого просмотра самим сервером, а, тем более, наличие индекса по отпечаткам, значительно бы ускорило обработку такого запроса.

Могут существовать и специализированные СУБД (или специализированные расширения универсальных СУБД), предназначенные для той или иной конкретной предметной области. Однако, производители СУБД не раскрывают свои внутренние интерфейсы, поэтому для разработчика прикладной системы написать подобное расширение не представляется возможным, а полностью разрабатывать свою собственную специализированную СУБД экономически невыгодно.

Следовательно, возможность расширения набора базовых типов, то есть возможность разрабатывать новые типы данных и внедрять их в СУБД увеличит эффективность использования СУБД для широкого класса задач. При введении нового базового типа крайне желательно иметь возможность определять для этого типа внутреннее представление (формат), описывать при необходимости новые методы хранения, доступа, индексирования и т.д. При определении нового базового типа для него имеет смысл доопределять те операции, которые уже существуют в языке доступа к данным (например, в SQL). Другими словами, в серверах БД необходимо обеспечить поддержку концепции абстрактных типов данных.

Рассмотрим простейший пример. Предположим, нам надо хранить информацию об адресах и делать выборку по улице. В Российских городах для названия улиц может использоваться дополнительная нумерация (1-я Парковая, 9-я Парковая) или прилагательное «Большой» или «Малый» (Малая Бронная). Соответственно, существует несколько способов записи таких названий. Следующие три названия обозначают одну и туже улицу:

```
М.Бронная
Бронная, М.
Малая Бронная
```

Очевидно, что если мы будем использовать для хранения названия улиц обычные строковые типы, то придется следить за тем, чтобы в базе данных названия улиц хранились в едином, унифицированном формате. А при запросе на поиск по названию улицы потребуется преобразовывать введенное пользователем название к этому единому формату. Было бы удобнее, если бы можно было определить новый тип данных «Название улицы», который бы делал это автоматически и, следовательно, минимизировал возможные ошибки. Для данного типа нужно было бы переопределить операцию сравнения, чтобы при сравнении двух значений учитывалось разное написание улиц. Кроме того, для данного типа потребуется переопределить операции сравнения, чтобы сортировка значений данного типа проводилась с учетом разных написаний:

```
SELECT streets FROM table1 ORDER by streets
. . . .
Бауманская
Б.Бронная
```

М.Бронная
Буденного
. . . .

Можно привести и другие примеры, когда стандартного набора базовых типов недостаточно. Введение в ту или иную СУБД новых предопределенных типов не может решить данную проблему - всегда найдется задачу, которую не предусмотрели.

4.2. Иерархия данных

В очень многих случаях, данные в той или иной предметной области, образуют иерархическую структуру. Например, на одном предприятии работают продавцы, инженеры и администрация. С точки зрения отдела кадров, структура информации о каждом из сотрудников идентична и включает в себя фамилию, имя, отчество, должность, дату рождения, дату приема на работу и т.д. Эта структура (набор атрибутов) не зависит от того, на какой должности нааходится тот или иной сотрудник.

С другой стороны, способ начисления зарплаты для каждой категории разный. Инженерные работники получают оклад плюс премию как процент от выполнения плана, административные работники получают строгий оклад, а продавцы имеют и оклад, и процент от заключенных сделок. Следовательно, для бухгалтерии при расчете зарплаты требуется в зависимости от позиции человека производить разный расчет и использовать разные атрибуты.

4.3. Концепция объектно-ориентированного подхода

Концепция объектно-ориентированного подхода возникла в конце 70-х - начале 80-х как альтернатива традиционному стилю программирования. Традиционный стиль программирования подразумевал главенствование алгоритма, программы над данными. Такой подход хорошо подходил для вычислительных задач, относительно неплохо подходил для коммерческих, в основном, учетно-расчетных задач. Как альтернатива традиционному подходу постепенно сформировался объектно-ориентированный подход.

