**Лекция 12. Пример общей организации СУБД. Физическое представление реляционных баз данных во внешней памяти. Индексные структуры**

**12.1. Введение**

В 1975-1979 г.г. в исследовательской лаборатории компании IBM разрабатывалась система управления реляционными базами данных System R. Эта работа оказала революционизирующее влияние на развитие теории и практики реляционных систем во всем мире. Именно System R практически доказала жизнеспособность реляционного подхода к управлению базами данных.

После успешного завершения работ по созданию этой системы и получения экспериментальных результатов ее использования был разработан целый ряд коммерчески доступных реляционных систем, в том числе и на основе непосредственного развития System R. Исключительно важен опыт, приобретенный при разработке этой системы. Практически во всех более поздних реляционных СУБД в той или иной степени используются методы, примененные в System R. Поэтому лекции, посвященные внутренней организации SQL-ориентированных СУБД, во многом опираются на материалы статей, посвященных System R.

Организации СУБД System R посвящена обширная библиография [[5.1–5.17]](http://citforum.ru/database/advanced_intro/87.shtml#ref.5) (здесь подобраны публикации, свободно доступные в Internet к моменту написания этого текста). Поскольку публикации появлялись по ходу практической реализации системы, каждая из них отражает состояние дел (идейное и практическое) именно на том этапе работы, когда была написана соответствующая статья. Некоторые идеи и представления, естественно, изменялись по ходу работы. Сравнительно законченное представление о системе в целом дают только заключительные публикации. Имеется обширный обзор литературы, посвященной System R и связанных с ней проектов [[3.6]](http://citforum.ru/database/advanced_intro/86.shtml#ref.3.6).

### 12.2. Основные понятия, цели и общая организация System R

Поскольку обсуждение принципов внутренней организации реляционных (точнее, SQL-ориентированных) СУБД в этой книге проводится в контексте System R, начнем с рассмотрения основных понятий этой системы.

#### 12.2.1. Используемая терминология

Несмотря на то, что при реализации System R использовался подход, несколько отличающийся от реляционного подхода Кодда (отсюда и пошли расхождения между реляционной моделью данных и моделью данных SQL), мы будем активно пользоваться терминами реляционной модели. К таким терминам относятся названия реляционных операций – ограничение, проекция, соединение; названия теоретико-множественных операций – объединение, пересечение, взятие разности и т.д.

В тех случаях, когда терминология System R расходится с реляционной терминологией, предпочтение будет отдаваться терминологии System R. В частности, это касается использования термина «поле таблицы» вместо термина «атрибут отношения». В самой System R при переходе к коммерческим системам также произошла некоторая смена терминологии. В частности, появилась тенденция к употреблению терминов, более привычных в среде пользователей IBM: файл, запись и т.д. Здесь будут использоваться термины System R, более близкие реляционным системам. Опишем некоторые основные термины System R, опираясь в основном не на теоретические соображения, а на практические аспекты соответствующих понятий.

Базовым понятием System R является понятие *таблицы* (приближенный к реализации аналог основного понятия реляционного подхода *отношения*; иногда, в зависимости от контекста, мы будем использовать и этот термин). Таблица – это регулярная структура данных, состоящая из конечного набора однотипных записей – кортежей. Каждый кортеж одной таблицы состоит из конечного (и одинакового) числа полей кортежа, причем *i*-тое поле каждого кортежа одной таблицы может содержать данные только одного типа, и набор допустимых типов данных в System R предопределен и фиксирован.

В силу регулярности структуры таблицы понятие поля кортежа расширяется до понятия поля таблицы. Тогда *i*-тое поле таблицы можно трактовать как набор одноместных кортежей, полученных выборкой *i*-тых полей из каждого кортежа этой таблицы, т.е. в общепринятой терминологии как проекцию таблицы на *i*-тый атрибут. В терминологию System R не входит понятие домена, оно заменяется здесь понятием типа поля, т.е. типом данных, хранение которых в данном поле допускается (это не вполне эквивалентная замена, но такова реальность System R).

Таблицы, составляющие базу данных System R, могут физически храниться в одном или нескольких сегментах, каждому из которых соответствует отдельный файл внешней памяти. Сегменты разбиваются на страницы, в которых располагаются кортежи таблиц и вспомогательные служебные структуры данных – индексы. Соответственно, каждый сегмент содержит две группы страниц – страницы данных и страницы индексной информации. Страницы каждой группы имеют фиксированный размер, но страницы с индексной информацией меньше по размеру, чем страницы данных. В страницах данных могут располагаться кортежи более чем одной таблицы (это очень важное свойство физической организации баз данных System R; следующие из этой организации преимущества разъясним позже).

Этим, конечно, не исчерпывается набор понятий System R, но остальные термины мы будем пояснять по ходу изложения, поскольку для этого требуется соответствующий понятийный контекст.

#### 12.2.2. Цели System R и их связь с общей организацией системы

При выполнении проекта System R преследовались следующие основные цели:

* обеспечить ненавигационный интерфейс высокого уровня пользователя с системой, позволяющий достичь независимости данных и дать возможность пользователям работать максимально эффективно;
* обеспечить многообразие допустимых способов использования СУБД, включая программируемые транзакции, диалоговые транзакции и генерацию отчетов;
* поддерживать динамически изменяемую среду баз данных, в которой таблицы, индексы, представления, транзакции и другие объекты могут легко добавляться и уничтожаться без приостановки нормального функционирования системы;
* обеспечить возможность параллельной работы с одной базой данных многих пользователей, допуская параллельную модификацию объектов базы данных при наличии необходимых средств защиты целостности базы данных;
* обеспечить средства восстановления согласованного состояния баз данных после разного рода сбоев аппаратуры или программного обеспечения;
* обеспечить гибкий механизм, позволяющий определять различные представления хранимых данных и ограничивать этими представлениями доступ пользователей к базе данных по выборке и модификации на основе механизма авторизации;
* обеспечить производительность системы при выполнении упомянутых функций, сопоставимую с производительностью существующих СУБД низкого уровня.

Прежде всего, отметим, что при разработке System R поставленные цели в основном были достигнуты. Рассмотрим теперь, какими средствами удалось достичь этих целей, и как можно более точно интерпретировать их в контексте System R.

Основой System R является «реляционный» язык SEQUEL (который достаточно быстро был переименован в SQL). Заметим, что разработчики System R искренне считали созданный ими язык реляционным; однако, как отмечалось в предыдущих лекциях и будет более детально показано в заключительных лекциях, в этом языке в действительности нарушаются многие важные принципы реляционной модели данных. Иногда его называют языком запросов или языком манипулирования данными, но на самом деле возможности SQL гораздо шире. Средствами SQL (с соответствующей системной поддержкой) решаются многие из поставленных целей. Язык SQL включает средства динамической компиляции запросов, на основе чего возможно построение диалоговых систем обработки запросов. Допускается динамическая параметризация статически откомпилированных запросов, в результате чего возможно построение эффективных (не требующих динамической компиляции) диалоговых систем со стандартными наборами (параметризуемых) запросов. Средствами SQL определяются все доступные пользователю объекты баз данных: таблицы, индексы, представления. Имеются средства уничтожения любого такого объекта. Соответствующие операторы языка могут выполняться в любой момент, и возможность выполнения операции данным пользователем зависит от ранее предоставленных ему прав.

Что касается целостности баз данных, то в System R под целостным состоянием базы данных понимается состояние, удовлетворяющее набору сохраняемых при базе данных предикатов целостности. Эти предикаты, называемые в System R *утверждениями целостности* (assertion), также задаются средствами языка SQL. Любой оператор языка выполняется в границах некоторой *транзакции* – последовательности операторов языка, неделимой в смысле состояния базы данных. Неделимость означает, что все изменения базы данных, произведенные в пределах одной транзакции, либо целиком отображаются в состоянии базы данных, либо полностью в нем отсутствуют. Последняя возможность возникает при *откате* транзакции, который может произойти по инициативе пользователя (при выполнении соответствующего оператора SQL) или по инициативе системы.

Одной из причин отката транзакции по инициативе системы является как раз нарушение целостности базы данных в результате действий данной транзакции (другие возможные условия отката транзакции по инициативе системы мы рассмотрим позже). Язык SQL System R (так мы будем называть вариант языка SQL, разработанный в проекте System R, чтобы отличать его от более поздних, «стандартных» вариантов этого языка) содержит средство установки так называемых *точек сохранения* (savepoint). При инициируемом пользователем откате транзакции можно указать номер точки сохранения, выше которого откат не распространяется. Инициируемый системой откат транзакции производится до ближайшей точки сохранения, в которой условие, вызвавшее откат, уже отсутствует. В частности, откат транзакции, инициированный по причине нарушения условия целостности, производится до ближайшей точки сохранения, в которой условия целостности соблюдены.

Естественно, что для реального выполнения отката транзакции необходимо запоминать некоторую информацию о выполнении транзакции. В System R для этих и других целей используется специальный набор данных – *журнал*, в который помещаются записи обо всех операциях всех транзакций, изменяющих состояние базы данных. При откате транзакции происходит процесс *обратного выполнения* транзакции (undo), в ходе которого в обратном порядке выполняются все изменения, запомненные в журнале.

В языке SQL System R имеется средство определения так называемых *триггеров* (trigger), позволяющих автоматически поддерживать целостность базы данных при модификациях ее объектов. В SQL System R триггер – это каталогизированная операция модификации, для которой задано условие ее автоматического выполнения. Особенно существенно наличие такого механизма в связи с наличием обсуждаемых ниже представлений базы данных, которыми может быть ограничен доступ к базе данных для ряда пользователей. Возможна ситуация, когда такие пользователи просто не могут соблюдать целостность базы данных без автоматического выполнения условных воздействий, поскольку они просто «не видят» всей базы данных и, в частности, не могут представить всех ограничений ее целостности.

Язык SQL содержит средства определения представлений. Представление – это каталогизированный именованный запрос на выборку данных (из одной или нескольких таблиц). Поскольку SQL – это «реляционный» язык, результатом выполнения любого запроса на выборку является таблица, и поэтому концептуально можно относиться к любому представлению как к таблице (при определении представления можно, в частности, присвоить имена полям этой таблицы). В языке допускается использование ранее определенных представлений практически везде, где допускается использование таблиц (с некоторыми ограничениями по поводу возможностей модификации через представления). Наличие возможности определять представления в совокупности с развитой системой авторизации позволяет ограничить доступ некоторых пользователей к базе данных выделенным набором представлений.

Авторизация доступа к базе данных также основана на средствах SQL. При создании любого объекта базы данных пользователь, выполняющий эту операцию, становится полновластным владельцем этого объекта, т.е. может выполнять по отношению к этому объекту любую допустимую операцию SQL. Далее этот пользователь может выполнить оператор SQL, означающий передачу всех его прав на этот объект (или их подмножества) любому другому пользователю. В частности, этому пользователю может быть передано право на передачу всех переданных ему прав (или их части) третьему пользователю и т.д. Одним из прав пользователя по отношению к объекту является право на изъятие у других пользователей всех или некоторых прав, которые ранее им были переданы. Эта операция распространяется транзитивно на всех дальнейших наследников этих прав.

Наличие в языке средств определения представлений и авторизации в принципе позволяет обойтись при эксплуатации System R без традиционного администратора баз данных, поскольку практически все системные действия производятся на основе средств SQL. Тем не менее, если организационно администратор баз данных требуется, то его работа достаточно упрощается за счет унифицированного набора средств управления. Кроме того, в System R каталоги баз данных поддерживаются также в виде таблиц, и к ним применены все запросы языка SQL. Заметим, что в более поздних SQL-ориентированных СУБД появился ряд дополнительных утилит, не связанных с языком SQL (например, утилиты сбора статистики или массовой загрузки базы данных), и в этих системах, видимо, без администратора базы данных не обойтись.

По части обеспечения параллельной работы многих пользователей с одной базой данных, основной подход System R состоит в том, что пользователь не обязан знать о наличии других пользователей, конкурирующих с ним за доступ к базе данных, т.е. система ответственна за обеспечение изолированности пользователей с гарантией отсутствия их взаимного влияния в пределах транзакций. Из этого следует, во-первых, что в интерфейсе пользователя с системой (т.е. в языке SQL) не должно быть средств регулирования взаимодействий с другими пользователями и, во-вторых, что система должна обеспечить автоматическую сериализацию набора транзакций, т.е. обеспечить режим выполнения этого набора транзакций, эквивалентный по конечному результату некоторому последовательному выполнению этих транзакций. Эта проблема решается в System R за счет автоматического выполнения синхронизационных блокировок всех изменяемых объектов базы данных.

Одним из основных требований к СУБД вообще и к System R в частности является обеспечение надежности баз данных по отношению к различного рода сбоям. К таким сбоям могут относиться программные ошибки прикладного и системного уровня, сбои процессора, поломки внешних носителей и т.д. В частности, к одному из видов сбоев можно отнести упоминавшиеся выше нарушения целостности базы данных и автоматический инициируемый системой откат транзакции – это системное средство восстановления базы данных после сбоев такого рода. Как уже отмечалось, такое восстановление происходит путем обратного выполнения транзакции на основе информации о внесенных ею изменениях, запомненной в журнале. На информации журнала также основано восстановление базы данных и после сбоев другого рода.

Что касается естественных требований к эффективности системы, то здесь основные решения связаны со спецификой физической организации баз данных во внешней памяти, использованием техники индексированного доступа к данным, буферизацией используемых страниц базы данных в основной памяти и развитой техникой оптимизации SQL-запросов, производимой на стадии их компиляции.

Структурная организация System R согласуется с поставленными при ее разработке целями и выбранными решениями. Основными структурными компонентами System R являются система управления реляционными данными (Relational Data System – RDS), состоящая, по существу, из компилятора языка SQL и подсистемы поддержки откомпилированных операторов, и система управления реляционной памятью (Relational Storage System – RSS).

RSS обеспечивает интерфейс довольно низкого, но достаточного для реализации SQL уровня для доступа к хранимым в базе данным (этот внутренний интерфейс System R напоминает внешний интерфейс систем, основанных на модели инвертированных таблиц, см. лекцию 2; более подробно он описывается ниже). Синхронизация транзакций, журнализация изменений и восстановление баз данных после сбоев также относятся к числу функций RSS.

Компилятор запросов использует интерфейс RSS для доступа к разнообразной справочной информации (каталоги таблиц, индексов, прав доступа, условий целостности, условных воздействий и т.д.) и производит рабочие программы, выполняемые в дальнейшем также с использованием интерфейса RSS.

Таким образом, система естественно разделяется на два уровня – уровень управления памятью и синхронизацией, фактически, не зависящий от базового языка запросов системы, и языковый уровень (уровень SQL), на котором решается большинство проблем System R. Заметим, что эта независимость скорее условная, чем абсолютная: язык SQL можно в принципе заменить каким-либо другим языком, но он должен обладать примерно такой же семантикой.

#### 12.2.3. Организация внешней памяти в базах данных System R

Как уже говорилось, база данных System R располагается в одном или нескольких сегментах внешней памяти. Каждый сегмент состоит из страниц данных и страниц индексной информации. Размер страницы данных в сегменте может быть выбран равным либо 4, либо 32 килобайтам; размер страницы индексной информации равен 512 байтам. Кроме того, при работе RSS поддерживается дополнительный набор данных для ведения журнала. Для повышения надежности журнала (а это наиболее критичная информация; при ее потере восстановление базы данных после сбоев невозможно) этот набор данных дублируется на двух внешних носителях.

###### Страницы данных и идентификаторы кортежей

В каждой странице данных хранятся кортежи одной или нескольких таблиц. Фундаментальным понятием RSS является *идентификатор кортежа (tuple identifier – tid)*. Гарантируется неизменяемость tid'а во все время существования кортежа в базе данных независимо от перемещений кортежа внутри страницы и даже при перемещении кортежа в другую страницу. Потребность в перемещении кортежей возникает по той причине, что кортеж, занесенный в некоторую таблицу базы данных, вообще говоря, во время своего существования может увеличиваться в размерах (если к этой таблице добавляется новое поле, или если в ней имеется хотя бы одно поле, типом данных которого являются строки символов переменного размера). Реально tid представляет собой пару <номер страницы, индекс описателя кортежа в странице>. При этом кортеж может реально располагаться в данной странице (рис. 12.1a) или в другой странице (рис. 12.1b).


*Рис. 12.1.* Идентификатор кортежа и расположение кортежа в странице данных

Как показывает рис. 12.1, в каждой странице данных имеются две области: область хранения описателей кортежей и область хранения самих кортежей. Один из остроумных приемов, примененных в System R, состоит в том, что обе эти области являются динамическими, т.е. в странице данных заранее не резервируется место под описатели кортежей. Легко видеть, что выделение фиксированной части страницы данных под описатели кортежей (вмещающей, скажем, *k* описателей) потенциально привело бы к потери памяти в этой странице, поскольку при размещении в ней *k* кортежей очень маленького размера пропадало бы место в области хранения кортежей, а при размещении *p<k* крупных кортежей полностью заполнялась бы область хранения кортежей, но пропадало бы место в области описателей. Для динамического распределения памяти внутри страницы память на описатели кортежей выделяется вниз от начала страницы, а память для хранения кортежей – вверх от конца страницы.

Второй вариант хранения кортежей возникает в том случае, когда некоторый кортеж после своего создания был размещен системой в странице с номером N, а после обновления с увеличением размера перестал помещаться в этой странице, и система была вынуждена разместить его в странице с номером M. Тогда исходный tid этого кортежа не изменится, но его описатель в странице N будет содержать не координаты кортежа в данной странице, а новый tid, указывающий на реальное положение кортежа в странице M. Легко видеть, что применение такого подхода позволяет ограничиться максимум одним уровнем косвенности (если данный кортеж в какой-то момент времени перестанет помещаться в странице M, и система переместит его в страницу P, то достаточно будет изменить косвенную ссылку на этот кортеж в странице N, и его исходный tid не изменится).

Поскольку допускается нахождение в одной странице данных кортежей разных таблиц, каждый кортеж должен, кроме содержательной части, включать служебную информацию, идентифицирующую таблицу, которой принадлежит данный кортеж. Кроме того, в System R (точнее, в языке SQL) допускается динамическое добавление полей к существующим таблицам. При этом реально происходит лишь модификация описателя таблицы в таблице-каталоге таблиц. В существующем кортеже таблицы новое поле возникает только при модификации этого кортежа, затрагивающей новое поле. Это позволяет избежать массовой перестройки хранимой таблицы при добавлении к ней новых полей, но, естественно, требует хранения при кортеже дополнительной служебной информации, определяющей реальное число полей в данном кортеже. (Заметим, что удалять существующие поля существующей таблицы в SQL System R не разрешалось.)

###### Индексы и кластеризация таблиц

На основе наличия уникальных, обеспечивающих почти прямой доступ к кортежам и не изменяемых во время существования кортежей tid'ов в System R поддерживаются дополнительные управляющие структуры – *индексы*. Каждый индекс определяется на одном или нескольких полях таблицы, значения которых составляют его ключ, и позволяет производить прямой поиск по ключу кортежей (их tid'ов) и последовательное сканирование таблицы по индексу, начиная с указанного ключа, в порядке возрастания или убывания значений ключа. Некоторые индексы при их создании могут обладать атрибутом уникальности. В таком индексе не допускаются дубликаты ключа. Это единственное средство SQL System R указания системе первичного ключа таблицы (фактически, набора первичного и всех возможных ключей таблицы).

Для организации индексов в System R применяется техника *B+-деревьев* (более подробно B+-деревья рассматриваются в подразделе [12.3.2. Индексы](http://citforum.ru/database/advanced_intro/38.shtml#12.3.2)). Каждый индекс занимает отдельный набор страниц, номер корневой страницы запоминается в описателе индекса. Использование B+-деревьев позволяет достичь эффективности при прямом поиске, поскольку они из-за своей *сильной ветвистости* обладают небольшой глубиной. Кроме того, B+-деревья сохраняют *порядок ключей* в листовых блоках иерархии, что позволяет производить последовательное сканирование таблицы в порядке возрастания или убывания значений полей, на которых определен индекс. Фундаментальное свойство B+-деревьев – *автоматическая балансировка* дерева – допускает произведение лишь локальных модификаций индекса при переполнениях и опустошениях страниц индекса. Насколько известно автору, System R была первой системой, в которой для организации индексов использовались B+-деревья. Эту традицию соблюдает большинство реляционных систем, возникших позднее.