Объектно-ориентированный подход предполагал сосредоточиться на описании предметной области как некоторого набора объектов, которые общаются между собой, к которым можно обратиться с запросами, но про внутреннее устройство которых нам на данном уровне абстракции знать не надо. Данная технология не предполагала никаких специальных конструкций в языках программирования - разрабатывать программу с использованием данной технологии можно было на любых языках (С, Фортран, Паскаль, Ассемблер и т.д.).

Следующим шагом была разработка Б.Страуструпом языка С++, который предназначался для поддержки технологии объектно-ориентированного программирования. Для осуществления такой поддержки в С++ был реализован механизм абстрактных типов данных (с такими свойствами, как полиморфизм и инкапсуляция) и механизм наследования типов. Такой симбиоз получился настолько удачным, что сейчас многие просто отождествляют объектно-ориентированное программирование с поддержкой инкапсуляции, полиморфизма, наследования.

4.4. Объектно-ориентированные СУБД

СУБД только тогда может считаться объектно-ориентированной, когда она поддерживает ряд описанных в документе объектно-ориентированных свойств.

Но кроме того, что бы поддерживать в той или иной степени объектно-ориентированную технологию, такая система должна оставаться собственно базой данной и решать связанные с этим вопросы. А именно, обеспечивать операции доступа и преобразования данных, одновременный доступ к данным нескольких пользователей, разграничение доступа и защиту данных от сбоев. В частности, ООСУБД должна предоставлять язык описания данных (ЯОД) и язык манипулирования данными (ЯМД). Язык манипулирования данными должен или быть встраиваемым в какой-либо язык программирования, или быть реализован в виде API.

В модели ODMG-93 были описаны и ЯОД, и ЯМД. К сожалению, пока ни одна из фирм не реализовала полностью этот стандарт. Именно отсутствие реально работающего стандарта является сдерживающим фактором в распространении объектно-ориентированных СУБД.

4.5. Объектно-реляционные СУБД

Реляционные СУБД, в отличии от «чистых» объектно-ориентированных СУБД, имеют реально работающие стандарты - стандарты на язык запросов SQL. Кроме того, существует огромная база заказчиков, которые пока не готовы отходить от реляционной технологии. И фирмы-производители реляционных СУБД пошли по пути внедрения объектной технологии в отработанную и популярную технологию реляционных СУБД. В частности, сейчас готовится к выходу стандарт SQL-3, в котором уже заложена поддержка объектно-ориентированной концепции.

Основная идея объектно-реляционного подхода - это допущение использовать в качестве атрибутов не только простые, атомарные типы данных, но и абстрактные типы данных. Кроме того, предполагается реализовать возможность наследования типов и данных.

Технология абстрактных типов данных предполагает:

- инкапсуляцию (сокрытие деталей реализации внутри типа);
- полиморфизм (применимость одной операции к разным типам и разный способ вычисления в зависимости от типа);
- позднее связывание (определение реального типа объекта в момент исполнения);
- расширяемость (возможность определить новый тип);
- наследуемость типов (возможность определить новый тип на основе существующего);

Данное предложение, формально говоря, противоречит концепции реляционных СУБД. Первая нормальная форма требует, чтобы значения атрибутов были только атомарными. Однако, если ввести модифицированную первую нормальную форму, в которой допускалось бы неатомарность атрибутов, то все последующие выводы и рассуждения реляционной

теории можно будет приложить и к модифицированной первой нормальной форме. Атомарность или неатомарность атрибутов явным образом нигде не используется, поэтому все выводы о полноте такой модели останутся верными.

Итак, объектно-реляционный подход представляет собой расширение классических реляционных СУБД, базирующихся на языке SQL. Язык SQL и, соответственно реляционная модель, расширяются возможностью вводить абстрактные типы данных и организовывать иерархию типов и данных. Следовательно, обеспечиваются все требования, ассоциируемые с объектно-ориентированной технологией (см. Объектно-ориентированные СУБД), но при этом обеспечивается совмест совместимость со старыми приложениями, инструментами и технологиями.