Видимо, наиболее важной особенностью физической организации баз данных в System R является возможность обеспечения *кластеризации* связанных кортежей одной или нескольких таблиц. Под кластеризацией кортежей понимается физически близкое расположение (в пределах одной страницы данных) логически связанных кортежей. Обеспечение соответствующей кластеризации позволяет добиться высокой эффективности системы при выполнении некоторого класса запросов. В силу большой важности понятия кластеризации в System R и ее развитиях рассмотрим историю вопроса более подробно.

В окончательном варианте System R существует только одно средство определения условий кластеризации таблицы – объявить до заполнения таблицы один (и только один) индекс, определенный на полях этой таблицы, кластеризованным. Тогда, если заполнение таблицы кортежами производится в порядке возрастания или убывания значений полей кластеризации (в зависимости от атрибутики индекса), система физически располагает кортежи в страницах данных в том же порядке.

Кроме того, в каждой странице данных кластеризованной таблицы оставляется некоторое резервное свободное пространство. При последующих вставках кортежей в такую таблицу система стремится поместить каждый кортеж в одну из страниц данных, в которых уже находятся кортежи этой таблицы с такими же (или близкими) значениями полей кластеризации. Естественно, что поддерживать идеальную кластеризацию таблицы можно только до определенного предела, пока не исчерпается резервная память в страницах. Далее этого предела степень кластеризации таблицы начинает уменьшаться, и для восстановления идеальной кластеризации таблицы требуется физическая реорганизация таблицы (ее можно произвести средствами SQL).

Очевидным преимуществом кластеризации таблицы является то, что при последовательном сканировании кластеризованной таблицы с использованием кластеризованного индекса потребуется ровно столько чтений страниц данных из внешней памяти, сколько страниц занимают кортежи этой таблицы. Следовательно, при правильно выбранных критериях кластеризации запросы, связанные с заданием условий на полях кластеризации можно выполнить почти оптимально.

В ранних версиях System R существовал еще один способ физического доступа к кортежам таблицы и, соответственно, еще один способ указания условия кластеризации с использованием так называемых *связей (links)*. На уровне физического представления связь – это физическая ссылка (tid) из одного кортежа на другой (не обязательно одной таблицы). В языке SEQUEL (до того момента, когда его стали называть SQL) существовали средства определения связей в иерархической манере: можно было объявить некоторую таблицу родительской по отношению к той же или другой таблице-потомку. При этом указывались поля родительской таблицы и таблицы-потомка, в соответствии со значениями которых образовывалась иерархия. Правила построения были очень простыми – проводились связи от кортежа родительской таблицы ко всем кортежам таблицы-потомка с теми же значениями полей связывания. На самом деле, все кортежи таблицы-потомка с общим значением полей связывания образовывали кольцевой список, на который проводилась одна связь из соответствующего кортежа родительской таблицы.

Следует заметить, что этот способ использования механизма связей поддерживался в ранних версиях SEQUEL. В интерфейсе RSS System R этого периода допускалась возможность произвольной установки связей без учета совпадения значений полей связывания. Тем самым, в системе в целом не использовались все возможности RSS, которые с избытком превосходили потребности организации иерархических бинарных связей по совпадению полей связывания.

Для одной таблицы допускалось создание многих связей: кортеж таблицы мог быть родителем нескольких иерархий и входить в несколько других иерархий в качестве потомка. При этом одна связь могла быть объявлена кластеризованной. Тогда система стремилась поместить в одну страницу данных все кортежи одной иерархии. При этом, естественно, использовалась возможность размещения в одной странице данных кортежей нескольких таблиц. Основной смысл такой кластеризации заключался в возможности оптимизации выполнения некоторых запросов, включающих (экви)соединение двух связанных таблиц в соответствии со значениями полей связывания.

В более поздних публикациях, посвященных System R, упоминания о механизме связей исчезли, из чего можно заключить, что разработчики отказались от его использования. Думается, что основными причинами отказа от использования связей были следующие. Во-первых, средства построения связей, обеспечиваемые RSS, были очень низкого уровня, гораздо более низкого, чем средства поддержки индексов. Если при занесении, удалении или обновлении кортежа RSS обеспечивала автоматическую коррекцию всех индексов, то для коррекции связей требовалось выполнить ряд дополнительных обращений к RSS, из-за чего время выполнения этих операций, конечно, увеличивалось.

Во-вторых, при реализации этого механизма возникают дополнительные синхронизационные проблемы нижнего уровня (уровня совместного доступа к страницам данных). В частности, наличие прямых ссылок между страницами данных увеличивает вероятность возникновения синхронизационных тупиков.

Наконец, в-третьих, все эти дополнительные накладные расходы не окупались преимуществами, предоставляемыми механизмом связей. Действительно, максимального эффекта от использования связей можно достичь только при выполнении операции соединения двух таблиц, кластеризованных по этой связи, если поле соединения совпадает с полем связывания и условия, накладываемые на родительскую таблицу, выделяют в нем ровно один кортеж. Очевидно, что такие запросы на практике редки.

(Отметим, что приведенные соображения принадлежат автору и не излагались в публикациях по System R, так что на самом деле причины могли быть и другими.)

Кроме таблиц и индексов при работе System R во внешней памяти могут располагаться еще и временные объекты – *списки (list)*. Список – это временная структура данных, создаваемая с целью оптимизации выполнения SQL-запроса, содержащая некоторые кортежи хранимой таблицы базы данных, не имеющая имени и, следовательно, не видимая на уровне интерфейса SQL. Кортежи списка могут быть упорядочены по возрастанию или убыванию полей соответствующей таблицы. Средства работы со списками имеются в интерфейсе RSS, но их, естественно, нет в SQL. Соответственно, эти средства используются только внутри системы при выполнении запросов (в частности, один из наиболее эффективных алгоритмов выполнения соединений основан на использовании отсортированных списков кортежей). Публикации по System R не дают точного представления о структурах данных, используемых при организации списков, но исходя из здравого смысла можно предположить, что они устроены не так, как таблицы (например, для кортежа, входящего в список, не требуется адресация через tid), и что располагаются они во временных файлах (в случае сбоя системы все временные объекты пропадают).

#### 12.2.4. Интерфейс RSS

Следует заметить, что описываемый в этом подразделе интерфейс RSS не соответствует в точности ни одной из публикаций, посвященных System R, а является скорее некоторой компиляцией, согласующейся с завершающими публикациями.

На уровне RSS отсутствует именование объектов базы данных, употребляемое на уровне SQL. Вместо имен объектов используются их уникальные идентификаторы, являющиеся прямыми или косвенными адресами внутренних описателей объектов во внешней памяти для постоянных объектов или в основной памяти для временных объектов. Замена имен объектов базы данных на их идентификаторы производится компилятором SQL на основе информации, черпаемой им из системных таблиц-каталогов.

Можно выделить следующие группы операций:

* операции сканирования таблиц и списков;
* операции создания и уничтожения постоянных и временных объектов базы данных;
* операции модификации таблиц и списков;
* операция добавления поля к таблице;
* операции управления прохождением транзакций;
* операция явной синхронизации.

###### Операции сканирования таблиц и списков

Операции группы сканирования позволяют последовательно, в порядке, определяемом типом сканирования, прочитать кортежи таблицы или списка, удовлетворяющие требуемым условиям. Группа включает операции OPEN, NEXT и CLOSE, означающие, соответственно, начало сканирования, требование чтения следующего кортежа, удовлетворяющего условиям, и конец сканирования.

Для таблицы возможны два режима сканирования: прямое сканирование и сканирование через индекс. При прямом сканировании единственным параметром операции OPEN является идентификатор таблицы (включающий и идентификатор сегмента, в котором эта таблица хранится). По причине того, что в System R допускается размещение в одной странице данных кортежей нескольких таблиц, прямое сканирование предполагает последовательный просмотр всех страниц сегмента с выделением в них кортежей, входящих в данную таблицу; это очень дорогой способ сканирования таблицы. При этом порядок выборки кортежей определяется их физическим размещением в страницах сегмента, т.е. предопределен системой.

При начале сканирования таблицы через индекс в число параметров операции OPEN входит идентификатор какого-либо индекса, определенного ранее на полях этой таблицы. Кроме того, можно указать диапазон сканирования в терминах значений поля (полей), составляющего ключ индекса. При открытии сканирования через индекс производится начальная установка указателя сканирования в позицию листа B-дерева (см. подраздел [12.3.2. Индексы](http://citforum.ru/database/advanced_intro/38.shtml#12.3.2)) индекса, соответствующую левой границе заданного диапазона. Процесс сканирования состоит в последовательном продвижении по листовым вершинам индекса до достижения правой границы диапазона сканирования с выборкой идентификаторов кортежей и чтением соответствующих кортежей. Легко видеть, что в худшем случае может потребоваться столько чтений страниц данных из внешней памяти, сколько идентификаторов кортежей было встречено, т.е. эффективность сканирования по индексу определяется «широтой» заданного диапазона сканирования. При этом, конечно, имеется то преимущество, что порядок сканирования соответствует порядку возрастания или убывания значений ключа индекса.

Наконец, при сканировании списка, как и при прямом сканировании таблицы, единственным параметром операции OPEN является идентификатор списка, но, в отличие от прямого сканирования таблицы это сканирование максимально эффективно: читаются только страницы, содержащие кортежи из данного списка, и порядок сканирования совпадает с порядком занесения кортежей в список или порядком списка, если он упорядочен.

В результате успешного выполнения операции открытия сканирования (если нет ошибок в параметрах) вырабатывается и возвращается идентификатор сканирования, который используется в качестве аргумента других операций этой группы.

Операция NEXT выполняет чтение следующего кортежа указанного сканирования, удовлетворяющего условию данной операции. Условие представляет собой дизъюнктивную нормальную форму простых условий, накладываемых на значения указанных полей таблицы. Простое условие – это условие вида номер-поля op константа, где op – операция сравнения <, <=, >, >=, = или !=. Общее условие является параметром операции NEXT.

Семантика операции NEXT следующая: начиная с текущей позиции сканирования выбираются кортежи таблицы в порядке, определяемом типом сканирования, до тех пор, пока не встретится кортеж, значения полей которого удовлетворяют указанному условию. Этот кортеж и является результатом операции. Если при выборке кортежа достигается правая граница диапазона сканирования (правая граница значения ключа при сканировании через индекс или последний кортеж таблицы или списка при прямом сканировании), то вырабатывается особый признак результата. После этого единственным разумным действием является закрытие сканирования – операция CLOSE

Операция CLOSE может быть выполнена в данной транзакции по отношению к любому ранее открытому сканированию независимо от его состояния (т.е. независимо от того, достигнута ли при сканировании правая граница диапазона сканирования). Параметром операции является идентификатор сканирования, и ее выполнение приводит к тому, что этот идентификатор становится недействительным (и, соответственно, уничтожаются служебные структуры памяти RSS, относящиеся к данному сканированию).

###### Операции создания и уничтожения постоянных и временных объектов базы данных

Группа операций создания и уничтожения постоянных и временных объектов базы данных включает операции создания таблиц (CREATE TABLE), списков (CREATE LIST), индексов (CREATE IMAGE) и уничтожения любого из подобных объектов (DROP TABLE, DROP LIST и DROP IMAGE). Входным параметром операций создания таблиц и списков является спецификатор структуры объекта, т.е. число полей объекта и спецификаторы их типов. Кроме того, при спецификации полей таблицы указывается разрешение или запрещение наличия неопределенных значений полей в кортежах этой таблицы или списка. Неопределенные значения кодируются специальным образом. Любая операция сравнения константы данного типа с неопределенным значением по определению вырабатывает значение *false*, кроме операции сравнения на совпадение со специальной литеральной константой NULL.

В результате выполнения этих операций заводится описатель в служебной таблице описателей таблиц или основной памяти (в зависимости от того, создается ли постоянный объект или временный), и вырабатывается идентификатор объекта, который служит входным параметром других операций, относящихся к соответствующему объекту (в частности, параметром операции OPEN при открытии сканирования объекта).

Входными параметрами операции CREATE IMAGE являются идентификатор таблицы, для которой создается индекс, список номеров полей, значения которых составляют ключ индекса, и признаки упорядочения по возрастанию или убыванию для всех полей, составляющих ключ. Кроме того, может быть указан признак уникальности индекса, т.е. запрещения наличия в данном индексе ключей-дубликатов. Если операция выполняется по отношению к пустой в этот момент таблице, то выполнение операции такое же простое, как и для операций создания таблиц и списков: создается описатель в служебной таблице описателей индексов и возвращается идентификатор индекса (который, в частности, используется в качестве аргумента операции открытия сканирования таблицы через индекс).

Если же к моменту создания индекса соответствующая таблица не пуста (а это допускается), то операция становится существенно более дорогостоящей, поскольку при ее выполнении происходит реальное создание B-дерева индекса, что требует, по меньшей мере, одного последовательного просмотра таблицы. При этом, если создаваемый индекс имеет признак уникальности, то это контролируется при создании B-дерева, и если уникальность нарушается, то операция не выполняется (т.е. индекс не создается). Из этого следует, что хотя создание индексов в динамике не запрещается, более эффективно создавать все индексы на данной таблице до ее заполнения. Заметим, что создание кластеризованного индекса для непустой таблицы запрещено, поскольку соответствующую кластеризацию таблицы без ее реструктуризации получить невозможно.

Операции DROP TABLE, DROP LIST и DROP IMAGE могут быть выполнены в любой момент независимо от состояния объектов. Выполнение операции приводит к уничтожению соответствующего объекта и, вследствие этого, недействительности его идентификатора.

Следует отметить, что массовые операции над постоянными объектами (CREATE IMAGE и DROP TABLE) требуют дополнительных накладных расходов в связи с необходимостью обеспечения возможности откатов транзакции, для чего требуется выполнение массовых обратных действий. Особенно сильно это затрагивает операцию уничтожения непустых таблиц, поскольку требует журнализации всех кортежей, содержащихся в них к моменту уничтожения. Поэтому, хотя уничтожение непустых таблиц и не запрещено, нужно иметь в виду, что это очень дорогостоящая операция.

###### Операции модификации таблиц и списков

Группа операций модификации таблиц и списков включает операции вставки кортежа в таблицу или список (INSERT), удаления кортежа из таблицы (DELETE) и обновления кортежа в таблице (UPDATE).

Параметрами операции вставки кортежа являются идентификатор таблицы или списка и набор значений полей кортежа. Среди значений полей могут быть литеральные неопределенные значения NULL. Естественно, при выполнении операции контролируется допустимость неопределенных значений в соответствующих полях. При занесении кортежа в кластеризованную таблицу поиск места в сегменте под кортеж производится с использованием кластеризованного индекса: система пытается вставить кортеж в страницу данных, уже содержащую кортежи с теми же или близкими значениями полей кластеризации. При занесении кортежа в некластеризованную таблицу место под кортеж выделяется в первой подходящей странице данных. Наконец, при вставке кортежа в список он помещается в конец списка.

При занесении кортежа в таблицу производится коррекция всех индексов, определенных на этой таблице. Реально это выражается во вставке новой записи во все B-деревья индексов. При этом могут произойти переполнения одной или нескольких страниц индекса, что вызовет переливание части записей в соседние страницы или расщепление страниц. Если индекс определен с атрибутом уникальности, то проверяется соблюдение этого условия, и если оно нарушено, операция вставки считается невыполненной. Из этого видно, что операция вставки кортежа тем более накладна, чем больше индексов определено для данной таблицы (это относится и к операциям удаления и модификации кортежей).

В результате успешного выполнения операции вставки кортежа в таблицу вырабатывается идентификатор нового кортежа, который выдается в качестве результата операции и может быть в дальнейшем использован как прямой параметр операций удаления и модификации кортежей таблицы. При занесении кортежа в список значение идентификатора кортежа не вырабатывается (для списков допускается только последовательное сканирование и добавление новых кортежей в конец списка; над ними нельзя определить индексов, и поэтому косвенная адресация кортежей списков через их идентификаторы не требуется).

Операции удаления и модификации кортежей допускаются только для кортежей таблиц. Естественно, что для выполнения этих операций необходимо идентифицировать соответствующий кортеж. В интерфейсе RSS допускаются два способа такой идентификации: с помощью идентификатора кортежа (явная адресация) и с использованием идентификатора открытого к этому времени сканирования. Первый вариант возможен, поскольку идентификатор кортежа сообщается как ответный параметр операции занесения кортежа в постоянную таблицу. При идентификации кортежа с помощью идентификатора сканирования имеется в виду кортеж, прочитанный с помощью последней операции NEXT. Если при такой идентификации выполняется операция DELETE или операция UPDATE, задевающая порядок сканирования (т.е. сканирование ведется по индексу и операция модификации меняет поле кортежа, входящее в состав ключа этого индекса), то текущий кортеж сканирования теряется, и его идентификатор нельзя использовать для идентификации кортежа, пока не будет выполнена следующая операция NEXT.

Единственным параметром операции DELETE является идентификатор кортежа или идентификатор сканирования. Параметры операции UPDATE включают, кроме этого, спецификацию изменяемых полей кортежа (список номеров полей и их новых значений). Среди значений могут находиться литеральные изображения неопределенных значений, если соответствующие поля таблицы допускают хранение неопределенных значений. При выполнении операции DELETE производится коррекция всех индексов, определенных на данной таблице. Операция UPDATE также может повлечь коррекцию индексов, если затрагивает поля, входящие в состав их ключей.

Кроме описанных «атомарных» операций сканирования и модификации таблиц и списков, интерфейс RSS включает одну «макрооперацию» BUILDLIST, позволяющую за одно обращение к RSS построить список, отсортированный в соответствии со значениями заданных полей. Эта операция включает сканирование заданной таблицы или списка, создание нового списка, в который включаются указанные поля выбираемых кортежей, и сортировку построенного списка в соответствии со значениями указанных полей. Идентификатор заново построенного отсортированного списка является ответным параметром операции.

Соответственно, параметрами операции BUILDLIST являются набор параметров для открытия сканирования (допускается любой способ сканирования), список номеров полей, составляющих кортежи нового списка, и список номеров полей, по которым нужно производить сортировку (как и в случае создания нового индекса, можно отдельно для каждого из этих полей указать требование к сортировке по возрастанию или убыванию значений данного поля). Отдельным параметром операции BUILDLIST является признак, в соответствии со значением которого в новом списке допускаются или не допускаются кортежи-дубликаты.

###### Операция добавления поля к существующей таблице

Операция RSS добавления поля к существующей таблице позволяет в динамике изменять схему таблицы. Параметрами операции CHANGE являются идентификатор существующей таблицы и спецификация нового поля (его тип). При выполнении операции изменяется только описатель данной таблицы в служебной таблице описателей таблиц. До выполнения первой операции UPDATE, затрагивающей новое поле таблицы, реально ни в одном кортеже таблицы память под новое поле выделяться не будет. По умолчанию значения нового поля во всех кортежах таблицы, в которые еще не производилось явное занесение значения, считаются неопределенными. Тем самым, ни для одного поля, динамически добавленного к существующей таблице, не может быть запрещено хранение неопределенных значений.

###### Операции управления прохождением транзакций

Каждая операция RSS выполняется в пределах некоторой транзакции. Интерфейс RSS включает набор операций управления прохождением транзакции: начать транзакцию (BEGIN TRANSACTION), закончить транзакцию (END TRANSACTION), установить точку сохранения (SAVE) и выполнить откат до указанной точки сохранения или до начала транзакции (RESTORE).

Это не отмечалось раньше, но на самом деле при вызове любой операции функции RSS, кроме BEGIN TRANSACTION, должен указываться еще один параметр – идентификатор транзакции. Этот идентификатор и вырабатывается при выполнении операции BEGIN TRANSACTION, которая сама входных параметров не требует.

В любой точке транзакции до выполнения операции END TRANSACTION может быть выполнен откат данной транзакции, т.е. обратное выполнение всех изменений, произведенных в данной транзакции, и восстановление состояния позиций сканирования. Откат может быть произведен до начала транзакции (в этом случае о восстановлении позиций сканирования говорить бессмысленно) или до установленной ранее в транзакции точки сохранения.