4.6. Реализация объектного подхода в Informix

Informix Universal Server представляет собой реализацию объектно-ориентированной технологии на основе встраивания механизма абстрактных типов данных и механизма наследования в популярный и надежный сервер реляционных баз данных Informix Dynamic Server.

Встраиваемая объектно-ориентированная технология была известна на практике по объектно-реляционной СУБД Illustra (позднее, приобретеннной фирмой Informix и называвшейся Informix Illustra).

4.7. Определение новых базовых типов в СУБД Informix

Informix Universal Server позволяет вводить новые базовые типы данных. При этом можно использовать как встроенные в Informix Universal Server методы доступа и хранения, так и определять новые. Рассмотрим способ создания новых базовых типов с использованием встроенных механизмов хранения. Создание базовых типов вместе с алгоритмами хранения и доступа будет рассмотрено ниже.

Необходимость в создании новых базовых типов может возникать во многих случаях. Одним из самых простых случаев - это использование разных метрических систем для одного и того же понятия. Например, если некоторая фирма закупает детали в Америке, то их размеры будут указаны в футах, а цена в долларах, если аналогичные детали закупаются в Германии, то их размеры указываются в метрической системе, а цена в марках.

Если бы речь шла только про размеры, то алгоритм перевода размеров в единую систему измерений четко известен и в базе данных можно было бы хранить только размеры в метрах и сантиметрах. С ценами сложнее - курс обмена зависит от даты конвертации. И если мы будем искать деталь с минимальной ценой, то надо учитывать текущий курс. ОРСУБД Informix Universal Server позволяет построить новый базовый тип данных, основанный на существующем, но обеспечивающий автоматическое преобразование к нужному значению. Сами типы вводятся следующими операторами:

```
CREATE DISTINCT TYPE usd AS MONEY;
CREATE DISTINCT TYPE dm AS MONEY;
```

Далее надо ввести функции преобразования значений из долларов в марки и наоборот, а также описать возможность такого преобразования:

```
CREATE FUNCTION usd_to_dm(v usd) RETURNS dm; . . .
CREATE FUNCTION dm_to_usd(v dm) RETURNS usd; . . .
CREATE IMPLICIT CAST (usd AS dm WITH usd_to_dm);
CREATE IMPLICIT CAST (dm AS usd WITH dm_to_usd);
```

После этого можно сравнивать значения типов usd и dm, полученные из разных таблиц, не вызывая явно функцию преобразования. Такое решение существенно снижает возможность внесения ошибок, связанных с преобразованием значений.

Другой причиной, по которой может возникнуть необходимость во введении нового базового типа данных - это принципиальное отсутствие такого типа. Например, для экспериментальных данных, которые будут храниться в нашей базе данных, недостаточна точность, обеспечиваемая стандартным типом FLOAT. Informix Universal Server позволяет ввести новый тип данных FLOAT16, будет использовать для хранения своих значений 16 байт и будет соответствовать нашим требованиям по числу значащих цифр в мантиссе и диапазону порядка:

```
CREATE OPAQUE TYPE float16 (INTERNALLENGTH=16, ALIGHNMENT=4);
```

Одного такого оператора недостаточно. Необходимо также задать функции преобразования значений данного вида в текстовый вид (тип данных LVARCHAR) и обратно (это нужно для ввода/вывода значений, экспорта/импорта базы и т.д.). Кроме того, нужно задать дополнительные функции преобразования и сравнения, которые будут использоваться при построении стандартных индексов и при сравнении со значениями других типов:

```
{обязательные функции
преобразования в строку и обратно}
CREATE FUNCTION float16_out(float16) RETURNING LVARCHAR . . . .;
CREATE FUNCTION float16_in
(lvarchar) RETURNING float16 . . . .;
{реализация стандартных операторов "+","-","*","/",">","<" и т.д.}
CREATE FUNCTION Plus(float16, float16) RETURNING float16 . . . .;
CREATE FUNCTION Plus(float16, float) RETURNING float16 . . . .;
. . . . . .
```

После того, как все нужные функции определены, можно использовать тип float16 наравне с другими базовыми типами (FLOAT, SMALLFLOAT, INTEGER и т.д.). При этом для хранения, поиска и индексирования используются стандартные механизмы Informix Universal Server.