Точка сохранения устанавливается с помощью операции SAVE. При выполнении этой операции запоминаются состояние сканов данной транзакции, открытых к моменту выполнения SAVE, и координаты последней записи об изменениях в базе данных в журнале, произведенной от имени данной транзакции. Ответным параметром операции SAVE (а прямых параметров, кроме идентификатора транзакции, она не требует) является идентификатор точки сохранения. Этот идентификатор в дальнейшем может быть использован как аргумент операции RESTORE, при выполнении которой производится восстановление базы данных по журналу (с использованием записей о ее изменениях от данной транзакции) до того состояния, в котором находилась база данных к моменту установки указанной точки сохранения. Кроме того, по локальной информации в оперативной памяти, привязанной к транзакции, восстанавливается состояние ее сканов. Откат к началу транзакции инициируется также вызовом операции RESTORE, но с указанием некоторого предопределенного идентификатора точки сохранения.

При выполнении своих транзакций пользователи System R изолированы один от другого, т.е. не ощущают того, что система функционирует в многопользовательском режиме. Это достигается за счет наличия в RSS механизма неявной синхронизации. До конца транзакции никакие изменения базы данных, произведенные в пределах этой транзакции, не могут быть использованы в других транзакциях (попытка использования таких данных приводит к вр*е*менным синхронизационным блокировкам этих транзакций). При выполнении операции END TRANSACTION происходит "фиксация" изменений, произведенных в данной транзакции, т.е. они становятся видимыми в других транзакциях. Реально это означает снятие синхронизационных блокировок с объектов базы данных, изменявшихся в транзакции. Из этого следует, что после выполнения END TRANSACTION невозможны индивидуальные откаты данной транзакции. RSS просто делает недействительным идентификатор данной транзакции, и после выполнения операции окончания транзакции отвергает все операции с таким идентификатором.

###### Операция явной синхронизации

Последняя операция интерфейса RSS – операция явной синхронизации LOCK. Эта операция позволяет установить явную синхронизационную блокировку указанной таблицы (параметром операции является идентификатор таблицы). Выполнение операции LOCK гарантирует, что никакая другая транзакция до конца данной не сможет изменить эту таблицу (вставить в нее новый кортеж, удалить или модифицировать существующий), если установлена блокировка в режиме чтения, или даже прочитать любой кортеж этой таблицы, если установлена монопольная блокировка.

Из всего, что говорилось раньше по поводу подхода к синхронизации в System R и соответствующего разбиения системы на уровни, следует нелогичность наличия этой операции в интерфейсе RSS. На самом деле, логически эта операция избыточна, т.е. если бы ее не было, можно было бы реализовать SQL с использованием оставшейся части операций. Предварительно (до подробного обсуждения средств управления транзакциями в лекции 13) заметим, что операция LOCK введена в интерфейс RSS для возможности оптимизации выполнения запросов.

Дело в том, что, как видно из описания интерфейса RSS, этот интерфейс является покортежным. Следовательно, и информация для синхронизации носит достаточно узкий характер. В то же время, на уровне SQL имеется более полная информация. Например, если обрабатывается предложение SQL DELETE FROM table\_name, то известно, что будут удалены все кортежи указанной таблицы. Понятно, что как бы не реализовывался механизм синхронизации в RSS, в данном случае выгоднее сообщить сразу, что изменения касаются всей таблицы.

Но ситуации, в которых очевидна выгода от использования явной синхронизации, достаточно редки. Пользоваться этим средством можно только очень осмотрительно, потому что неоправданные захваты таких крупных объектов могут резко ограничить степень асинхронности выполнения транзакций.

### 12.3. Общие принципы организации данных во внешней памяти в SQL-ориентированных СУБД

В этом разделе кратко обсуждаются основные подходы к организации данных во внешней памяти, принятые в современных SQL-ориентированных СУБД. В большинстве случаев они основаны на идеях, заложенных в System R, хотя, конечно, в любой развитой системе имеются собственные приемы, которые здесь обсуждаться не будут.

SQL-ориентированные СУБД обладают рядом особенностей, влияющих на организацию внешней памяти. Наиболее важным автору кажутся следующие особенности:

* Наличие двух уровней системы: уровня непосредственного управления данными во внешней памяти (а также обычно управления буферами оперативной памяти, управления транзакциями и журнализацией изменений БД) и языкового уровня (уровня, реализующего язык SQL). При такой организации подсистема нижнего уровня должна поддерживать во внешней памяти набор базовых структур, конкретная интерпретация которых входит в число функций подсистемы верхнего уровня.
* Поддержка таблиц-каталогов. Информация, связанная с именованием объектов базы данных и их конкретными свойствами (например, структура ключа индекса), поддерживается подсистемой языкового уровня. С точки зрения структур внешней памяти таблица-каталог ничем не отличается от обычной таблицы базы данных.
* Регулярность структур данных. Поскольку основным объектом модели данных SQL является плоская таблица, основной набор объектов внешней памяти может иметь очень простую регулярную структуру.
* Необходимость обеспечения возможности эффективного выполнения операторов языкового уровня как над одной таблицей (простые селекция и проекция), так и над несколькими таблицами (наиболее распространено и трудоемко соединение нескольких таблиц). Для этого во внешней памяти должны поддерживаться дополнительные «управляющие» структуры – индексы.
* Наконец, для выполнения требования надежного хранения баз данных необходимо поддерживать избыточность хранения данных, что обычно реализуется в виде журнала изменений базы данных.

Соответственно возникают следующие разновидности объектов во внешней памяти базы данных:

* строки таблиц – основная часть базы данных, большей частью непосредственно видимая пользователям;
* управляющие структуры – индексы, создаваемые по инициативе пользователя (администратора) или верхнего уровня системы из соображений повышения эффективности выполнения запросов и обычно автоматически поддерживаемые нижним уровнем системы;
* журнальная информация, поддерживаемая для удовлетворения потребности в надежном хранении данных;
* служебная информация, поддерживаемая для удовлетворения внутренних потребностей нижнего уровня системы (например, информация о свободной памяти).

#### 12.3.1. Хранение таблиц

Существуют два принципиальных подхода к физическому хранению таблиц. Наиболее распространенным является покортежное хранение таблиц (единицей физического хранения является кортеж). Естественно, это обеспечивает быстрый доступ к целому кортежу, но при этом во внешней памяти дублируются общие значения разных кортежей одной таблицы и, вообще говоря, могут потребоваться лишние обмены с внешней памятью, если нужна часть кортежа.

Альтернативным (менее распространенным) подходом является хранение таблицы по столбцам, т.е. единицей хранения является столбец таблицы с исключенными дубликатами. Естественно, что при такой организации суммарно в среднем тратится меньше внешней памяти, поскольку дубликаты значений не хранятся; за один обмен с внешней памятью в общем случае считывается больше полезной информации. Дополнительным преимуществом является возможность использования значений столбца таблицы для оптимизации выполнения операций соединения. Но при этом требуются существенные дополнительные действия для сборки целого кортежа (или его части).

Поскольку гораздо более распространено хранение по строкам, рассмотрим немного более подробно этот способ хранения таблиц (в дополнение к тому, что говорилось в разделе [12.2 Основные понятия, цели и общая организация System R](http://citforum.ru/database/advanced_intro/35.shtml#12.2)). Типовой, унаследованной от System R, структурой страницы данных является та, которая показана на рис. 12.1.

Эту организацию хранения кортежей можно в целом охарактеризовать следующим образом:

* Каждый кортеж обладает уникальным идентификатором (tid), не изменяемым во все время существования кортежа и позволяющим выбрать кортеж в основную память не более чем за два обращения к внешней памяти. Структура tid следует из рис. 12.1.
* Обычно каждый кортеж хранится целиком в одной странице. Из этого следует, что максимальная длина кортежа любой таблицы ограничена размерами страницы. Возникает вопрос: как быть с «длинными» данными, которые в принципе не помещаются в одной странице? Применяется несколько методов. Наиболее простым решением является хранение таких данных в отдельных (вне базы данных) файлах с заменой «длинного» данного в кортеже на имя соответствующего файла. В некоторых системах такие данные хранились внутри базы данных в отдельном наборе страниц внешней памяти, связанном физическими ссылками. Оба эти решения сильно ограничивают возможность работы с длинными данными (как, например, удалить несколько байт из середины 2-мегабайтной строки?). В настоящее время все чаще используется метод, предложенный много лет тому назад в проекте Exodus [[3.13]](http://citforum.ru/database/advanced_intro/86.shtml#ref.3.13), когда «длинные» данные организуются в виде B-деревьев последовательностей байт.
* Как правило, в одной странице данных хранятся кортежи только одной таблицы. Существуют, однако, варианты с возможностью хранения в одной странице кортежей нескольких таблиц. Это вызывает некоторые дополнительные расходы по части служебной информации (при каждом кортеже нужно хранить информацию о соответствующей таблице), но зато иногда позволяет резко сократить число обменов с внешней памятью при выполнении соединений.
* Изменение схемы хранимой таблицы с добавлением нового поля не вызывает потребности в физической реорганизации таблицы. Достаточно лишь изменить информацию в описателе таблицы и расширять кортежи только при занесении информации в новое поле.
* Поскольку таблицы могут содержать неопределенные значения, необходима соответствующая поддержка на уровне хранения. Обычно это достигается путем хранения соответствующей шкалы при каждом кортеже, который в принципе может содержать неопределенные значения.
* Проблема распределения памяти в страницах данных связана с проблемами синхронизации и журнализации и не всегда тривиальна. Например, если в ходе выполнения транзакции некоторая страница данных опустошается, то ее нельзя перевести в статус свободных страниц до конца транзакции, поскольку при откате транзакции удаленные при прямом выполнении транзакции и восстановленные при ее откате кортежи должны получить те же самые идентификаторы.
* Распространенным способом повышения эффективности СУБД является кластеризация таблицы по значениям одного или нескольких столбцов. Полезной для оптимизации соединений является совместная кластеризация нескольких таблиц.
* С целью использования возможностей распараллеливания обменов с внешней памятью иногда применяют схему декластеризованного хранения таблиц: кортежи с общим значением столбца декластеризации размещают на разных дисковых устройствах, обмены с которыми можно выполнять параллельно.

Что же касается хранения таблицы по столбцам, то основная идея состоит в совместном хранении всех значений одного (или нескольких) столбцов. Для каждого кортежа таблицы хранится кортеж той же степени, состоящий из ссылок на места расположения соответствующих значений столбцов.

#### 12.3.2. Индексы

Как бы не были организованы индексы в конкретной СУБД, их основное назначение состоит в обеспечении эффективного прямого доступа к кортежу таблицы по ключу. Обычно индекс определяется для одной таблицы, и ключом является значение ее поля (возможно, составного). Если ключом индекса является возможный ключ таблицы, то индекс должен обладать свойством уникальности, т.е. не содержать дубликатов ключа. На практике ситуация выглядит обычно противоположно: при объявлении первичного ключа таблицы автоматически заводится уникальный индекс, а единственным способом объявления возможного ключа, отличного от первичного, является явное создание уникального индекса. Это связано с тем, что для проверки сохранения свойства уникальности возможного ключа, так или иначе, требуется индексная поддержка.

Поскольку при выполнении многих операций уровня SQL требуется сортировка кортежей таблиц в соответствии со значениями некоторых полей, полезным свойством индекса является обеспечение последовательного просмотра кортежей таблицы в заданном диапазоне значений ключа в порядке возрастания или убывания значений ключа.

Наконец, одним из способов оптимизации выполнения эквисоединения таблиц (наиболее распространенная из числа дорогостоящих операций) является организация так называемых мультииндексов для нескольких таблиц, обладающих общими атрибутами. Любой из этих атрибутов (или их набор) может выступать в качестве ключа мультииндекса. Значению ключа сопоставляется набор кортежей всех связанных мультииндексом таблиц, значения выделенных атрибутов которых совпадают со значением ключа.

Общей идеей любой организации индекса, поддерживающего прямой доступ по ключу и последовательный просмотр в порядке возрастания или убывания значений ключа является хранение упорядоченного списка значений ключа с привязкой к каждому значению ключа списка идентификаторов кортежей. Одна организация индекса отличается от другой, главным образом, в способе поиска ключа с заданным значением.

###### B+-деревья

Наиболее популярным подходом к организации индексов в базах данных является использование техники B+-деревьев. Техника B- и B+-деревьев была предложена в начале 1970-х гг. Рудольфом Байером (Rudolf Bayer) и Эдом Маккрейтом (Ed McCreight) [[3.17]](http://citforum.ru/database/advanced_intro/86.shtml#ref.3.17). С точки зрения внешнего логического представления B-дерево – это сбалансированное сильно ветвистое дерево во внешней памяти. Сбалансированность означает, что длина пути от корня дерева к любому его листу одна и та же. Ветвистость дерева – это свойство каждого узла дерева ссылаться на большое число узлов-потомков. С точки зрения физической организации B-дерево представляется как мультисписочная структура страниц внешней памяти, т.е. каждому узлу дерева соответствует блок внешней памяти (страница). В B+-дереве внутренние и листовые страницы обычно имеют разную структуру.

Типовая структура внутренней страницы B+-дерева показана на рис. 12.2.


*Рис. 12.2.* Типовая структура внутренней страницы B+-дерева

При этом выдерживаются следующие свойства:

* ключ1 ключ2 ... ключm;
* в странице дерева Nm находятся ключи k со значениями ключm <= k <= ключm+1.

Листовая страница обычно имеет следующую структуру, показанную на рис. 12.3.


*Рис. 12.3.* Структура листовой страницы B+-дерева

Листовая страница обладает следующими свойствами:

* ключ1 < ключ2 < ... < ключk;
* списокr – упорядоченный список идентификаторов кортежей (tid), включающих значение ключr;
* листовые страницы связаны одно- или двунаправленным списком.

Поиск в B+-дереве – это прохождение от корня к листу в соответствии с заданным значением ключа. Заметим, что поскольку B+-деревья являются сильно ветвистыми и сбалансированными, для выполнения поиска по любому значению ключа потребуется одно и то же (и обычно небольшое) число обменов с внешней памятью. Более точно, в сбалансированном дереве, где длины всех путей от корня к листу одни и те же, если во внутренней странице помещается n ключей, то при хранении m записей требуется дерево глубиной logn(m). Если n достаточно велико (обычный случай), то глубина дерева невелика, и производится быстрый поиск.

Основной «изюминкой» B+-деревьев является автоматическое поддержание свойства сбалансированности. Рассмотрим, как это делается при выполнении операций занесения и удаления записей.

При занесение новой записи выполняются следующие действия.

* Поиск листовой страницы. Фактически, производится обычный поиск по ключу. Если в B+-дереве не содержится ключ с заданным значением, то будет получен номер страницы, в которой ему надлежит содержаться, и соответствующие координаты внутри страницы.
* Помещение записи на место. Естественно, что вся работа производится в буферах оперативной памяти. Листовая страница, в которую требуется занести запись, считывается в буфер, и в нем выполняется операция вставки. Размер буфера должен превышать размер страницы внешней памяти.
* Если после выполнения вставки новой записи размер используемой части буфера не превосходит размера страницы, то на этом выполнение операции занесения записи заканчивается. Буфер может быть немедленно вытолкнут во внешнюю память или временно сохранен в основной памяти в зависимости от политики управления буферами.
* Если же возникло переполнение буфера (т.е. размер его используемой части превосходит размер страницы), то выполняется расщепление страницы. Для этого запрашивается новая страница внешней памяти, используемая часть буфера разбивается примерно пополам (так, чтобы вторая половина также начиналась с ключа), и вторая половина записывается во вновь выделенную страницу, а в старой странице модифицируется значение размера свободной памяти. Естественно, модифицируются ссылки по списку листовых страниц.
* Чтобы обеспечить доступ от корня дерева к заново заведенной странице, необходимо соответствующим образом модифицировать внутреннюю страницу, являющуюся предком ранее существовавшей листовой страницы, т.е. вставить в нее соответствующее значение ключа и ссылку на новую страницу. При выполнении этого действия может снова произойти переполнение теперь уже внутренней страницы, и она будет расщеплена на две. В результате потребуется вставить значение ключа и ссылку на новую страницу во внутреннюю страницу-предка выше по иерархии и т.д.
* Предельным случаем является переполнение корневой страницы B+-дерева. В этом случае она тоже расщепляется на две, и заводится новая корневая страница дерева, т.е. его глубина увеличивается на единицу.

При удалении записи выполняются следующие действия.

* Поиск записи по ключу. Если запись не найдена, то удалять ничего не нужно.
* Реальное удаление записи в буфере, в который прочитана соответствующая листовая страница.
* Если после выполнения этой подоперации размер занятой в буфере области оказывается таковым, что его сумма с размером занятой области в листовых страницах, являющихся левым или правым братом данной страницы, больше, чем размер страницы, операция завершается.
* Иначе производится слияние с правым или левым братом, т.е. в буфере производится новый образ страницы, содержащей общую информацию из данной страницы и ее левого или правого брата. Ставшая ненужной листовая страница заносится в список свободных страниц. Соответствующим образом корректируется список листовых страниц.
* Чтобы устранить возможность доступа от корня к освобожденной странице, нужно удалить соответствующее значение ключа и ссылку на освобожденную страницу из внутренней страницы – ее предка. При этом может возникнуть потребность в слиянии этой страницы с ее левым или правым братом и т.д.
* Предельным случаем является полное опустошение корневой страницы дерева, которое возможно после слияния последних двух потомков корня. В этом случае корневая страница освобождается, а глубина дерева уменьшается на единицу.

Как видно, при выполнении операций вставки и удаления свойство сбалансированности B+-дерева сохраняется, а внешняя память расходуется достаточно экономно.

Проблемой является то, что при выполнении операций модификации слишком часто могут возникать расщепления и слияния. Чтобы добиться эффективного использования внешней памяти с минимизацией числа расщеплений и слияний, применяются более сложные приемы, в том числе:

* упреждающие расщепления, т.е. расщепления страницы не при ее переполнении, а несколько раньше, когда степень заполненности страницы достигает некоторого уровня;
* переливания, т.е. поддержание равновесного заполнения соседних страниц;
* слияния 3-в-2, т.е. порождение двух листовых страниц на основе содержимого трех соседних.

Следует заметить, что при организации мультидоступа к B+-деревьям, характерного при их использовании в СУБД, приходится решать ряд нетривиальных проблем. Конечно, грубые решения очевидны, например, возможен монопольный захват B+-дерева (т.е. его корневого блока) на все выполнение операции модификации. Но существуют и более тонкие решения, рассмотрение которых выходит за пределы материала этой книги.

###### Хэширование

Альтернативным и достаточно популярным подходом к организации индексов является использование техники *хэширования*. Это очень обширная тема, которая заслуживает отдельного рассмотрения. Ограничимся здесь лишь несколькими замечаниями. Общей идеей методов хэширования является применение к значению ключа некоторой *функции свертки (хэш-функции)*, вырабатывающей значение меньшего размера. Значение хэш-функции затем используется для доступа к записи.

В самом простом, классическом случае свертка ключа используется как адрес в таблице, содержащей ключи и записи. Основным требованием к хэш-функции является равномерное распределение значение свертки (одним из распространенных видов «хороших» хэш-функций являются функции, выдающие остаток от деления значения ключа на некоторое простое число). При возникновении *коллизий* (одна и та же свертка для нескольких значений ключа) образуются *цепочки переполнения*. Главным ограничением этого метода является фиксированный размер таблицы. Если таблица заполнена слишком сильно или переполнена, но возникнет слишком много цепочек переполнения, и главное преимущество хэширования – доступ к записи почти всегда за одно обращение к таблице – будет утрачено. Расширение таблицы требует ее полной переделки на основе новой хэш-функции (со значением свертки большего размера).

Идея доступа к данным на основе хэширования настолько привлекательна (потенциальная возможность за одно обращение к памяти получить требуемые данные), что от нее невозможно отказаться при работе с данными во внешней памяти. Исходная идея кажется очевидной: если при управлении данными на основе хэширования в основной памяти хэш-функция вырабатывает адрес требуемого элемента, то при обращении к внешней памяти необходимо генерировать номер блока дискового пространства, в котором находится запрашиваемый элемент данных. Основная проблема относится к коллизиям. Если при работе в основной памяти потенциально возникающими потребностями дополнительного поиска информации при возникновении коллизий можно, вообще говоря, пренебречь (поскольку время доступа к основной памяти мало), то при использовании внешней памяти любое дополнительное обращение вызывает существенные накладные расходы. Основные методы хэширования для поиска информации во внешней памяти направлены на решение именно этой задачи.

В основе подхода *расширяемого хэширования (Extendible Hashing)* [[3.14]](http://citforum.ru/database/advanced_intro/86.shtml#ref.3.14) лежит принцип использования деревьев цифрового поиска в основной памяти. В основной памяти поддерживается справочник, организованный на основе бинарного дерева цифрового поиска, ключами которого являются значения хэш-функции, а в листовых вершинах хранятся номера блоков записей во внешней памяти. В этом случае любой поиск в дереве цифрового поиска является «успешным», т.е. ведет к некоторому блоку внешней памяти. Входит ли в этот блок искомая запись, обнаруживается уже после прочтения блока в основную память.

Проблема коллизий переформулируется следующим образом. Как таковых, коллизий не существует. Может возникнуть лишь ситуация переполнения блока внешней памяти. Значение хэш-функции указывает на этот блок, но места для включения записи в нем уже нет. Эта ситуация обрабатывается так. Блок расщепляется на два, и дерево цифрового поиска переформируется соответствующим образом. Конечно, при этом может потребоваться расширение самого справочника.

Расширяемое хэширование хорошо работает в условиях динамически изменяемого набора записей в хранимом файле, но требует наличия в основной памяти справочного дерева.

Идея *линейного хэширования (Linear Hashing)* [[3.15]](http://citforum.ru/database/advanced_intro/86.shtml#ref.3.15) состоит в том, чтобы можно было обойтись без поддержания справочника в основной памяти. Основой метода является то, что для адресации блока внешней памяти всегда используются младшие биты значения хэш-функции. Если возникает потребность в расщеплении, то записи перераспределяются по блокам так, чтобы адресация осталась правильной.

#### 12.3.3. Журнальная информация

Структура журнала обычно является сугубо частным делом конкретной реализации. Отметим только самые общие свойства.

Журнал обычно представляет собой чисто последовательный файл с записями переменного размера, которые можно просматривать в прямом или обратном порядке. Обмены производятся стандартными порциями (страницами) с использованием буфера оперативной памяти. В грамотно организованных системах структура (и тем более, смысл) журнальных записей известна только компонентам СУБД, ответственным за журнализацию и восстановление.

Поскольку содержимое журнала является критичным при восстановлении базы данных после сбоев, к ведению файла журнала предъявляются особые требования по части надежности. В частности, обычно стремятся поддерживать две идентичные копии журнала на разных устройствах внешней памяти.

#### 12.3.4. Служебная информация

Для корректной работы подсистемы управления данными во внешней памяти необходимо поддерживать информацию, которая используется только этой подсистемой и не видна подсистеме языкового уровня. Набор структур служебной информации зависит от общей организации системы, но обычно требуется поддержание следующих служебных данных:

* Внутренние каталоги, описывающие физические свойства объектов базы данных, например, число атрибутов таблицы, их размер и, возможно, типы данных; описание индексов, определенных для данной таблицы и т.д.
* Описатели свободной и занятой памяти в страницах данных. Такая информация требуется для нахождения свободного места при занесении кортежа. Отдельно приходится решать задачу поиска свободного места в случаях некластеризованных и кластеризованных таблиц (в последнем случае приходится дополнительно использовать кластеризованный индекс). Как уже отмечалось, нетривиальной является проблема освобождения страницы в условиях мультидоступа.
* Связывание страниц одной таблицы. Если в одном файле внешней памяти могут располагаться страницы нескольких таблиц (обычно к этому стремятся), то нужно каким-то образом связать страницы одной таблицы. Тривиальный способ использования прямых ссылок между страницами часто приводит к затруднениями при синхронизации транзакций (например, особенно трудно освобождать и заводить новые страницы таблицы). Поэтому стараются использовать косвенное связывание страниц с использованием служебных индексов. В частности, известен общий механизм для описания свободной памяти и связывания страниц на основе B-деревьев.

### 12.4. Заключение

В этой лекции была кратко описана экспериментальная реляционная СУБД System R, оказавшая гигантское влияние на становление современной технологии баз данных. Были рассмотрены основные цели проекта System R, общая архитектура системы, основные структуры данных внешней памяти и интерфейс подсистемы управления внешней памятью.

Далее были обсуждены основные подходы к организации внешней памяти, применяемые в современных СУБД. Конечно, в любой конкретной SQL-ориентированной СУБД используется ряд собственных приемов организации хранения таблиц и индексов, но практически во всех случаях общие принципы похожи на те, которые описаны в разд. 12.2.

## Лекция 13. Методы управления транзакциями. Сихронизационные блокировки, временные метки и версии

### 13.1. Введение

Поддержка механизма транзакций – показатель уровня развитости СУБД. Корректное поддержание транзакций одновременно является основой обеспечения целостности баз данных (и поэтому транзакции вполне уместны и в однопользовательских персональных СУБД), а также составляют базис изолированности пользователей в многопользовательских системах. Часто эти два аспекта рассматриваются по отдельности, но на самом деле они взаимосвязаны, что и будет показано в этой лекции.

### 13.2. Общее понятие транзакции и основные характеристики транзакций

Более точно, в современных СУБД поддерживается понятие транзакции, характеризуемое аббревиатурой ACID (Atomicy, Consistency, Isolation и Durability). В соответствии с этим понятием под транзакцией разумеется последовательность операций над базой данных, обладающая следующими свойствами.

* *Атомарность (Atomicy)*. Это свойство означает, что результаты всех операций, успешно выполненных в пределах транзакции, должны быть отражены в состоянии базы данных, либо в состоянии базы данных не должно быть отражено действие ни одной операции (конечно, здесь речь идет об операциях, изменяющих состояние базы данных). Свойство атомарности, которое часто называют свойством “все или ничего”, позволяет относиться к транзакции, как к динамически образуемой составной операции над базой данных (в общем случае состав и порядок выполнения операций, выполняемых внутри транзакции, становится известным только на стадии выполнения).
* *Согласованность (Consistency).* В классическом смысле это свойство означает, что транзакция может быть успешно завершена с *фиксацией* результатов своих операций только в том случае, когда действия операций не нарушают *целостность* базы данных, т.е. удовлетворяют набору ограничений целостности, определенных для этой базы данных. Это свойство расширяется тем, что во время выполнения транзакции разрешается устанавливать точки согласованности и явным образом проверять ограничения целостности. (С точки зрения автора, в контексте баз данных термины *согласованность* и *целостность* эквивалентны. Единственным критерием согласованности данных является их удовлетворение ограничениям целостности, т.е. база данных находится в согласованном состоянии тогда и только тогда, когда она находится в целостном состоянии.)
* *Изоляция (Isolation).* Требуется, чтобы две одновременно (параллельно или квазипараллельно) выполняемые транзакции никоим образом не действовали одна на другую. Другими словами, результаты выполнения операций транзакции *T1* не должны быть видны никакой другой транзакции *T2* до тех пор, пока транзакция *T1* не завершится успешным образом.
* *Долговечность (Durability).* После успешного завершения транзакции все изменения, которые были внесены в состояние базы данных операциями этой транзакции, должны гарантированно сохраняться, даже в случае сбоев аппаратуры или программного обеспечения. Этому аспекту транзакционных систем посвящается лекция 14.

Заметим, что хотя с точки зрения обеспечения целостности баз данных механизм транзакций следовало бы поддерживать в персональных СУБД, на практике это обычно не выполняется. Поэтому при переходе от персональных к многопользовательским СУБД пользователи сталкиваются с необходимостью четкого понимания природы транзакций.

#### 13.2.1. Атомарность транзакций

В этом смысле под транзакцией понимается неделимая с точки зрения воздействия на БД последовательность операторов манипулирования данными (чтения, удаления, вставки, модификации), такая, что либо результаты всех операторов, входящих в транзакцию, отображаются в состоянии базы данных, либо воздействие всех этих операторов полностью отсутствует.

Лозунгом транзакции является «Все или ничего»: при завершении транзакции оператором COMMIT (высокоуровневый аналог операции END TRANSACTION в интерфейсе RSS, см. лекцию 12) результаты гарантированно фиксируются во внешней памяти (смысл термина *commit* состоит в запросе «фиксации» результатов транзакции); при завершении транзакции оператором ROLLBACK (высокоуровневый аналог операции RESTORE в интерфейсе RSS, см. лекцию 12) результаты гарантированно отсутствуют во внешней памяти (смысл термина *rollback* состоит в запросе ликвидации результатов транзакции).

Каким образом в СУБД поддерживаются индивидуальные откаты транзакций, описывается в лекции 14.

#### 13.2.2. Транзакции и целостность баз данных

Понятие транзакции имеет непосредственную связь с понятием целостности базы данных. Очень часто база данных может обладать такими ограничениями целостности, которые просто невозможно не нарушить, выполняя только один оператор изменения базы данных. Например, в базе данных СЛУЖАЩИЕ-ОТДЕЛЫ (см. лекцию 1) естественным ограничением целостности является совпадение значения атрибута ОТД\_РАЗМЕР в кортеже таблицы ОТДЕЛЫ, описывающей данный отдел (например, отдел 625), с числом кортежей таблицы СЛУЖАЩИЕ, таких, что значение поля СЛУ\_ОТД\_НОМЕР равно 625. Как в этом случае принять на работу в отдел 625 нового сотрудника? Независимо от того, какая операция будет выполнена первой, вставка нового кортежа в таблице СОТРУДНИКИ или модификация существующего кортежа в отношении ОТДЕЛЫ, после выполнения операции база данных окажется в нецелостном состоянии.

Поэтому для поддержки подобных ограничений целостности допускается их нарушение внутри транзакции с тем условием, чтобы к моменту завершения транзакции условия целостности были соблюдены. В системах с развитыми средствами ограничения и контроля целостности каждая транзакция начинается при целостном состоянии базы данных и должна оставить это состояние целостными после своего завершения. Несоблюдение этого условия приводит к тому, что вместо фиксации результатов транзакции происходит ее откат (т.е. вместо оператора COMMIT выполняется оператор ROLLBACK), и база данных остается в таком состоянии, в котором находилась к моменту начала транзакции, т.е. в целостном состоянии.

Более точно, различаются два вида ограничений целостности: немедленно проверяемые и откладываемые. К немедленно проверяемым ограничениям целостности относятся такие ограничения, проверку которых бессмысленно или даже невозможно откладывать. Примером ограничения, проверку которого откладывать бессмысленно, являются ограничения домена (например, возраст сотрудника не может превышать 150 лет). Более сложным ограничением, проверку которого невозможно отложить, является следующее: зарплата сотрудника не может быть увеличена за одну операцию более чем на 100000 рублей. Немедленно проверяемые ограничения целостности соответствуют уровню отдельных операторов языкового уровня СУБД. При их нарушениях не производится откат транзакции, а лишь отвергается соответствующий оператор.

Откладываемые ограничения целостности – это ограничения на базу данных, а не на какие-либо отдельные операции. По умолчанию такие ограничения проверяются при конце транзакции, и их нарушение вызывает автоматическую замену оператора COMMIT на оператор ROLLBACK. Однако в некоторых системах поддерживается специальный оператор насильственной проверки ограничений целостности внутри транзакции. Если после выполнения такого оператора обнаруживается, что условия целостности не выполнены, пользователь может сам выполнить оператор ROLLBACK с откатом транзакции до ее начала или до установленной ранее точки сохранения или постараться устранить причины нецелостного состояния базы данных внутри транзакции (видимо, это осмысленно только при использовании интерактивного режима работы).

Заметим, что концептуально в момент завершения транзакции проверяются все откладываемые ограничения целостности, определенные в этой базе данных. Однако в реализации стремятся при выполнении транзакции динамически выделить те ограничения целостности, которые действительно могли бы быть нарушены. Например, если при выполнении транзакции над базой данных СЛУЖАЩИЕ-ОТДЕЛЫ в ней не выполнялись операторы вставки или удаления кортежей из отношения СЛУЖАЩИЕ, то проверять упоминавшееся выше ограничение целостности не требуется (а для проверки подобных ограничений требуется достаточно большая работа).

Понятно, что описанный механизм поддержки целостности баз данных обеспечивает требуемое свойство транзакций: никакая транзакция не может быть зафиксирована, если ее действия нарушили целостность базы данных. Однако в этом подходе имеются два серьезных дефекта.

Во-первых, если при выполнении транзакции не устанавливать точки сохранения и не проверять периодически соответствие текущего состояния базы данных (с точки зрения данной транзакции) ограничениям целостности, то долговременно выполняемая транзакция вполне вероятно может быть «откачена» системой при выполнении завершающего оператора COMMIT. Конечно, это означает непроизводительный расход системных ресурсов и времени пользователей. Во-вторых, чем длиннее транзакция, модифицирующая состояние базы данных, тем потенциально больше ограничений целостности придется проверять при ее завершении и тем накладнее становится оператор COMMIT.

Простое и элегантное решение этой проблемы предлагается в [[1.5]](http://citforum.ru/database/advanced_intro/86.shtml#ref.1.5). Авторы предлагают отказаться от откладываемых ограничений целостности базы данных, а вместо этого ввести составные операторы изменения базы данных (нечто наподобие блоков BEGIN … END, поддерживаемых в языках программирования). После выполнения каждого такого блока (или отдельного оператора изменения базы данных, используемого без операторов начала и конца блока) база данных должна находится в целостном состоянии. Если составной оператор нарушает ограничение целостности, то он целиком отвергается, и вырабатывается соответствующий код ошибки. Транзакция в этом случае не откатывается. Понятно, что при использовании такого подхода при выполнении оператора COMMIT не требуется проверять ограничения целостности, и каждая зафиксированная транзакция будет оставлять базу данных в целостном состоянии.

Интересно, что для реализации описанного подхода не требуются какие-либо новые механизмы, кроме точек сохранения транзакции, насильственной проверки ограничений целостности и частичных откатов транзакций, а отмеченные ранее проблемы снимаются. К сожалению, насколько известно автору данной книги, этот подход на практике пока не применяется.

#### 13.2.3. Изолированность транзакций

В многопользовательских системах с одной базой данных одновременно может работать несколько пользователей или прикладных программ. Предельной задачей системы является обеспечение изолированности пользователей, т.е. создание достоверной и надежной иллюзии того, что каждый из пользователей работает с базой данных в одиночку.

В связи со свойством сохранения целостности базы данных транзакции являются подходящими единицами изолированности пользователей. Действительно, если с каждым сеансом работы пользователя или приложений с базой данных ассоциируется транзакция, то каждый пользователь начинает работу с согласованным состоянием базы данных, т.е. с таким состоянием, в котором база данных могла бы находиться, даже если бы пользователь работал с ней в одиночку.

При соблюдении обязательного требования поддержки целостности базы данных возможно наличие нескольких уровней изолированности транзакций. Заметим, что впервые эти уровни изолированности транзакций были установлены и описаны участниками проекта System R.

##### Отсутствие потерянных изменений (первый уровень изолированности)

Рассмотрим сценарий совместного выполнения двух транзакций, показанный на рис. 13.1. В момент времени t1 транзакция T1 изменяет объект базы данных o (выполняет операцию W(o)). До завершения транзакции T1 в момент времени t2 > t1 транзакция T2 также изменяет объект o. В момент времени t3 > t2 транзакция T2 завершается оператором ROLLBACK (например, по причине нарушения ограничений целостности).


*Рис. 13.1.* Потерянные изменения

Тогда при повторном чтении объекта o (выполнении операции R(o)) в момент времени t4 > t3 транзакция T1 не видит своих изменений этого объекта, произведенных ранее (в частности, из-за этого может не удастся фиксация этой транзакции, что, возможно, повлечет потерю изменений у еще одной транзакции и т.д.).

Такая ситуация называется ситуацией *потерянных изменений*. Естественно, она противоречит требованию изолированности пользователей. Чтобы избежать такой ситуации в транзакции T1 требуется, чтобы до завершения транзакции T1 никакая другая транзакция не могла изменять никакой измененный транзакцией T1 объект o (в частности, достаточно заблокировать доступ по изменению к объекту o до завершения транзакции T1). Отсутствие потерянных изменений является минимальным требованием к СУБД при обеспечении изолированности одновременно выполняемых транзакций.

##### Отсутствие чтения «грязных» данных (второй уровень изолированности)

Рассмотрим сценарий совместного выполнения транзакций T1 и T2, показанный на рис. 13.2. В момент времени t1 транзакция T1 изменяет объект базы данных o (выполняет операцию W(o)). В момент времени t2 > t1 транзакция T2 читает объект o (выполняет операцию R(o)). Поскольку транзакция T1 еще не завершена, транзакция T2 видит несогласованные *«грязные» данные*. В частности, в момент времени t3 > t2 транзакция T1 может завершиться откатом (например, по причине нарушения ограничений целостности).


*Рис. 13.2.* «Грязные» чтения

Эта ситуация тоже не соответствует требованию изолированности пользователей (каждый пользователь начинает свою транзакцию при согласованном состоянии базы данных и имеет право видеть только согласованные данные). Чтобы избежать ситуации чтения "грязных" данных, до завершения транзакции T1, изменившей объект базы данных o, никакая другая транзакция не должна читать объект o (например, достаточно заблокировать доступ по чтению к объекту o до завершения изменившей его транзакции T1).

##### Отсутствие неповторяющихся чтений (третий уровень изоляции)

Рассмотрим сценарий совместного выполнения транзакций T1 и T2, показанный на рис. 13.3. В момент времени t1 транзакция T1 читает объект базы данных o (выполняет операцию R(o)). До завершения транзакции T1 в момент времени t2 > t1 транзакция T2 изменяет объект o (выполняет операцию W(o)) и успешно завершается оператором COMMIT. В момент времени t3 > t2 транзакция T1 повторно читает объект o и видит его измененное состояние.


*Рис. 13.3.* Неповторяющиеся чтения

Чтобы избежать *неповторяющихся чтений*, до завершения транзакции T1 никакая другая транзакция не должна изменять объект o (для этого достаточно заблокировать доступ по записи к объекту o до завершения транзакции T1). Часто это является максимальным требованием к средствам обеспечения изолированности транзакций, хотя, как будет видно немного позже, отсутствие неповторяющихся чтений еще не гарантирует реальной изолированности пользователей.

Заметим, что существует возможность обеспечения разных уровней изолированности для разных транзакций, выполняющихся в одной системе баз данных (кстати, соответствующие операторы были предусмотрены уже в стандарте SQL:1992). Как уже отмечалось, для корректного соблюдения ограничений целостности достаточен первый уровень. Существует ряд приложений, которым хватает первого уровня изолированности (например, прикладные или системные статистические утилиты, для которых некорректность индивидуальных данных несущественна). При этом удается существенно сократить накладные расходы СУБД и повысить общую эффективность.

##### Проблема фантомов

К более тонким проблемам изолированности транзакций относится так называемая проблема *кортежей-«фантомов»*, приводящая к ситуациям, которые также противоречат изолированности пользователей. Рассмотрим сценарий, показанный на рис. 13.4.


*Рис. 13.4.* Проблема фантомов

В момент времени t1 транзакция T1 выполняет оператор выборки кортежей таблицы Tab с условием выборки S (т.е. выбирается часть кортежей таблицы Tab, удовлетворяющих условию S). До завершения транзакции T1 в момент времени t2 > t1 транзакция T2 вставляет в таблицу Tab новый кортеж r, удовлетворяющий условию S, и успешно завершается. В момент времени t3 > t2 транзакция T1 повторно выполняет тот же оператор выборки, и в результате появляется кортеж, который отсутствовал при первом выполнении оператора.

Конечно, такая ситуация противоречит идее изолированности транзакций и может возникнуть даже на третьем уровне изолированности транзакций. Чтобы избежать появления кортежей-фантомов, требуется более высокий «логический» уровень изоляции транзакций. Идеи требуемого механизма (предикатные синхронизационные блокировки) появились также еще во время выполнения проекта System R, но в большинстве систем не реализованы.

#### 13.2.4. Сериализация транзакций

Чтобы добиться изолированности транзакций, в СУБД должны использоваться какие-либо методы регулирования совместного выполнения транзакций.

Пусть в системе одновременно выполняется некоторое множество транзакций S = {T1, T2, …, T*n*}. План (способ) выполнения набора транзакций S (в котором, вообще говоря, чередуются или реально параллельно выполняются операции разных транзакций) называется сериальным, если результат совместного выполнения транзакций эквивалентен результату некоторого последовательного выполнения этих же транзакций (T*i*1, T*i*2, …, T*in*).

Сериализация транзакций – это механизм их выполнения по некоторому сериальному плану. Обеспечение такого механизма является основной функцией компонента СУБД, ответственного за управление транзакциями. Система, в которой поддерживается сериализация транзакций, обеспечивает реальную изолированность пользователей.