4.8. Составные типы данных

Informix Universal Server позволяет определять новые составные типы данных. К доступным структурам, которые можно использовать для построения составных типов, относятся:

- запись
- множество
- список

Запись представляет собой возможность ввести именованные поля. Структура запись структуре record в языке Паскаль и struct в языке C/C++. Тип данных со структурой запись вводится оператором:

```
CREATE ROW TYPE <имя типа> (
        <имя поля> <тип поля>, . . .
)
Например:
CREATE ROW TYPE fio_t (
        last_name       CHAR(40),
{фамилия}
        first_name      CHAR(40), {имя}
        second_name     CHAR(40)
{отчество}
)
```

Созданный таким образом составной тип может использоваться наравне и с предопределенными типами для описания колонок в отношении.

```
CREATE TABLE persons (
        tabel_num       INTEGER,
{табельный номер}
        fio             fio_t,    {ФИО}
        . . . . . .
)
```

Для доступа к отдельным полям внутри типа записи используется традиционный синтаксис - через точку надо указать имя поля:

```
SELECT tabel_num, fio.last_name, fio.first_name FROM persons
        WHERE tabel_num = 157
```

Множество представляет собой неупорядоченное множество значений. В Informix Universal Server используется два варианта реализации структуры множества - set и multiset. Первая структура (будем называть ее просто множеством) не допускает повторений элементов внутри себя. Вторая структура (будем называть ее мультимножеством) допускает повторение элементов. Приведем пример, как можно ввести в таблицу persons в качестве атрибута тип данных «дети» со структурой SET:

```
CREATE TYPE children_t SET (
        fio             fio_t,
        wasborn DATE
)
CREATE TABLE persons (
        tabel_num       INTEGER,    {табельный номер}
        fio             fio_t,      {ФИО}
        children        children_t, {дети}
        . . . . .
)
```

Список очень похож на мультимножество, но его элементы упорядочены. То есть в списке могут быть повторяющиеся значения, и эти значения можно перебирать по порядку. Рассмотрим реализацию типа со структурой «список» для хранения трудовой биографии (учреждение, дата приема, должность):

```
CREATE TYPE lab_byog_t LIST (
        name            CHAR(20),
        work_from       DATETIME YEAR TO DAY,
        position        CHAR(10)
)
```

```
CREATE TABLE persons (
        tabel_num       INTEGER,    {табельный номер}
        fio             fio_t,      {ФИО}
        children        children_t, {дети}
        lab_byog        lab_byog_t, {трудовая биография}
        . . . . .
)
```

4.9. Наследование типов и данных

Пример, когда и инженеры, и администраторы, и продавцы с точки зрения работников отдела кадров выглядели совершенно одинаково, но по-разному обрабатывались в бухгалтерии, очень удобно может быть реализован с использованием иерархии типов и данных.

Прежде всего заметим, что имеется базовое понятие, сущность - сотрудник. Сотрудник характеризуется табельным номером, фамилией, именем, отчеством, должностью и окладом:

```
CREATE ROW TYPE employee_t (
        tabel_num       INTEGER,
{табельный номер}
        last_name       CHAR(40),
{фамилия}
        first_name      CHAR(40), {имя}
        second_name     CHAR(40),
{отчество}
        dolgnost        CHAR(20)
{должность}
        base_salary     MONEY     {оклад}
)
```