Основная реализационная проблема состоит в выборе метода сериализации набора транзакций, который не слишком ограничивал бы чередование их операций или реальную параллельность. Приходящим на ум тривиальным решением является действительно последовательное выполнение транзакций. Но существуют ситуации, в которых можно выполнять операторы разных транзакций в любом порядке с сохранением свойства сериальности. Примерами могут служить только читающие транзакции, а также транзакции, не конфликтующие по объектам базы данных.

Между транзакциями T1 и T2 могут существовать следующие виды конфликтов:

* W/W – транзакция T2 пытается изменять объект, измененный не закончившейся транзакцией T1 (наличие такого конфликта может привести к возникновению ситуации потерянных изменений);
* R/W – транзакция T2 пытается изменять объект, прочитанный не закончившейся транзакцией T1 (наличие такого конфликта может привести к возникновению ситуации неповторяющихся чтений);
* W/R – транзакция T2 пытается читать объект, измененный не закончившейся транзакцией T1 (наличие такого конфликта может привести к возникновению ситуации «грязного» чтения).

Практические методы сериализации транзакций основываются на учете этих конфликтов.

### 13.3. Методы сериализации транзакций

Существуют два базовых подхода к сериализации транзакций – основанный на синхронизационных захватах объектов базы данных и на использовании временных меток. Суть обоих подходов состоит в обнаружении конфликтов транзакций и их устранении. Ниже мы рассмотрим эти подходы сравнительно подробно. Кроме того, кратко обсудим возможности использования версий объектов базы данных для ускорения выполнения «только читающих» транзакций, т.е. транзакций, в которых не выполняются операции изменения базы данных.

Предварительно заметим, что для каждого из подходов имеются две разновидности – пессимистическая и оптимистическая. При применении пессимистических методов, ориентированных на ситуации, когда конфликты возникают часто, конфликты распознаются и разрешаются немедленно при их возникновении. Оптимистические методы основываются на том, что результаты всех операций модификации базы данных сохраняются в рабочей памяти транзакций. Реальная модификация базы данных производится только на стадии фиксации транзакции. Тогда же проверяется, не возникают ли конфликты с другими транзакциями.

Далее мы ограничимся рассмотрением более распространенных пессимистических разновидностей методов сериализации транзакций. Пессимистические методы сравнительно просто трансформируются в свои оптимистические варианты.

#### 13.3.1. Синхронизационные блокировки

Наиболее распространенным в централизованных СУБД (включающих системы, основанные на архитектуре «клиент-сервер») является подход, основанный на соблюдении двухфазного протокола синхронизационных захватов объектов баз данных (Two-Phase Locking Protocol, 2PL). В общих чертах подход состоит в том, что перед выполнением любой операции в транзакции T над объектом базы данных *o* от имени транзакции T запрашивается синхронизационная блокировка объекта *o* в соответствующем режиме (в зависимости от вида операции).

Основными режимами синхронизационных блокировок являются следующие:

* совместный режим – S (Shared), означающий совместную (по чтению) блокировку объекта и требуемый для выполнения операции чтения объекта;
* монопольный режим – X (eXclusive), означающий монопольную (по записи) блокировку объекта и требуемый для выполнения операций вставки, удаления и модификации объекта.

Блокировки одних и тех же объектов по чтению несколькими транзакциями совместимы, т.е. нескольким транзакциям допускается одновременно читать один и тот же объект. Блокировка объекта одной транзакцией по чтению не совместима с блокировкой другой транзакцией того же объекта по записи, т.е. никакой транзакции нельзя изменять объект, читаемый некоторой транзакцией (кроме самой этой транзакции), и никакой транзакции нельзя читать объект, изменяемый некоторой транзакцией (кроме самой этой транзакции). Блокировки одного и того же объекта по записи разными транзакциями не совместимы, т.е. никакой транзакции нельзя изменять объект, изменяемый некоторой транзакцией (кроме самой этой транзакции). Правила совместимости захватов одного объекта разными транзакциями приведены в таблице 10.1.

В первом столбце приведены возможные состояния объекта с точки зрения синхронизационных захватов. При этом "-" соответствует состоянию объекта, для которого не установлен никакой захват. Транзакция, запросившая синхронизационный захват объекта БД, уже захваченный другой транзакцией в несовместимом режиме, блокируется до тех пор, пока захват с этого объекта не будет снят.

Таблица 9.1. Совместимость блокировок S и X

|  |  |  |
| --- | --- | --- |
|   | X | S |
| - | да | да |
| X | нет | нет |
| S | нет | да |

Заметим, что слово «нет» (отсутствие совместимости блокировок) в этой таблице соответствует описанным ранее возможным случаям конфликтов транзакций по доступу к объектам базы данных (W/W, R/W, W/R). Совместимость S-блокировок соответствует тому, что конфликт R/R не существует.

Для обеспечения сериализации транзакций (третьего уровня изолированности) синхронизационные блокировки объектов, произведенные по инициативе транзакции, можно снимать только при ее завершении (см. примеры сценариев, обсуждавшихся в разделе [13.2 Общее понятие транзакции и основные характеристики транзакций](http://citforum.ru/database/advanced_intro/39.shtml#13.2)). Это требование порождает двухфазный протокол синхронизационных захватов – 2PL. В соответствии с этим протоколом выполнение транзакции разбивается на две фазы:

* первая фаза транзакции (выполнение операций над базой данных) – накопление блокировок;
* вторая фаза (фиксация или откат) – снятие блокировок.

Достаточно легко убедиться, что при соблюдении двухфазного протокола синхронизационных блокировок действительно обеспечивается сериализация транзакций на третьем уровне изолированности. Также легко видеть, что для обеспечения отсутствия потерянных данных достаточно блокировать в режиме X изменяемые объекты базы данных и удерживать эти блокировки до конца транзакции, а для обеспечения отсутствия чтения «грязных» данных достаточно блокировать в режиме X изменяемые объекты до конца транзакции и блокировать в режиме S читаемые объекты на время выполнения операции чтения.

Основная проблема состоит в том, что следует считать объектом для синхронизационного захвата? В контексте реляционных баз данных возможны следующие альтернативы:

* файл (сегмент в терминах System R) – физический (с точки зрения базы данных) объект, область хранения нескольких таблиц и, возможно, индексов;
* таблица – логический объект, соответствующий множеству кортежей данной таблицы;
* страница данных – физический объект, хранящий кортежи одной или нескольких таблиц, индексную или служебную информацию;
* кортеж – элементарный физический объект базы данных.

На самом деле, любая операция над объектом базы данных фактически воздействует и на объемлющие его объекты. Например, операция над кортежем является и операцией над страницей, в которой этот кортеж хранится, и над соответствующей таблицей, и над файлом, содержащим таблицу. Поэтому действительно имеется выбор уровня объекта блокировки.

Понятно, что для поддержки блокировок требуются системные ресурсы, и что чем крупнее объект синхронизационного захвата (неважно, какой природы этот объект – логический или физический), тем меньше синхронизационных блокировок будет поддерживаться в системе, и на это, соответственно, будут тратиться меньшие накладные расходы. Более того, если устанавливать блокировки на уровне файлов или таблиц, то будет решена даже проблема фантомов (если это не ясно сразу, посмотрите еще раз описание проблемы фантомов и определение двухфазного протокола захватов).

Но вся беда в том, что при использовании для блокировок крупных объектов возрастает вероятность конфликтов транзакций и, тем самым, уменьшается допускаемая степень чередования их операций или реального параллельного выполнения. Фактически, при укрупнении объекта синхронизационной блокировки мы умышленно огрубляем ситуацию и видим конфликты в тех ситуациях, в которых на самом деле конфликтов нет.

Разработчики многих систем начинали с использования страничных блокировок, полагая это некоторым компромиссом между стремлениями сократить накладные расходы и сохранить достаточно высокий уровень параллельности транзакций. Но это не очень хороший выбор. Не будем останавливаться на деталях, но заметим, что использование страничных блокировок в двухфазном протоколе иногда вызывает очень неприятные синхронизационные проблемы, усложняющие организацию СУБД (коротко говоря, эти проблемы связаны с тем, что страницы приходится блокировать на двух разных уровнях – уровне управления буферами страниц в основной памяти и уровне выполнения логических операций). В большинстве современных систем используются покортежные синхронизационные блокировки.

Но при этом возникает очередной вопрос. Если единицей блокировки является кортеж, то какие синхронизационные блокировки потребуются при выполнении таких операций как уничтожение заполненной таблицы? Было бы довольно нелепо перед выполнением такой операции потребовать блокировки всех существующих кортежей таблицы. Кроме того, это не предотвратило бы возможности параллельной вставки нового кортежа в уничтожаемое отношение в некоторой другой транзакции.

##### Гранулированные синхронизационные блокировки

Подобные рассуждения привели к разработке механизма гранулированных синхронизационных блокировок. При применении этого подхода синхронизационные блокировки могут запрашиваться по отношению к объектам разного уровня: файлам, таблицам и кортежам. Требуемый уровень объекта определяется тем, какая операция выполняется (например, для выполнения операции уничтожения таблицы объектом синхронизационной блокировки должна быть вся таблица, а для выполнения операции удаления кортежа – этот кортеж). Объект любого уровня может быть заблокирован в режиме S или X.

Для согласования блокировок разного уровня вводятся специальный протокол гранулированных блокировок и новые типы блокировок. Коротко говоря, перед установкой блокировки на некоторый объект базы данных в режиме S или X соответствующий объект верхнего уровня должен быть заблокирован в режиме IS, IX или SIX. Что же собой представляют эти режимы блокировок?

Блокировка в режиме IS (Intented for Shared lock) некоторого составного объекта o базы данных означает намерение заблокировать некоторый объект o', входящий в o, в совместном режиме (режиме S). Например, при намерении читать кортежи из таблицы Tab эта таблица должна быть заблокирована в режиме IS (а до этого в таком же режиме должен быть заблокирован файл, в котором располагается таблица Tab).

Блокировка в режиме IX (Intented for eXclusive lock) некоторого составного объекта o базы данных означает намерение заблокировать некоторый объект o', входящий в o, в монопольном режиме (режиме X). Например, для удаления кортежей из таблицы Tab эта таблица должна быть заблокирована в режиме IX (а до этого в таком же режиме должен быть заблокирован файл, в котором располагается таблица Tab).

Блокировка в режиме SIX (Shared, Intented for eXclusive lock) некоторого составного объекта o базы данных означает совместную блокировку всего этого объекта с намерением впоследствии блокировать какие-либо входящие в него объекты в монопольном режиме (режиме X). Например, если выполняется длинная операция просмотра таблицы Tab с возможностью удаления некоторых просматриваемых кортежей, то экономичнее всего заблокировать таблицу Tab в режиме SIX (а до этого заблокировать в режиме IS файл, в котором располагается таблица Tab).

Таблица 9.2. Совместимость блокировок S, X, IS, IX и SIX

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|   | X | S | IX | IS | SIX |
| - | да | да | да | да | да |
| X | нет | нет | нет | нет | нет |
| S | нет | да | нет | да | нет |
| IX | нет | нет | да | да | нет |
| IS | нет | да | да | да | да |
| SIX | нет | нет | нет | да | нет |

В таб. 9.2 приведена таблица совместимости блокировок S, X, IS, IX и SIX. Немного поясним правила совместимости. Должно быть понятно, что для атомарных объектов разумны только блокировки в режимах S и X, для которых правила совместимости остаются такими же, как были показаны в таб. 9.1. Пусть теперь *o* – это некоторый составной объект.

Тогда блокировка объекта *o* в режиме X в транзакции T1 не совместима с блокировкой этого объекта в режимах X, S, IX, IS или SIX в транзакции T2. Действительно, блокировка объекта *o* в режиме X в транзакции T1 направлена на то, чтобы изменять объект *o* целиком. Несовместимость блокировки объекта *o* в режиме X в транзакции T1 с его блокировкой в режиме X или IX в транзакции T2 устраняет конфликты транзакций T1 и T2 вида W/W. Несовместимость блокировки объекта *o* в режиме X в транзакции T1 с его блокировкой в режиме S или IS в транзакции T2 устраняет конфликты транзакций T1 и T2 вида W/R. Наконец, несовместимость блокировки объекта *o* в режиме X в транзакции T1 с его блокировкой в режиме SIX в транзакции T2 устраняет конфликты транзакций T1 и T2 вида W/R и W/W.

Блокировка объекта *o* в режиме S в транзакции T1 совместима с блокировкой этого объекта в режимах S или IS в транзакции T2, поскольку эти блокировки в транзакциях T1 и T2 направлены только на то, чтобы только читать некоторые объекты *o*', входящие в *o*. Блокировка объекта *o* в режиме S в транзакции T1 не совместима с блокировкой этого объекта в режимах X, IX или SIX в транзакции T2, поскольку любая из этих блокировок направлена на то, чтобы изменять в транзакции T2 объект o целиком или какой-либо объект *o*', входящий в o. Несовместимость блокировки объекта *o* в режиме S в транзакции T1 с блокировкой этого объекта в режимах X, IX или SIX в транзакции T2, тем самым, устраняет конфликты транзакций T1 и T2 вида R/W.

Блокировка объекта *o* в режиме IХ в транзакции T1 совместима с блокировкой этого же объекта в режимах IS или IX в транзакции T2. Действительно, блокировка объекта *o* в режиме IX в транзакции T1 направлена на то, чтобы в этой транзакции изменять какой-либо объект *o*', входящий в *o*, а блокировка этого же объекта в режиме IS в транзакции T2 – на то, чтобы читать в транзакции T2 какой-либо объект *o*'', входящий в *o*. Если объекты *o*' и *o*'' – разные, то конфликт транзакций T1 и T2 не возникнет. Если *o*' = *o*'', то перед изменением этот объект будет заблокирован в транзакции T1 в режиме X, а перед чтением – в транзакции T2 в режиме S. Несовместимость этих блокировок позволит избежать конфликта транзакций T1 и T2 вида W/R, и для этого не требуется несовместимость блокировок IX и IS объекта *o*. Аналогично обосновывается совместимость блокировок IX и IX. Блокировка IХ не совместима с блокировкой S, поскольку иначе мог бы проявиться конфликт транзакций T1 и T2 вида W/R. Блокировка IХ не совместима с блокировкой X, поскольку иначе мог бы проявиться конфликт транзакций T1 и T2 вида W/W. Наконец, блокировка IХ не совместима с блокировкой SIX, поскольку иначе мог бы проявиться конфликт транзакций T1 и T2 вида W/R или W/W.

Блокировка объекта *o* в режиме IS в транзакции T1 совместима с блокировкой этого же объекта в режимах S, IS, IX или SIX в транзакции T2. Совместимость с блокировкой в режиме S или IS уже обосновывалась. Покажем, что блокировка объекта *o* в режиме IS в транзакции T1 совместима с блокировкой того же объекта в режиме IX в транзакции T2. Действительно, блокировка объекта *o* в режиме IS в транзакции T1 направлена на то, чтобы в этой транзакции читать какой-либо объект *o*', входящий в *o*, а блокировка этого же объекта в режиме IX в транзакции T2 – на то, чтобы в транзакции T2 изменять какой-либо объект *o*'', входящий в *o*. Если объекты *o*' и *o*'' – разные, то конфликт транзакций не возникнет. Если *o*' = *o*'', то перед чтением этот объект будет заблокирован в транзакции T1 в режиме S, а перед изменением – в транзакции T2 в режиме X. Несовместимость этих блокировок позволит избежать конфликта транзакций T1 и T2 вида R/W, и для этого не требуется несовместимость блокировок IS и IX объекта *o*. Аналогично можно показать совместимость блокировок IS и SIX. Несовместимость блокировок IS и X очевидна, поскольку иначе мог бы проявиться конфликт транзакций T1 и T2 вида R/W.

Блокировка объекта *o* в режиме SIX в транзакции T1 позволяет этой транзакции читать любой объект *o*', входящий в *o*, без его дополнительной блокировки и изменять любой объект *o*', входящий в *o*, с его предварительной блокировкой в режиме X. Эта блокировка совместима с блокировкой объекта *o* в режиме IS в транзакции T2. Действительно, блокировка объекта *o* в режиме IS в транзакции T2 направлена на то, чтобы в транзакции T2 читать какой-либо объект *o*', входящий в *o*. Перед этим в транзакции T2 должна быть установлена блокировка объекта *o*' в режиме S. К этому моменту у объекта *o*' может отсутствовать явная блокировка, установленная в транзакции T1, что, в соответствии с семантикой блокировки SIX, означает наличие неявной блокировки *o*' по чтению. Очевидно, что в этом случае конфликт транзакций T1 и T2 не возникает. К этому же моменту у объекта *o*' может иметься блокировка в режиме X, установленная в транзакции T1. В этом случае запрос блокировки объекта *o*' в режиме S удовлетворен не будет, и конфликт транзакций T1 и T2 вида W/R будет предотвращен без потребности в несовместимости блокировок SIX и IS. Блокировка объекта *o* в режиме SIX в транзакции T1 не совместима с блокировкой объекта *o* в режиме X в транзакции T2, поскольку иначе мог бы проявиться конфликт транзакций T1 и T2 вида R/W. Блокировка объекта *o* в режиме SIX в транзакции T1 не совместима с блокировкой объекта o в режиме S или IS в транзакции T2, поскольку иначе мог бы проявиться конфликт транзакций T1 и T2 вида W/R при доступе к некоторым объектам *o*', входящим в o. Наконец, блокировка объекта *o* в режиме SIX в транзакции T1 не совместима с блокировкой объекта *o* в режиме IX или SIX в транзакции T2, поскольку иначе мог бы проявиться конфликт транзакций T1 и T2 вида R/W при доступе к некоторым объектам *o*', входящим в *o*.

##### Предикатные синхронизационные блокировки

Несмотря на привлекательность метода гранулированных синхронизационных захватов, следует отметить, что он не решает проблему фантомов (если, конечно, не ограничиться использованием блокировок таблиц в режимах S и X). Давно известно, что для решения этой проблемы необходимо перейти от блокировок индивидуальных («физических») объектов базы данных, к блокировке условий (предикатов), которым удовлетворяют эти объекты. Проблема фантомов не возникает при использовании для блокировок уровня таблиц именно потому, что таблица как логический объект представляет собой неявное условие для входящих в него кортежей. Блокировка таблицы – это простой и частный случай предикатной блокировки.

Поскольку любая операция над реляционной базой данных задается некоторым условием (т.е. в ней указывается не конкретный набор объектов базы данных, над которыми нужно выполнить операцию, а условие, которому должны удовлетворять объекты этого набора), идеальным выбором было бы требовать синхронизационную блокировку в режиме S или X именно этого условия. Но если посмотреть на общий вид условий, допускаемых, например, в языке SQL, то становится абсолютно непонятно, как определить совместимость двух предикатных блокировок. Ясно, что без этого использовать предикатные блокировки для сериализации транзакций невозможно, а в общей форме проблема неразрешима.

Один из компромиссных подходов предлагался участниками проекта System R. Подход основывался на том, что при открытии сканирования таблицы по индексу в RSS передается дополнительная информация (диапазон сканирования), которая ограничивает множество кортежей, среди которых не должны возникать фантомы.

Опираясь на наличие этой информации, предлагалось ввести в систему блокировок System R элементы предикатных блокировок. Заметим сначала, что в System R блокировки сегментов (файлов), таблиц и кортежей технически трактовались единообразно, как блокировки идентификаторов кортежей (tid'ов). При блокировке кортежа на самом деле блокировался его tid. При блокировке сегмента или таблицы на самом деле блокировался tid описателя соответствующего объекта во внутренних таблицах-каталогах сегментов или таблиц.

Предлагалось расширить систему синхронизации, разрешив применять блокировки к паре «идентификатор индекса, интервал значений ключа этого индекса». К такой паре можно было применять блокировки в любом из допустимых режимов, причем две такие блокировки считались совместимыми в том и только в том случае, если они были совместимы в соответствии с приведенной таб. 13.2 или указанные диапазоны значений ключей не пересекались.

При наличии такой возможности, если открывается сканирование таблицы через индекс, то таблица блокируется в режиме IS, и в этом же режиме блокируется пара «идентификатор индекса, диапазон сканирования». При занесении (удалении) кортежа таблица блокируется в режиме IX, и в этом же режиме для каждого индекса, определенного на данной таблице отношении, блокируется пара «идентификатор индекса, значение ключа из затрагиваемого операцией кортежа». Это позволяет избежать конфликтов читающих транзакций с теми изменяющими транзакциями, которые затрагивают диапазоны сканирования читающих транзакций. При этом решается проблема фантомов, и параллельность транзакций ограничивается «по существу», т.е. только в тех случаях, когда их параллельное выполнение создает проблемы.

Заметим сразу, что описанное решение проблемы фантомов далеко от идеального. Во-первых, по-прежнему при сканировании таблиц без использования индексов отсутствие фантомов можно гарантировать только при блокировке всего отношения в режиме S. Во-вторых, даже при сканировании по индексу условие реальной выборки кортежа часто может быть гораздо строже простого указания диапазона сканирования, а это значит, что блокировка этого диапазона будет слишком сильной, т.е. затронет более широкое множество кортежей, чем то, которое будет реальным результатом сканирования.

Известно следующее более совершенное решение. Будем называть простым условием конъюнкцию простых предикатов сравнения, имеющих вид имя\_поля { = > < } значение. В типичных СУБД, поддерживающих двухуровневую организацию (языковой уровень и уровень управления внешней памяти), в интерфейсе подсистемы управления памятью (которая обычно заведует и сериализацией транзакций) допускаются только простые условия. Подсистема языкового уровня производит компиляцию оператора SQL со сложным условием в последовательность обращений к подсистеме управления памятью, в каждом из которых содержатся только простые условия.

Более точно, простое условие явно указывается в операции открытия сканирования таблицы (напрямую или через индекс; в последнем случае оно конъюнктивно соединяется с условием, задаваемым диапазоном сканирования). Кроме того, при открытии сканирования всегда можно указать, для какой цели оно будет использоваться: для выборки кортежей, для их удаления или для их обновления (это известно компилятору SQL). Кроме того, неявные условия задаются операциями вставки и удаления кортежей (конъюнктивное логическое выражение, состоящее из простых предикатов вида имя\_поля = значение для всех полей таблицы), а также операциями обновления кортежей (конъюнктивное логическое выражение, состоящее из простых предикатов вида имя\_поля = значение для всех обновляемых полей таблицы). Поэтому в случае типовой организации SQL-ориентированной СУБД простые условия можно использовать как основу предикатных захватов.

Для простых условий совместимость предикатных блокировок легко определяется на основе следующей геометрической интерпретации. Пусть Tab – таблица с полями a1, a2, ..., a*n*, а m1, m2, ..., m*n* – множества допустимых значений a1, a2, ..., a*n* соответственно (естественно, все эти множества – конечные). Тогда можно сопоставить Tab конечное *n*-мерное пространство возможных значений кортежей Tab. Легко видеть, что любое простое условие, представляющее собой конъюнкцию простых предикатов, «вырезает» в этом пространстве *k*-мерный прямоугольник (*k* ≤ n).

Достаточно очевидно следующее утверждение:

Пусть имеются два простых условия scond1 и scond2. Пусть транзакция T1 запрашивает блокировку scond1, а транзакция T2 – scond2 в режимах, которые были бы несовместимы, если бы scond1 и scond2 являлись не условиями, а объектами базы данных (S-X, X-S, X-X). Эти блокировки совместимы в том и только в том случае, когда прямоугольники, соответствующие scond1 и scond2, не пересекаются.

Это утверждение действительно очевидно (каждому *k*-мерному прямоугольнику в *n*-мерном пространстве возможных значений кортежей Tab соответствует некоторое подмножество возможных значений кортежей, и отсутствие пересечения у двух прямоугольников гарантирует отсутствие конфликтов транзакций), но для наглядности на рис. 13.5 приводится иллюстрирующий пример, показывающий, что в каких бы режимах не требовала транзакция T1 блокировки условия (0 < a < 5) & (b = 5), а транзакция T2 – блокировки условия (0 < a <6) & (0 < b <4), эти блокировки всегда будут совместимы.


*Рис. 13.5.* Простые условия, блокировки которых совместимы

Интересно, что при поддержке такой системы блокировок простых условий можно обойтись без гранулированных блокировок. В частности, чтобы гарантированно заблокировать таблицу целиком, достаточно заблокировать условие &1*i*n (min(m*i*) < имя\_поля*i* < max(m*i*)). Чтобы заблокировать базу данных, достаточно заблокировать условие, являющееся конъюнкцией условий блокировки всех таблиц этой базы данных.

Заметим, что блокировки простых условий описываются таблицами, немногим отличающимися от таблиц традиционных синхронизаторов с гранулированными блокировками. Поэтому введение в СУБД механизма предикатных блокировок не приводит к значительным усложнениям.

#### 13.3.2. Синхронизационные тупики, их распознавание и разрушение

Одним из наиболее чувствительных недостатков метода сериализации транзакций на основе синхронизационных блокировок является возможность возникновение тупиков (deadlocks) между транзакциями. Синхронизационные тупики возможны при применении любого из рассмотренных выше вариантов механизмов блокировок.

На рис. 13.6 показан простой сценарий возникновения синхронизационного тупика между транзакциями T1 и T2:


*Рис. 13.6.* Ситуация синхронизационного тупика между транзакциями T1 и T2

* транзакции T1 и T2 устанавливают монопольные блокировки объектов *o*1 и *o*2 соответственно;
* после этого T1 требуется совместная блокировка объекта *o*2, а T2 – совместная блокировка объекта *o*1;
* ни одно из этих требований блокировки не может быть удовлетворено, следовательно, ни одна из транзакций не может продолжаться; поэтому монопольные блокировки объектов никогда не будут сняты, а требования совместных блокировок не будут удовлетворены.

Поскольку тупики возможны, и никакого естественного выхода из тупиковой ситуации не существует, то эти ситуации необходимо обнаруживать и искусственно устранять.

##### Обнаружение тупиковых ситуаций

Основой обнаружения тупиковых ситуаций является построение (или постоянное поддержание) графа ожидания транзакций. Граф ожидания транзакций – это ориентированный двудольный граф, в котором существует два типа вершин – вершины, соответствующие транзакциям (будем изображать их прямоугольниками), и вершины, соответствующие объектам блокировок (будем изображать их окружностями). В этом графе дуги соединяют только вершины-транзакции с вершинами-объектами. Дуга из вершины-транзакции к вершине-объекту существует в том и только в том случае, если для этой транзакции имеется удовлетворенная блокировка данного объекта. Дуга из вершины-объекта к вершине-транзакции существует тогда и только тогда, когда эта транзакция ожидает удовлетворения запроса блокировки данного объекта. Легко показать, что в системе существует тупиковая ситуация в том и только в том случае, когда в графе ожидания транзакций имеется хотя бы один цикл. Простейший пример графа ожидания транзакций с циклом показан на рис. 13.6.

Для распознавания тупиковых ситуаций периодически производится построение графа ожидания транзакций (как уже отмечалось, иногда граф ожидания поддерживается постоянно), и в этом графе ищутся циклы. Традиционной техникой (для которой существует множество разновидностей) нахождения циклов в ориентированном графе является редукция графа.

Пример применения алгоритма редукции к графу ожидания транзакций показан на рис. 13.7 (в целях упрощения примера предполагается, что все блокировки являются монопольными, т.е. для каждой вершины-объекта имеется не более одной входящей дуги). В этом случае редукция состоит в том, что, прежде всего, из графа ожидания (начальное состояние которого показано на рис. 13.7 (a)) удаляются все дуги, исходящие из вершин-транзакций, в которые не входят дуги из вершин-объектов. (Это основывается на том разумном предположении, что транзакции, не ожидающие удовлетворения запроса блокировок, могут успешно завершиться и освободить блокировки). Кроме того, удаляются дуги, входящие в вершины-транзакции, из которых не исходят, ведущие к вершинам-объектам (транзакции, ожидающие удовлетворения блокировок, но не удерживающие заблокированные объекты, не могут быть причиной тупика). Для тех вершин-объектов, для которых не осталось входящих дуг, но существуют исходящие, ориентация одной из исходящих дуг (выбираемой произвольным образом) изменяется на противоположную (это моделирует удовлетворение запроса блокировки). Состояние графа после выполнения первого шага редукции показано на рис. 13.7 (b). После этого снова повторяются описанные действия (cостояние графа после выполнения второго шага редукции показано на рис. 13.7 (c)), и так до тех пор, пока не прекратится удаление дуг. Если в графе остались дуги, то они обязательно образуют цикл (см. рис. 13.7 (c)).


*Рис. 13.7.* Применение алгоритма редукции к графу ожидания транзакций

Предположим теперь, что нам удалось найти цикл в графе ожидания транзакций. Что делать теперь?

##### Разрушение тупиков

Нужно каким-то образом обеспечить возможность продолжения работы хотя бы для части транзакций, попавших в тупик. Разрушение тупика начинается с выбора в цикле транзакций так называемой транзакции-жертвы, т.е. транзакции, которой решено пожертвовать, чтобы обеспечить возможность продолжения работы других транзакций.

Выбрать «жертву» не так уж легко, поскольку для этого могут использоваться различные, зачастую противоречивые критерии. С одной стороны, было бы разумно жертвовать наиболее «богатой» транзакцией, т.е. той транзакцией, которая удерживает наиболее число блокировок объектов. В этом случае после принудительно завершения такой транзакции освободилось бы наибольшее число объектов, что с большой вероятностью привело бы к исчезновению тупиковой ситуации. Но, с другой стороны, «богатая» транзакция, скорее всего, выполнялась дольше других транзакций. На ее выполнение уже затрачено большое количество системных ресурсов и, вероятно, она скоро завершится самостоятельно. Поэтому этот выбор может оказаться в системном отношении не самым удачным.

Можно пожертвовать самой «молодой» транзакцией, которая существует в системе в течение наименьшего времени. Такую транзакцию менее всего жалко, поскольку она еще не успела израсходовать много системных ресурсов. Но, с другой стороны, такая транзакция не могла и накопить много блокировок, и поэтому ее насильственное завершение вряд ли поможет устранить тупиковую ситуацию. Так стоит ли ею жертвовать?

Можно выбрать транзакцию-жертву случайным образом из всех транзакций, попавших в тупик. Возможно, что в среднем этот подход привел бы к хорошим результатам. Но, к сожалению, в нем не учитывается возможная приоритетность транзакций. Было бы не слишком хорошо, например, жертвовать транзакцией, запущенной от имени руководителя организации.

Поэтому обычно при выборе транзакции-жертвы используется многофакторная оценка ее стоимости, в которую с разными весами входят время выполнения, число накопленных блокировок, приоритет и т.д. В качестве «жертвы» выбирает транзакция, для которой эта оценка выдает наиболее подходящий результат.

После выбора транзакции-жертвы выполняется откат этой транзакции, который может носить полный или частичный (до некоторой точки сохранения) характер. При этом, естественно, освобождаются блокировки, и может быть продолжено выполнение других транзакций.

Естественно, такое насильственное устранение тупиковых ситуаций является нарушением принципа изолированности пользователей, которого невозможно избежать.

Заметим, что в централизованных системах стоимость построения графа ожидания сравнительно невелика, но она становится слишком большой в распределенных СУБД, в которых транзакции могут выполняться в разных узлах сети. Поэтому в таких системах обычно используются другие методы сериализации транзакций.

#### 13.3.3. Метод временных меток

Альтернативный метод сериализации транзакций, хорошо работающий в условиях редкого возникновения конфликтов транзакций и не требующий построения графа ожидания транзакций, основан на использовании *временн*ы*х меток*. Основная идея метода временн*ы*х меток (Timestamp Ordering, TO), у которого существует множество разновидностей, состоит в следующем: если транзакция T1 началась раньше транзакции T2, то система обеспечивает такой сериальный план, как если бы транзакция T1 была целиком выполнена до начала T2.

Для этого каждой транзакции T предписывается временн*а*я метка t(T), соответствующая времени начала выполнения транзакции T. При выполнении операции над объектом o транзакция T помечает его своими идентификатором, временн*о*й меткой и типом операции (чтение или изменение).

Перед выполнением операции над объектом o транзакция T2 выполняет следующие действия:

* Проверяет, помечен ли объект o какой-либо транзакцией T1. Если не помечен, то помечает этот объект своей временн*о*й меткой и типом операции и выполняет операцию. Конец действий.
* Иначе транзакция T2 проверяет, не завершилась ли транзакция T1, пометившая этот объект. Если транзакция T1 закончилась, то T2 помечает объект o и выполняет свою операцию. Конец действий.
* Если транзакция T1 не завершилась, то T2 проверяет конфликтность операций. Если операции неконфликтны, то при объекте o запоминается идентификатор транзакции T2, остается или проставляется временн*а*я метка с меньшим значением, и транзакция T2 выполняет свою операцию.
* Если операции транзакций T2 и T1 конфликтуют, то если t(T1) > t(T2) (т.е. транзакция T1 является более «молодой», чем T2), то производится откат T1 и всех других транзакций, идентификаторы которых сохранены при объекте o, и T2 выполняет свою операцию.
* Если же t(T1) < t(T2) (T1 «старше» T2), то производится откат T2; T2 получает новую временн*у*ю метку и начинается заново.

К недостаткам метода TO относятся потенциально более частые откаты транзакций, чем в случае использования синхронизационных захватов. Это связано с тем, что конфликтность транзакций определяется более грубо. Кроме того, в распределенных системах не очень просто вырабатывать глобальные временн*ы*е метки с отношением полного порядка (это отдельная большая наука).

Но в распределенных системах эти недостатки окупаются тем, что не нужно распознавать тупики, а как мы уже отмечали, построение графа ожидания в распределенных системах стоит очень дорого.

#### 13.3.4. Методы сериализации транзакций на основе поддержки версий объектов базы данных

Основная идея алгоритмов сериализации транзакций, описываемых в этом разделе, состоит в том, что в базе данных допускается существование нескольких «версий» одного и того же объекта. Эти алгоритмы, главным образом, направлены на преодоление конфликтов транзакций категорий R/W и W/R, позволяя выполнять операции чтения над некоторой предыдущей версией объекта базы данных. В результате операции чтения выполняются без задержек и тупиков, свойственных механизмам синхронизационных блокировок, а также без некоторых откатов, возможных при применении метода временн*ы*х меток, описанного в предыдущем подразделе.

Алгоритмы управления транзакциями, основанные на поддержке версий, достаточно широко распространены в области SQL-ориентированных СУБД. В частности, подобные алгоритмы используются в СУБД Oracle и PostgreSQL. В дальнейшем в этом подразделе будем называть алгоритмы этой категории *версионными* алгоритмами.

##### Версионный вариант алгоритма временных меток

Одним из наиболее старых и простых версионных алгоритмов является *версионный вариант алгоритма временн*ы*х меток (Multiversion Timestamp Ordering, MVTO)*. Как и в простом методе временн*ы*х меток, описанном в предыдущем подразделе, в алгоритме MVTO порядок выполнения операций одновременно выполняемых транзакций задается порядком временн*ы*х меток, которые получают транзакции во время старта. Временн*ы*е метки также используются для идентификации версий данных при чтении и модификации – каждая версия получает временн*у*ю метку той транзакции, которая ее записала. Алгоритм MVTO не только следит за порядком выполнения операций транзакций, но также отвечает за трансформацию операций над объектами базы данных в операции над версиями этих объектов, т.е. каждая операция над объектом базы данных *o* преобразуется в соответствующую операцию над некоторой версией объекта o.

При описании алгоритма будем использовать следующие обозначения. Как и раньше, временн*у*ю метку, полученную транзакцией Ti в начале ее работы, будем обозначать как t(Ti). Операция чтения объекта базы данных *o*, выполняемая в транзакции Ti, будет обозначаться как Ri(*o*). Для обозначения того, что транзакция Ti читает версию объекта базы данных *o*, созданную транзакцией Tk, будем использовать запись Ri(*o*k). Для обозначения того, что транзакция Ti записывает версию элемента данных *o*, будем использовать запись Wi(*o*i).

Алгоритм MVTO работает следующим образом.

* Любая операция Ri(*o*) преобразуется в операцию Ri(*o*k), где *o*k – это версия объекта *o*, помеченная наибольшей временной меткой t(Tk), такой что t(Tk) t(Ti). Другими словами, транзакции Ti для чтения дается версия объекта *o*, созданная транзакцией Tk, которая не моложе Ti, но старше любой другой транзакции Tn, создававшей свою версию объекта *o*.
* При обработке операции Wi(*o*) выполняются следующие действия:
	+ если к этому времени некоторой незафиксированной транзакцией Tn уже выполнена некоторая операция Rn(*o*k), такая что t(Tk) t(Ti) < t(Tn), то операция Wi(*o*) не выполняется, а транзакция Ti откатывается;
	+ в противном случае Wi(*o*) преобразуется в Wi(*o*i), т.е. образуется еще одна версия объекта *o*.
* При откате любой транзакции уничтожаются все созданные ею версии объектов базы данных и откатываются все транзакции, прочитавшие хотя бы одну из этих версий. Тем самым, откаты транзакций могут быть «каскадными».
* Выполнение операции фиксации транзакции Ti (COMMIT) откладывается до того момента, когда завершатся все транзакции, записавшие версии данных, прочитанные Ti. Легко видеть, что без соблюдения этого требования не соблюдалось бы свойство долговечности (durability) транзакций, поскольку при откате некоторых транзакций потребовалось бы откатывать и ранее зафиксированные транзакции.

Преимущества алгоритма MVTO лучше всего иллюстрируются поведением транзакций T1 и T2 (см. рис. 13.8). При использовании блокировок между ними возник бы синхронизационный тупик, а при использовании обычного метода временных меток одна из транзакций подверглась бы откату. Однако при применении версий такие неприятности не возникают из-за того, что первая транзакция читает «старые» версии объектов *o* и ω.


*Рис. 13.8.* Пример работы алгоритма MVTO

Транзакция T3 ожидает фиксации транзакции T2 перед своим собственным завершением (на рис. 13.8 это показано пунктирной линией). Это происходит потому, что транзакция T3 прочитала версию *o*2 объекта *o*, образованную еще не зафиксированной транзакцией.

Транзакция T4 пытается создать версию ω4 объекта ω после того, как еще не зафиксированная транзакция T5 (начавшаяся позже) уже прочитала более раннюю версию ω4. Поэтому транзакция T5 не сможет «увидеть» изменения объекта ω, произведенные транзакцией T4. Следовательно, сериализация транзакций в порядке получения ими временных меток становится невозможной, и приходится произвести откат транзакции T4.

Итак, основными преимуществами алгоритма MVTO является отсутствие задержек и откатов при выполнении операций чтения, а основным недостатком – возможность возникновение каскадных откатов транзакций при выполнении операций записи. Кроме того, в базе данных может накапливаться произвольное число версий одного и того же объекта, и определение того, какие версии больше не требуются, является серьезной технической проблемой.

##### Версионный вариант двухфазного протокола синхронизационных блокировок

При описании *двухверсионного варианта протокола 2PL (Two-Version Two-Phase Locking Protocol, 2V2PL)* будем называть *текущими* версиями объектов базы данных версии, созданные зафиксированными транзакциями с наиболее поздним временем фиксации; *незафиксированными* версиями *–* версии, созданные еще незавершившимися транзакциями. При следовании протоколу 2V2PL в каждый момент времени существует не более одной незафиксированной версии каждого объекта базы данных.

Операции любой транзакции Ti над объектом базы данных *o* обрабатываются следующим образом:

* операция Ri(*o*) немедленно выполняется над текущей версией объекта *o*;
* операция Wi(*o*), приводящая к созданию новой версии объекта *o*, выполняется только после завершения (фиксации или отката) транзакции, создавшей незафиксированную версию объекта *o*;
* выполнение операции COMMIT откладывается до тех пор, пока не завершатся все транзакции Tk, прочитавшие текущие версии объектов базы данных, которые должны замениться незафиксированными версиями этих объектов, созданными транзакцией Ti.

Для реализации такого поведения используются три типа блокировок:

* RL (Read Lock) – в этом режиме блокируется любой объект базы данных *o* перед выполнением операции чтения его текущей версии; удержание этой блокировки до конца транзакции гарантирует, что при повторном чтении объекта *o* будет прочитана та же версия этого объекта;
* WL (Write Lock) – в этом режиме блокируется любой объект базы данных *o* перед выполнением операции, приводящей к созданию новой (незафиксированной) версии этого объекта; удержание этой блокировки до конца транзакции гарантирует, что в любой момент времени будет существовать не более одной незафиксированной версии любого объекта базы данных;
* CL (Commit Lock) – блокировка устанавливается во время выполнения операции COMMIT транзакции и затрагивает любой объект базы данных, новую версию которого создала данная транзакция; удовлетворение этой блокировки для данной транзакции гарантирует, что завершились все транзакции, читавшие текущие версии объектов, новые версии которых были созданы при выполнении данной транзакции, и, следовательно, их можно заменить.