В этом типе данных содержится вся информация, которая может быть интересна отделу кадров предприятия. Однако, как мы выяснили выше, для финансового отдела нужны дополнительные сведения. Для продавцов и инженеров дополнительно должна храиться и другая информация, необходимая для расчета зарплаты. Эту информацию можно учесть, создав типы engineer_t и sale_t как наследники типа employee_t:

```
CREATE ROW TYPE engineer_t (
        bonus           DECIMAL (5,2)   {премия в процентах}
) UNDER TYPE employee_t
CREATE ROW TYPE sale_t (
        comission       DECIMAL (5,2),  {размер комиссионных в %}
        revenue MONEY                   {сумма заключенных котрактов}
) UNDER TYPE employee_t
```

Таким образом, имеется 3 типа данных, два из которых (sales_t и engineer_t) являются наследниками одного (employee_t). Если мы используем эти типы для создания таблиц, мы можем создать иерархию данных:

```
CREATE TABLE employees OF TYPE employee_t;
CREATE TABLE engineers OF TYPE engineer_t UNDER TABLE employees;
CREATE TABLE sales OF TYPE sale_t UNDER TABLE employees;
```

В результате, мы имеем не три разных независимых таблицы, а одну главную таблицу (employees) и две наследованные таблицы (sales и engineers).

ЗАКЛЮЧЕНИЕ

В данном реферате были рассмотрены две различные СУБД, применяемые в САПР. Были рассмотрены их основные концепции и представлен обзор решаемых с их помощью задач. Трудно переоценить, насколько велико влияние САПР на все аспекты нашей повседневной жизни. Благодаря использованию СУБД, САПР становится доступным не только узкому кругу специалистов, но и рядовому пользователю. В совокупности со Специальными Возможностями ОС Windows, все, даже самые скрытые от невооруженного глаза возможности, станут доступны даже неполноценным людям.

Список литературы

1. Бойко В.В., Савинков В.М. Проектирование информационной базы автоматизированной системы на основе СУБД. – М.: Финансы и статистика. 1982. – 174 с.

2. Вендров А.М. CASE-технологии. Современные методы и средства проектирования информационных систем. – М.: Финансы и статистика, 1998. – 176 с.

3. Вейнеров О.М., Самохвалов Э.Н. Проектирование баз данных САПР. – М.: Высшая школа, 1990. – 144 с.

4. Замулин А.В. Типы и модели данных //Банки данных: Материалы 3-й Всесоюзной конф. (Таллин, 24-26 сентября 1985 г.). – Таллин: ТПИ, 1985, с. 3-15.

5. Замулин А.В. Системы программирования баз данных и знаний. – Новосибирск: Наука. Сибирское отделение, 1990. – 351 с.

6. Как работать над терминологией. Основы и методы КНТТ АН СССР. – М.: Наука, 1968. – 76 с.

7. Когаловский М.Р. Проблемы терминологии в теории систем баз данных // УСиМ, 6, 1986, с. 85-92.

8. Когаловский М.Р. Архитектура механизмов отображения данных в многоуровневых СУБД //Техника реализации многоуровневых систем управления базами данных. – М.: ЦЭМИ АН СССР, 1982, с. 3-19.

9. Когаловский М.Р. Технология баз данных на персональных ЭВМ. – М.: Финансы и статистика, 1992. – 224 с.

10. Мальцев А.И. Алгебраические системы. – М.: Наука, 1970. – 392 с.

11. Михновский С.Д. Автоматизация проектирования баз данных. Общий анализ проблемы //УСиМ, 4, 1981, с. 35-44.

12. Пржиялковский В.В. Абстракции в проектировании баз данных //СУБД, 1-2/98, с. 90-97.

13. Савинков В.М., Вейнеров О.М., Казаров М.С. Основные концепции автоматизации проектирования баз данных //Прикладная информатика. Вып.1. – М.: Финансы и статистика, 1982, с. 30-41.

14. Зубок И.С. Проблемы больших баз данных в САПР на базе процессора R3000. – М.: Наука, 2002, 8325 с.

15. Евклид А.А. Моделирование плоских контуров и поверхностей на выпуклых поверхностях современных устройств вывода для САПР типа Earth&Stick. – А.: Самиздат, -2002, 2 с.