В таб. 13.3 показаны правила совместимости этих блокировок.

Таблица 9.3. Таблица совместимости «версионных» блокировок

|  |  |  |  |
| --- | --- | --- | --- |
|   | RL(o) | WL(o) | CL(o) |
| RL(o) | да | да | нет |
| WL(o) | да | нет | нет |
| CL(o) | нет | нет | нет |

Как видно, операция чтения может блокироваться только на время фиксации транзакции, заменяющей текущую версию требуемого объекта базы данных. Для выполнения операции записи требуется долговременная монопольная блокировка соответствующего объекта базы данных, которая, однако, в этом случае совместима с блокировкой этого же объекта по чтению (поскольку в действительности блокируются разные версии этого объекта). И, конечно, как и во всех схемах сериализации транзакций на основе блокировок, здесь возможны синхронизационные тупики.

##### Версионно-блокировочный протокол сериализации транзакций для поддержки только читающих транзакций

В заключение обсудим гибридный протокол, поддерживающий эффективное выполнение транзакций, не изменяющих состояние базы данных (*Multiversion Protocol for Read-Only Transactions, ROMV*). При применении этого протокола при образовании каждой транзакции явно указывается ее тип – только читающая (read-only) или изменяющая (update) транзакция. В только читающих транзакциях допускается использование только операций чтения объектов базы данных, а в изменяющих транзакциях – операций и чтения, и записи.

Изменяющие транзакции выполняются в соответствии с обычным протоколом 2PL, т.е. перед выполнением операции чтения или записи объекта базы данных o этот объект должен быть заблокирован в режиме S или X соответственно, и блокировки объектов удерживаются до конца изменяющей транзакции. Каждая операции записи объекта o создает его новую версию, которая при завершении транзакции помечается временн*о*й меткой, соответствующей моменту фиксации этой транзакции.

Каждая только читающая транзакция при своем образовании получает соответствующую временн*у*ю метку. При выполнении операции чтения объекта базы данных o транзакция получает доступ к версии объекта o, образованной изменяющей транзакцией, которая хронологически последней зафиксировалась к моменту образования данной читающей транзакции.

Основным плюсом протокола ROMV по сравнению с ранее описанным протоколом 2V2PL является принципиальное отсутствие синхронизационных задержек при выполнении операций чтения только читающих транзакций. Если сравнивать ROMV с MVTO, то он выигрывает в принципиальном отсутствии откатов только читающих транзакций. Конечно, при работе изменяющих транзакций возможно возникновение синхронизационных тупиков и откатов, и здесь требуется использовать обычные методы распознавания и разрушения тупиков.

Кроме того, при использовании протокола ROMV в базе данных может возникать произвольное число версий объектов. Требуется создание специального сборщика мусора, который должен удалять ненужные версии данных. Простейший сборщик мусора удаляет все неиспользуемые версии, значения временных меток которых меньше значения временной метки старейшей активной только читающей транзакции.

### 13.4. Заключение

В этой лекции описаны основные принципы управления транзакциями в системах управления базами данных, различные методы, алгоритмы и протоколы, способствующие достижению целей управления транзакциями. Следует заметить, что существует достаточно развитая теория управления транзакциями с собственными средствами формализации постановки задач и доказательства корректности алгоритмов. Для обеспечения более простого понимания сути материала в него не включены все эти формализмы.

В лекции описаны два основных подхода к сериализации транзакций – на основе синхронизационных блокировок и временн*ы*х меток. У каждого из этих подходов имеются свои достоинства и недостатки, но на практике существенно больше распространен метод синхронизационных блокировок. В заключение лекции были рассмотрены расширения этих подходов с применением версий объектов базы данных. Соответствующие алгоритмы и протоколы позволяют уменьшить число потенциальных конфликтов транзакций, но для их поддержки требуются дополнительные расходы внешней памяти и усложнение общей архитектуры СУБД.

## Лекция 14. Средства журнализации и восстановления баз данных

### 14.1. Введение

Одним из основных требований к развитым СУБД является надежность хранения баз данных. Это требование предполагает, в частности, возможность восстановления согласованного состояния базы данных после любого рода аппаратных и программных сбоев. Очевидно, что для выполнения восстановлений необходима некоторая дополнительная информация. В подавляющем большинстве современных реляционных СУБД такая избыточная дополнительная информация поддерживается в виде журнала изменений базы данных.

Итак, общей целью журнализации изменений баз данных является обеспечение возможности восстановления согласованного состояния базы данных после любого сбоя. Поскольку основой поддержания целостного состояния базы данных является механизм транзакций, журнализация и восстановление тесно связаны с понятием транзакции. Общими принципами восстановления являются следующие:

* результаты зафиксированных транзакций должны быть сохранены в восстановленном состоянии базы данных (т.е. должно поддерживаться свойство *долговечности (durability)* транзакций);
* результаты незафиксированных транзакций должны отсутствовать в восстановленном состоянии базы данных (в противном случае состояние базы данных могло бы оказаться не целостным).

Это, собственно, и означает, что восстанавливается последнее по времени согласованное состояние базы данных.

Возможны следующие ситуации, при которых требуется производить восстановление состояния базы данных:

* Индивидуальный откат транзакции. Тривиальной ситуацией отката транзакции является ее явное завершение оператором ROLLBACK. Возможны также ситуации, когда откат транзакции инициируется системой. Примерами могут быть возникновение исключительной ситуации в прикладной программе (например, деление на ноль) или выбор транзакции в качестве жертвы при разрушении синхронизационного тупика. Для восстановления согласованного состояния базы данных при индивидуальном откате транзакции нужно устранить последствия операторов модификации базы данных, которые выполнялись в этой транзакции.
* Восстановление после внезапной потери содержимого оперативной памяти (мягкий сбой). Такая ситуация может возникнуть при аварийном выключении электрического питания, при возникновении неустранимого сбоя процессора (например, срабатывании контроля основной памяти) и т.д. Ситуация характеризуется потерей той части базы данных, которая к моменту сбоя содержалась в буферах оперативной памяти СУБД.
* Восстановление после поломки основного внешнего носителя базы данных (жесткий сбой). Эта ситуация при достаточно высокой надежности современных устройств внешней памяти может возникать сравнительно редко, но, тем не менее, СУБД должна быть в состоянии восстановить базу данных даже и в этом случае. Основой восстановления является архивная копия и журнал изменений базы данных.

Во всех трех случаях основой восстановления является хранение избыточных данных. Эти избыточные данные хранятся в журнале, содержащем последовательность записей об изменении базы данных.

Возможны два основных варианта ведения журнальной информации. В первом варианте для каждой транзакции поддерживается отдельный локальный журнал изменений базы данных этой транзакцией. Эти локальные журналы используются для индивидуальных откатов транзакций и могут поддерживаться в основной (правильнее сказать, в виртуальной) памяти СУБД. Кроме того, поддерживается общий журнал изменений базы данных, используемый для восстановления состояния базы данных после мягких и жестких сбоев.

Данный подход позволяет быстро выполнять индивидуальные откаты транзакций, но приводит к дублированию информации в локальных и общем журналах. Поэтому чаще используется второй вариант – поддержка только общего журнала изменений базы данных, который используется и при выполнении индивидуальных откатов. Здесь мы рассматриваем именно этот вариант.

В этой лекции сначала мы проанализируем особенности подсистемы СУБД, управляющей буферами основной памяти, и связь механизмов буферизации и журнализации. Затем на содержательном уровне без технических деталей обсуждим общие принципы журнализации изменений и восстановления целостного состояния базы данных после сбоев, опираясь, в основном на методы, применявшиеся в System R и ее ранних предшественниках.

### 14.2. Буферизация блоков базы данных в основной памяти и ее связь с журнализацией

Журнализация операций изменения базы данных[66)](http://citforum.ru/database/advanced_intro/44.shtml%22%20%5Cl%20%22footnote66) тесно связана не только с управлением транзакциями, но и с буферизацией блоков базы данных в основной памяти. По причинам объективно существующей разницы в скорости работы процессоров и основной памяти и устройств внешней памяти (эта разница в скорости существовала, существует, и будет существовать всегда) буферизация блоков базы данных в основной памяти является единственным реальным способом достижения приемлемой эффективности СУБД. Без поддержки буферизации базы данных СУБД работала бы со скоростью магнитных дисков, т.е. на несколько порядков медленнее, чем если бы обработка данных происходила в основной памяти.

Если бы каждая запись об изменении базы данных, которая должна поступить в журнал при выполнении любой операции обновления базы данных, реально немедленно перемещалась бы во внешнюю память, это привело бы к существенному замедлению работы системы. Фактически, тогда каждая операция обновления базы данных выполнялась бы со скоростью магнитного диска. Поэтому записи в журнал тоже буферизуются: при нормальной работе буфер выталкивается во внешнюю память журнала только при полном заполнении записями. Более точно, для буферизации записей журнала обычно используются два буфера. После полного заполнения первый буфер выталкивается на магнитный диск, и пока совершается этот обмен, журнальные записи размещаются во втором буфере. К моменту конца обмена заполняется второй буфер, он выталкивается во внешнюю память, а журнальные записи снова размещаются в первом буфере и т.д.

Здесь следует заметить, что здесь идет речь об использовании буферов (и базы данных, и журнала), располагающихся именно в физической основной памяти, управляемой непосредственно СУБД, а не виртуальной памяти СУБД, управляемой операционной системой. Использование буферов виртуальной памяти является практически бессмысленным делом, поскольку в этом случае операционная система, руководствуясь своими собственными стратегиями управления основной памяти, в любой момент может удалить буферную страницу СУБД из основной памяти и перенести ее копию во внешнюю память в область свопинга. Тогда при следующей попытке записи СУБД в эту страницу возникнет прерывание, при обработке которого операционная система подкачает страницу в основную память, выполнив совершенно не ожидаемый СУБД обмен с внешней памятью.

Нельзя надеяться на то, что операционная система настолько грамотно управляет основной памятью, что нужные страницы виртуальной памяти СУБД в нужное время будут находиться в основной памяти. Операционная система просто не обладает достаточной информацией, чтобы всегда принимать правильные решения. Правильно управлять своей буферной памятью может только сама СУБД, «отбирающая» у операционной системы часть физической основной памяти для размещения в ней буферов базы данных и журнала.

#### 14.2.1. Управление буферным пулом базы данных

В развитых (вернее сказать, правильно организованных) СУБД поддерживается собственная стратегия замещения страниц буферного пула. Задача, которую решает СУБД, очень похожа на задачу, которую решает операционная система при управлении виртуальной памятью.

В случае операционной системы, если некоторый процесс требует обеспечения доступа к странице виртуальной памяти, отсутствующей в основной памяти, и нет свободных страниц основной памяти, в соответствии с некоторым критерием выбирается некоторая занятая страница основной памяти, освобождается (т.е. изымается из виртуальной памяти какого-то процесса и, может быть, копируется на диск) и подключается к виртуальной памяти запросившего процесса с предварительным считыванием с диска нужных данных.

В случае СУБД, если при выполнении некоторой операции в некоторой транзакции требуется доступ к некоторому блоку базы данных, и копия этого блока отсутствует в буферном пуле, СУБД должна выделить какую-либо страницу буферного пула, считать в нее с диска требуемый блок базы данных и предоставить доступ к этой странице запросившей операции. Конечно, в буферном пуле может не оказаться свободных страниц, и тогда СУБД в соответствии с некоторым критерием находит некоторую занятую страницу, освобождает ее (возможно, выталкивает во внешнюю память).

Основная разница между этими случаями состоит в критерии выборки занятой страницы для «откачки». Не будем обсуждать здесь стратегии замещения страниц, используемые в операционных системах. Заметим лишь, что почти всегда операционная система стремится заменить страницу, к которой предположительно дольше всего не будет обращений, но, поскольку предвидение будущего невозможно, оно аппроксимируется прошлым. В частности, в одном из популярных алгоритмов замещения страниц LRU (Least Recently Used) принимается предположение, что дольше всего в будущем не потребуется та страница, к которой дольше всего не обращались в прошлом.

В стратегии замещения страниц буферного пула СУБД тоже чаще всего используется некоторая разновидность алгоритма LRU. Но, как уже отмечалось выше, СУБД располагает большей информацией о страницах буферного пула, чем операционная система о страницах основной памяти.

Например, если в некоторой транзакции выполняется сканирование некоторой таблицы без использования индекса, и при выполнении операции NEXT был затребован доступ к некоторому блоку базы данных (с соответствующим перемещением копии этого блока в некоторую страницу буферного пула), то подсистема управления буферным пулом «знает», что эта страница еще точно потребуется до тех пор, пока не будет прочитан последний кортеж сканируемой таблицы, располагающийся в данной странице. Более того, СУБД «знает», какой блок базы данных потребуется после завершения просмотра кортежей данного блока, и может заранее переместить его копию в некоторую страницу буферного пула.

Кроме того, некоторые блоки базы данных заведомо требуются чаще других блоков. Например, при любом просмотре таблицы на основе некоторого индекса гарантированно потребуется доступ к корневому блоку соответствующего B-дерева. При вставке кортежа в любую таблицу или удалении из нее кортежа будет необходимо должным образом изменить все определенные для нее индексы, и для этого тоже гарантированно потребуется доступ к корневым блокам всех соответствующих B-деревьев.

Поэтому в стратегии замещения страниц буферного пула базы данных обычно используется алгоритм LRU с приоритетами страниц (грубо говоря, высокоприоритетные страницы стареют, т.е. становятся кандидатами на замещение, медленнее, чем низкоприоритетные страницы). В частности, страницы, содержащие копии корневых блоков индексов, являются настолько высокоприоритетными, что обычно никогда не замещаются. Кроме того, поддерживается предварительное считывание в буферную память копий блоков, доступ к которым вскоре понадобится.

#### 14.2.2. Физическая синхронизация

Поскольку в СУБД может одновременно («параллельно») выполняться несколько транзакций, вполне реальна ситуация, когда в двух одновременно выполняемых операциях требуется доступ к одному и тому же блоку базы данных (т.е. к одной и той же буферной странице, содержащей копию этого блока). Понятно, что в одновременном доступе для чтения содержимого блока ничего плохого нет, но параллельное изменение блока может привести к непредсказуемым результатам.

Следует заметить, что, вообще говоря, координацию параллельного доступа к страницам буферного пула не обеспечивает логическая синхронизация, используемая для сериализации транзакций (см. лекцию 13). Например, предположим, что в двух параллельно выполняемых транзакциях одновременно выполняются операции модификации кортежей, у одного из которых tid = (*n*, 1), а у другого tid = (*n*, 2). Если в СУБД используются блокировки на уровне кортежей, то система допустит параллельное выполнение этих двух операций, и они будут одновременно изменять страницу, содержащую копию блока базы данных с номером *n*. При выполнении обеих операций может потребоваться перемещение кортежей внутри этого блока, и понятно, что в результате ничего хорошего, скорее всего, не получится. Аналогично, логическая синхронизация может легко допустить параллельное выполнение нескольких операций, требующих обновления одного и того же индекса. Некоординированное параллельное обновление B-дерева с большой вероятностью приводит к разрушению его структуры.

Поэтому при выполнении операций уровня RSS необходимо поддерживать дополнительную «физическую» синхронизацию, в которой единицами блокировки служат страницы буферного пула (или блоки) базы данных. В пределах операции перед чтением из страницы буферного пула (блока базы данных) требуется запросить у подсистемы управления буферным пулом блокировку соответствующей страницы (блока) в режиме S, а перед записью в страницу (в блок) – ее блокировку в режиме X. Совместимость блокировок обычная, такая же, как в табл. 13.1.

Но блокировки страниц буферного пула нужны не только для координации параллельного доступа к страницам при параллельном выполнении транзакций. При выполнении операций уровня RSS могут возникать ошибки, обнаруживаемые в середине операции, уже после того, как одна или несколько страниц буферного пула (блоков базы данных) было изменено. Например, может выполняться операция вставки кортежа в некоторую таблицу, нарушающая уникальность некоторого индекса, определенного над этой таблицей. Нарушение уникальности этого индекса будет обнаружено при попытке вставить в него новый ключ, но до этого новый кортеж уже мог быть размещен в блоке данных, и некоторые индексы уже могли быть успешно обновлены.

При обнаружении ошибки операции нужно ликвидировать все ее следы в базе данных и выдать соответствующий код ошибки на уровень RDS. Проще всего сделать это, произведя обратные изменения всех страниц (блоков базы данных), которые были изменены при прямом выполнении операции. Но для этого требуется, чтобы все страницы (блоки базы данных), заблокированные при выполнении операции, оставались заблокированными до конца этой операции.

Тем самым, для подсистемы управления буферным пулом операции уровня RSS являются (почти) тем же, чем являются транзакции для подсистемы управления транзакциями. Достаточным условием корректного выполнения операций является соблюдение двухфазного протокола синхронизационных блокировок над страницами буферного пула в пределах операций.

Заметим (хотя и без подробных объяснений), что это условие не является необходимым. Каждую операцию уровня RSS можно разбить на последовательность «микроопераций» и потребовать соблюдения двухфазного протокола синхронизационных блокировок в пределах микроопераций. Например, операцию INSERT уровня RSS можно разбить на следующие микрооперации:

1) нахождение блока данных для вставки;
2) вставка кортежа в найденный блок;
3) обновление индекса 1;
…….
*n*) обновление индекса *n*,

где *n* – число индексов, определенных для данной таблицы. Общий принцип состоит в том, что в пределах одной микрооперации блокируются все блоки базы данных, которые обязаны быть изменены согласованным образом.

#### 14.2.3. Протокол упреждающей записи в журнал и его связь с буферизацией

Реальная ситуация является более сложной. Имеются два вида буферов – буфера журнала и буферный пул страниц основной памяти, – которые содержат связанную информацию. И те, и другие буфера могут выталкиваться во внешнюю память. Основной причиной выталкивания буфера журнала является его полное заполнение журнальными записями. Страницы буферного пула базы данных чаще всего выталкиваются во внешнюю память, когда требуется переместить в основную память некоторый блок базы данных, а свободных страниц в буферном пуле нет. Тогда срабатывает алгоритм замещения страниц, выбирается страница, содержимое которой, вероятно, дольше всего не потребуется, и эта страница (если ее содержимое изменялось) выталкивается в соответствующий блок внешней памяти базы данных. Проблема состоит в выработке некоторой общей политики выталкивания, которая обеспечивала бы возможность восстановления состояния базы данных после сбоев.

Заметим, что эта проблема не возникает при индивидуальных откатах транзакций, поскольку в этих случаях содержимое основной памяти не утрачено, и при восстановлении можно пользоваться содержимым как буфера журнала, так и буферных страниц базы данных. Но если произошел мягкий сбой, и содержимое буферов утрачено, то для проведения восстановления базы данных необходимо иметь некоторое согласованное состояние журнала и базы данных во внешней памяти.

Основным принципом согласованной политики выталкивания буфера журнала и буферных страниц базы данных является то, что запись об изменении объекта базы данных должна оказаться во внешней памяти журнала раньше, чем измененный объект окажется во внешней памяти базы данных. Соответствующий протокол журнализации (и управления буферизацией) называется WAL (Write Ahead Log, «пиши сначала в журнал») и состоит в том, что если требуется вытолкнуть во внешнюю память буферную страницу, содержащую измененный объект базы данных, то перед этим нужно гарантировать выталкивание во внешнюю память журнала буферной страницы журнала, содержащей запись об изменении этого объекта.

При следовании протоколу WAL, если во внешней памяти базы данных находится некоторый объект базы данных, по отношению к которому выполнена операция модификации, то во внешней памяти журнала обязательно находится запись, соответствующая этой операции. Обратное неверно, т.е. если во внешней памяти журнала содержится запись о некоторой операции изменения объекта базы данных, то сам измененный объект может отсутствовать во внешней памяти базы данных.

Дополнительное условие на выталкивание буферов накладывается тем требованием, что каждая успешно завершенная транзакция должна быть реально зафиксирована во внешней памяти. Какой бы сбой не произошел, система должна быть в состоянии восстановить состояние базы данных, содержащее результаты всех транзакций, зафиксированных до момента сбоя.

Самым простым решением было бы выталкивание буфера журнала, за которым следует массовое выталкивание буферов страниц базы данных, изменявшихся данной транзакцией. Довольно часто так и делают, но это вызывает существенные накладные расходы при выполнении операции фиксации транзакции.

Оказывается, что минимальным требованием, гарантирующим возможность восстановления последнего согласованного состояния базы данных, является выталкивание при фиксации транзакции во внешнюю память журнала всех записей об изменении базы данных этой транзакцией. При этом последней записью в журнал, производимой от имени данной транзакции, является специальная запись о конце транзакции.

Рассмотрим теперь, как можно выполнять операции восстановления базы данных в различных ситуациях, если в системе поддерживается общий для всех транзакций журнал с общей буферизацией записей, поддерживаемый в соответствии с протоколом WAL.

### 14.3. Индивидуальный откат транзакции

Для обеспечения возможности индивидуального отката транзакции по общему журналу все записи в журнале от данной транзакции связываются в обратный список. В начале списка для незавершенных транзакций находится запись о последнем изменении базы данных, произведенном данной транзакцией. Заметим, что в этом случае хронологически последние записи могут быть еще не вытолкнуты во внешнюю память журнала и могут находиться в буфере основной памяти. Для закончившихся транзакций (индивидуальные откаты которых уже невозможны) началом списка является запись о конце транзакции, которая обязательно вытолкнута во внешнюю память журнала, т.е. весь список находится во внешней памяти. Концом списка всегда служит первая запись об изменении базы данных, произведенном данной транзакцией. Обычно в каждой записи проставляется уникальный идентификатор транзакции, чтобы можно было восстановить прямой список записей об изменениях базы данных данной транзакцией.

Итак, индивидуальный откат транзакции (еще раз подчеркнем, что это возможно только для незавершенных транзакций) выполняется следующим образом:

* Выбирается очередная журнальная запись из списка данной транзакции.
* Выполняется противоположная по смыслу операция: вместо операции INSERT выполняется соответствующая операция DELETE, вместо операции DELETE выполняется INSERT, и вместо прямой операции UPDATE – обратная операция UPDATE, восстанавливающая предыдущее состояние объекта базы данных.
* Любая из этих обратных операций также журнализуется. Собственно для индивидуального отката это не нужно, но при выполнении индивидуального отката транзакции может произойти мягкий сбой, при восстановлении после которого потребуется откатить транзакции, для которых не полностью выполнен индивидуальный откат.
* При успешном завершении отката в журнал заносится запись о конце транзакции. С точки зрения журнала такая транзакция является зафиксированной.

[66](http://citforum.ru/database/advanced_intro/44.shtml%22%20%5Cl%20%22footnote66_back) Следует подчеркнуть, что здесь речь идет о логических операциях низкого уровня, т.е. уровня RSS, а не SQL.

### 14.4. Восстановление после мягкого сбоя

К числу основных проблем восстановления после мягкого сбоя относится то, что одна логическая операция изменения базы данных может изменять несколько физических блоков базы данных, например, блок данных и несколько блоков индексов. Блоки базы данных буферизуются в оперативной памяти и выталкиваются независимо. После мягкого сбоя набор блоков внешней памяти базы данных может оказаться несогласованным, т.е. часть блоков внешней памяти соответствует объекту до изменения, часть – после изменения. Например, в результате выполнения операции UPDATE соответствующий кортеж мог переместиться в другой блок. В этом случае (см. лекцию 12) изменяются два блока: в описатель кортежа в его исходном блоке записывается его новый tid, а в новом блоке размещается сам модифицированный кортеж. Очевидно, что если хотя бы один из этих блоков не попал во внешнюю память базы данных к моменту мягкого сбоя, то при восстановлении не удастся вернуть кортеж на его прежнее место. Другими словами, к такому состоянию внешней памяти базы данных не применимы операции логического уровня.

Состояние внешней памяти базы данных называется *физически согласованным*, если наборы страниц всех объектов согласованы, т.е. соответствуют состоянию любого объекта либо после его изменения, либо до изменения.

#### 14.4.1. Схема восстановления от точки физической согласованности

Будем считать, что в журнале отмечаются точки физической согласованности базы данных – моменты времени, в которые во внешней памяти содержатся согласованные результаты операций, завершившихся до соответствующего момента времени, и отсутствуют результаты операций, которые не завершились, а буфер журнала вытолкнут во внешнюю память. Немного позже мы обсудим, как можно достичь физической согласованности. Назовем такие точки ppc (point of physical consistency).

Все возможные состояния транзакций к моменту мягкого сбоя показаны на рис. 14.1.


*Рис. 14.1.* Возможные состояния транзакций к моменту мягкого сбоя

Предположим, что каким-то образом удалось восстановить внешнюю память базы данных к состоянию на момент времени tppc (как это можно сделать, обсудим в следующем подразделе). Тогда восстановление последнего по времени логически целостного состояния базы данных производится следующим образом.

* Для транзакции T1 никаких действий производить не требуется. Она закончилась до момента *tppc*, и все ее результаты гарантированно отражены во внешней памяти базы данных.
* Для транзакции T2 нужно повторно выполнить (*redo*) последовательность операций, которые выполнялись после установки точки физически согласованного состояния в момент *tppc*. Действительно, во внешней памяти полностью отсутствуют следы операций, которые выполнялись в транзакции T2 после момента *tppc*. Следовательно, повторное прямое (по смыслу и хронологии) выполнение операций транзакции T2 корректно и приведет к логически согласованному состоянию базы данных. (Поскольку транзакция T2 успешно завершилась до момента мягкого сбоя *tfs*, в журнале содержатся записи обо всех изменениях базы данных, произведенных этой транзакцией.)
* Для транзакции T3 нужно выполнить в обратном направлении (undo) ту часть операций, которую она успела выполнить до момента *tppc*. Действительно, во внешней памяти базы данных полностью отсутствуют результаты операций T3, которые были выполнены после момента *tppc*. С другой стороны, во внешней памяти гарантированно присутствуют результаты операций T3, которые были выполнены до момента *tppc*. Следовательно, обратное выполнение (по смыслу и хронологии) операций T3 корректно и приведет к согласованному состоянию базы данных. (Поскольку транзакция T3 не завершилась к моменту мягкого сбоя *tfs*, при восстановлении необходимо устранить все последствия ее выполнения.)
* Для транзакции T4, которая успела начаться после момента *tppc* и закончиться до момента мягкого сбоя *tfs*, нужно произвести полное повторное выполнение операций в прямом направлении. (Поскольку транзакция T4 успешно завершилась до момента мягкого сбоя *tfs*, в журнале содержатся записи обо всех изменениях базы данных, произведенных этой транзакцией).
* Наконец, для транзакции T5, начавшейся после момента *tppc* и не успевшей завершиться к моменту мягкого сбоя *tfs*, никаких действий предпринимать не требуется. Результаты операций этой транзакции полностью отсутствуют во внешней памяти базы данных.

#### 14.4.2. Восстановление физической согласованности базы данных

Каким же образом можно обеспечить наличие точек физической согласованности базы данных, т.е. как восстановить состояние базы данных в момент *tppc*? Для этого используются два основных подхода: подход, основанный на использовании теневого механизма, и подход, в котором применяется журнализация постраничных изменений базы данных.

##### Теневой механизм

Теневой механизм был изначально предложен для поддержания целостности файлов при аварийном отключении питания компьютера. Общая идея теневого механизма для файлов показана на рис. 14.2. Файл представляется как набор блоков внешней памяти, для доступа к которым поддерживается таблица отображения (см. лекцию 1). При открытии файла таблица отображения номеров его логических блоков в адреса физических блоков внешней памяти считывается в оперативную память. При модификации любого блока файла во внешней памяти выделяется новый блок. При этом текущая таблица отображения (в основной памяти) изменяется, а теневая остается неизменной. Если во время работы с открытым файлом происходит сбой, во внешней памяти автоматически сохраняется состояние файла до его открытия. Для явного восстановления файла достаточно повторно считать в основную память теневую таблицу отображения.


*Рис. 14.2.* Теневой механизм для файлов

В контексте базы данных теневой механизм используется следующим образом [[3.16]](http://citforum.ru/database/advanced_intro/86.shtml#ref.3.16). Периодически выполняются операции установки точки физической согласованности базы данных. При выполнении этой операции все логические операции завершаются, все страницы буферного пула базы данных, содержимое которых отличается от содержимого соответствующих блоков внешней памяти, выталкиваются. Теневая таблица отображения файлов (сегментов) базы данных заменяется текущей таблицей отображения (правильнее сказать, текущая таблица отображения записывается на место теневой).

Здесь имеется некоторая проблема, состоящая в том, что в любой момент времени теневая таблица отображения должна быть корректной, т.е. соответствовать некоторому ранее зафиксированному физически целостному состоянию базы данных. Для этого необходимо обеспечить атомарность операции замены теневой таблицы отображения. В общем случае таблица отображения может занимать несколько блоков внешней памяти, и для записи текущей таблицы отображения на место теневой таблицы в этом случае потребуется несколько обменов с дисками. Если в промежутке между этими обменами возникнет мягкий сбой, то будет благополучно утрачена текущая таблица отображения и безнадежно испорчена теневая таблица, т.е. мы просто лишимся возможности восстанавливаться за счет использования последнего физически согласованного состояния базы данных.

Чтобы это не произошло, во внешней памяти поддерживаются две области хранения таблицы отображения файлов (будем называть их областями A и B). Кроме того, в отдельном блоке внешней памяти хранится флаг F, показывающий, какая из этих областей в данный момент содержит действующую теневую таблицу отображения (назовем соответствующие значения флага FA и FB). Тогда, если сохраненным во внешней памяти значением флага является FA, то текущая таблица отображения записывается в область B. Если эта операция выполняется успешно, то в блок флага записывается значение FB. Считается, что операция записи одного блока на диск является атомарной. Если эта операция заканчивается успешно, это означает, что новая теневая таблица отображения хранится в области B. Если же запись текущей таблицы отображения в область B не удалась, или если не выполнилась операция записи блока с флагом F, то продолжает действовать старая теневая таблица отображения.

Восстановление хронологически последнего сохраненного физически согласованного состояния базы данных происходит мгновенно: текущая таблица отображения заменяет теневой таблицей (при восстановлении просто считывается действующая теневая таблица отображения). Все проблемы восстановления решаются, но за счет слишком большого перерасхода внешней памяти. В пределе может потребоваться вдвое больше внешней памяти, чем реально нужно для хранения базы данных.

##### Журнализация постраничных изменений

Возможен другой подход, при использовании которого наряду с логической журнализацией операций изменения базы данных производится журнализация постраничных изменений. Первый этап восстановления после мягкого сбоя состоит в постраничном откате недовыполненных логических операций. Подобно тому, как это делается с логическими записями по отношению к транзакциям, последней записью о постраничных изменениях от одной логической операции является запись о конце операции. Вообще, выполнение логических операций уровня RSS носит транзакционный характер. В частности, как уже отмечалось выше, при выполнении логической операции обновления базы данных, вообще говоря, изменяется несколько блоков базы данных. Для обеспечения возможности отката отдельной операции (а это может потребоваться, например, если обнаруживается нарушение свойства уникальности какого-либо индекса) приходится до конца операции монопольно блокировать все страницы буферного пула базы данных, содержащие копии изменяемых этой операцией блоков базы данных.

Чтобы распознать, нуждается ли страница внешней памяти базы данных в восстановлении, при выталкивании любой страницы из буферного пула основной памяти в нее помещается номер последней записи о постраничном изменении этой страницы. Этот же номер запоминается в самой записи. Тогда, чтобы понять, нужно ли применить данную запись о постраничном изменении соответствующего блока внешней памяти для восстановления состояния этого блока, требуется всего лишь сравнить номер, содержащийся в этом блоке, с номером, содержащимся в журнальной записи. Если в блоке содержится номер, меньший номера журнальной записи, то это означает, что буферная страница, в которой выполнялось соответствующее изменение, не была к моменту мягкого сбоя вытолкнута во внешнюю память, и применять данную запись для восстановления соответствующего блока внешней памяти не требуется.

Для иллюстрации на рис. 14.3 показано пять записей об изменении блока b с номерами *n*-2, *n*-1, *n*, *n*+1, *n*+2. В блоке b содержится номер *n*. Это означает, что в состоянии блока отражены результаты операций изменения блока, соответствующих журнальным записям LR(b)*n*, LR(b)*n*-1 и LR(b)*n*-2. Изменения блока, произведенные операциями, которым соответствуют две хронологически последние журнальные записи LR(b)*n*+1 и LR(b)*n*+2, в его состоянии во внешней памяти не отражены, поскольку не было выполнено выталкивание во внешнюю память страницы буферного пула, содержащей копию блока b. Поэтому при восстановлении состояния блока требуется выполнить обратные операции изменения блока b, соответствующие журнальным записям LR(b)*n*, LR(b)*n*-1 и LR(b)*n*-2.


*Рис. 14.3.* Нумерация записей об изменении блока

В этом подходе имеются два поднаправления. В первом поднаправлении поддерживается общий журнал логических и страничных операций. Естественно, наличие двух видов записей, интерпретируемых абсолютно по-разному, усложняет структуру журнала. Кроме того, записи о постраничных изменениях, актуальность которых носит локальный характер, существенно (и не очень осмысленно) увеличивают журнал.

Поэтому распространено поддержание отдельного (короткого) журнала постраничных изменений. Такой журнал обычно называют *физическим* журналом, поскольку он содержит записи об изменении физических объектов – блоков внешней памяти. В отличие от этого, журнал логических операций принято называть *логическим* журналом, поскольку в нем содержатся записи об операциях над логическими объектами – кортежами.

Как уже отмечалось, логический и физический журналы имеют разную природу. Во-первых, логический журнал должен поддерживать как обратное выполнение журнализованных операций (*undo*), так и их повторное прямое выполнение (*redo*). В отличие от этого, от физического журнала требуется только поддержка обратного выполнения постраничных операций.

Во-вторых, логический журнал обычно начинает заполняться заново только после выполнения операций резервного копирования базы данных или архивирования самого журнала (см. следующий подраздел). До этого времени он линейно растет. Понятно, что в любом случае для размещения журнала выделяется внешняя память ограниченного размера. Предельный размер журнала определяется администратором базы данных и должен согласовываться с размером интервала времени, через которое производится резервное копирование базы данных.

Потенциальное переполнение логического журнала регулируется следующим образом. На пути к достижению максимально возможного размера журнала устанавливаются «желтая» и «красная» зоны. Когда записи в журнал достигают «желтой» зоны, выдается предупреждение администратору базы данных и прекращается образование новых транзакций. Если все существующие транзакции завершаются до достижения «красной» зоны, автоматически выполняется архивация базы данных или логического журнала. Если какие-то транзакции не успевают завершиться до достижения «красной» зоны журнала, выполняется их аварийный откат, после чего производится архивация базы данных или журнала. Естественно, размер «желтой» и «красной» зон логического журнала должен устанавливаться администратором базы данных с учетом максимально допустимого числа одновременно существующих транзакций и их возможной протяженности.

В отличие от этого, физический журнал существует сравнительно недолгое время (интервал времени между соседними операциями установки точки физической согласованности базы данных) и, как правило, занимает существенно меньшее дисковое пространство, чем логический журнал. При выполнении операции установки точки физической согласованности выполняются следующие действия:

* прекращают инициироваться новые логические операции;
* после завершения всех выполняемых логических операций происходит выталкивание во внешнюю память всех модифицированных страниц буферного пула;
* формируется и выталкивается во внешнюю память логического журнала специальная запись о точке физически согласованного состояния;
* в случае успешного предыдущего действия разрешается инициация новых логических операций, и физический журнал пишется заново.

Предпоследняя операция является атомарной (это опять же запись одного блока на диск): если она успешно выполняется, то при следующем восстановлении после мягкого сбоя будет использоваться новая точка физически согласованного состояния, иначе ситуация воспринимается как мягкий сбой с восстановлением логически согласованного состояния базы данных от предыдущей точки физически согласованного состояния (с оповещением об этом администратора базы данных).

### 14.5. Восстановление базы данных после жесткого сбоя

Понятно, что для восстановления последнего согласованного состояния базы данных после жесткого сбоя журнала изменений базы данных явно недостаточно. Самый простой способ восстановления основывается на использовании логического журнала и архивной копии базы данных.

Восстановление начинается с обратного копирования (на исправный носитель) базы данных из архивной копии. Затем для всех закончившихся транзакций выполняется *redo*, т.е. операции повторно выполняются в прямом смысле.

Более точно, происходит следующее:

* по журналу в прямом направлении выполняются все операции;
* для транзакций, которые не закончились к моменту сбоя, выполняется откат.

Очевидно, что после этого будет получено хронологически последнее до момента жесткого сбоя логически согласованное состояние базы данных.

Следует заметить, что при некоторой дисциплине выполнения логических операций над базой данных при восстановлении базы данных после жесткого сбоя можно просто последовательно повторно выполнять операции в соответствии с журнальными записями, не обращая внимание на то, в каких транзакциях они выполнялись до жесткого сбоя. В частности, если сериализация транзакций основывается на блокировках объектов, то эта дисциплина заключается в том, что при выполнении операции в штатном режиме нужно сначала дождаться удовлетворения блокировки изменяемого объекта, затем поместить запись в буфер логического журнала, и только после этого реально выполнять операцию.

На самом деле, поскольку жесткий сбой не сопровождается утратой буферов оперативной памяти, можно восстановить базу данных до такого уровня, чтобы можно было продолжить даже выполнение незавершенных транзакций. Но обычно это не делается, потому что восстановление после жесткого сбоя – это достаточно длительный процесс.

Хотя к ведению журнала предъявляются особые требования по части надежности, в принципе возможна и его утрата. Тогда единственным способом восстановления базы данных является возврат к архивной копии. Конечно, в этом случае не удастся получить последнее согласованное состояние базы данных, но это лучше, чем ничего.

Последний вопрос, который здесь следует еще раз обсудить, касается производства архивных копий базы данных и/или журнала. Самый простой способ состоит в архивировании базы данных по явному указанию администратора или при переполнении журнала. Но можно выполнять архивацию базы данных реже, чем переполняется журнал. Вместо базы данных можно архивировать сам журнал. В пределе для полного восстановления базы данных после жесткого сбоя достаточно иметь исходную архивную копию базы данных, последовательность архивных копий журналов и последний логический журнал. Может показаться, что восстановление базы данных на основе таких архивных источников будет занимать недопустимо большое время, однако здесь возможна значительная оптимизация.

Во-первых, архивированный логический журнал можно сжимать. Для этого для каждого объекта базы данных нужно найти последовательность журнальных записей, относящихся к этому объекту, в хронологическом порядке и заменить их одной записью, соответствующей операции над объектом, результат которой эквивалентен результату последовательного выполнения журнализованных операций из построенной последовательности. Например, на рис. 14.4 показан процесс сжатия последовательности журнальных записей, соответствующих последовательности операций над кортежем, у которого tid = *k* и имеются четыре целочисленных поля. Заметим, что если хронологически последней в последовательности является запись, соответствующая операции DELETE, то после сжатия этой последовательности она станет пустой.


*Рис. 14.4.* Процесс сжатия последовательности журнальных записей

Во-вторых, точно таким же образом можно совместно сжать два хронологически последовательных полных или сжатых журнала. Таким образом, для восстановления после жесткого сбоя можно воспользоваться исходной архивной копией, одним сжатым архивным журналом и последним логическим журналом. Снова могут возникнуть сомнения относительно сложности и продолжительности процесса сжатия журнала. Но здесь следует заметить, что эта работа может выполняться на отдельном компьютере в режиме off-line. Кроме того, если имеется архивная копия базы данных, сжатый архивный журнал и набор еще не сжатых архивных журналов, то этого уже достаточно для восстановления, так что сроки завершения процесса полного сжатия не являются критическими.

### 14.6. Заключение

В этой лекции рассматривались основные принципы и алгоритмы подсистем СУБД, предназначенных для управления буферами основной памяти, журнализации и восстановления базы данных после различных сбоев. Изложение велось без технических деталей, таких как возможные структуры данных журналов.

Заметим, что во многих современных производственных СУБД журнализация и восстановление основаны на применении семейства алгоритмов ARIES, разработанных в 1980-е гг. известным исследователем из компании IBM К. Моханом (C. Mohan). В этой лекции не приводится описание алгоритмов семейства ARIES, поскольку, по мнению автора, это перегрузило бы ее подробностями, не способствующими пониманию основных идей. Тем не менее, читателям, которых заинтересовала эта тема, полезно познакомится с этими алгоритмами, для чего можно воспользоваться, например, [[3.16]](http://citforum.ru/database/advanced_intro/86.shtml#ref.3.16) или оригинальными статьями Мохана и его коллег.