Российская Академия Наук

Институт Системного Программирования

На правах рукописи

Плешачков Петр Олегович

Методы управления транзакциями в XML-ориентированных СУБД

05.13.11 – математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей

ДИССЕРТАЦИЯ

на соискание ученой степени кандидата физико-математических наук

Научный руководитель
доктор технических наук
Кузнецов Сергей Дмитриевич

Москва 2006

Содержание

Введение **4**

1 Управление транзакциями и технологии XML **10**

1.1 Обзор методов управления транзакциями в СУБД . . . 10

1.1.1 Методы обеспечения изоляции параллельных транзакций . . . 11

1.1.2 Методы обеспечения атомарности и надежности транзакций . . . 18

1.2 Платформа XML . . . 23

1.2.1 Расширяемый язык разметки - XML . . . 23

1.2.2 Язык запросов XQuery . . . 24

1.2.3 Язык модификаций XUpdate . . . 27

1.3 XML-ориентированные СУБД . . . 28

1.3.1 РСУБД с поддержкой XML . . . 28

1.3.2 Прирожденные XML-СУБД . . . 37

1.4 Выводы . . . 43

2 Существующие методы управления параллельными XML-транзакциями **45**

2.1 XML-транзакции в реляционных СУБД . . . 45

2.1.1 Блокировки в РСУБД для XML-документов при использовании отображения XML-документов на отношения . . . 46

2.1.2 Блокировки в РСУБД для XML-документов при использовании типа XML или метода STORED . . . 49

2.2 XML-транзакции в прирожденных XML-СУБД . . . 50

2.2.1 Основные приложения XML-СУБД . . . 50

2.2.2 Обзор родственных работ по изоляции XML-транзакций . . . 51

2.3 Выводы . . . 60

3 Протокол изоляции XML-транзакций XDGL **62**

3.1 Введение . . . 62

3.2 Основные определения и обозначения 64
3.3 Семантические особенности языков XQuery/XUpdate 66
3.3.1 Путевые выражения 66
3.3.2 Запросы на XQuery 67
3.3.3 Операция вставки новых узлов 67
3.3.4 Операция удаления узлов 68
3.3.5 Операция переименования узлов 68
3.4 XDGL-блокировки 68
3.4.1 Структурные блокировки 68
3.4.2 Предикатные блокировки 72
3.4.3 Логические блокировки 72
3.5 XDGL-планировщик 74
3.6 Обоснование корректности протокола XDGL 77
3.7 Дополнительные оптимизации в XDGL 83
3.8 Примеры использования протокола XDGL 84
3.9 Выводы 85

4 Управление XML-транзакциями в реляционных СУБД 87

4.1 Многоуровневые модели транзакций и их применение для управления XML-транзакциями в РСУБД 87
4.2 Применение XDGL для изоляции транзакций в РСУБД 89
4.2.1 Поддержка описывающей схемы в SXTM 90
4.3 Атомарность XML-транзакций в двухуровневой модели 93
4.4 Индивидуальные откаты транзакций и восстановление базы данных после сбоев 97
4.5 Повышение параллелизма внутри XML-транзакций 99
4.6 Экспериментальная оценка семантического менеджера управления XML-транзакциями 103
4.6.1 Экспериментальная установка 103
4.6.2 Эксперимент 1: накладные расходы 104
4.6.3 Эксперимент 2: пропускная способность 106
4.6.4 Эксперимент 3: время отклика 107
4.6.5 Эксперимент 4: время отклика транзакций при использовании параллелизма внутри транзакций 108
4.7 Выводы 109

5 Управление транзакциями в прирожденных XML-СУБД 110

5.1 Требования к управлению транзакциями в прирожденных XML-СУБД 110

5.2 Снимки базы данных и их применение для изоляции читающих и изменяющих транзакций 116

5.3 Продвижение снимков 119

5.4 Отображение логических версий на физические версии 120

5.5 Адресация версий и идентификация страниц из снимков базы данных 122

5.6 Изоляция T^w транзакций 126

5.6.1 Изоляция T^w транзакций на уровне блоков 128

5.6.2 Изоляция T^w транзакций на основе протокола XDGL 134

5.7 Метод восстановления транзакций после мягких сбоев в системе 142

5.7.1 Физический журнал 142

5.7.2 Логический журнал 143

5.7.3 Контрольные точки базы данных 144

5.7.4 Индивидуальный откат транзакции 148

5.7.5 Восстановление базы данных после сбоя 149

5.8 Экспериментальная оценка методов управления XML-транзакциями 151

5.8.1 Эксперимент 1: пропускная способность 151

5.8.2 Эксперимент 2: время отклика 153

5.9 Выводы 155

Заключение 156

Список литературы 158

Введение

Актуальность темы

В настоящее время язык XML используется как основное средство для представления слабоструктурированных данных. Накоплены огромные массивы XML-данных, для которых необходимы развитые средства управления. Это стимулировало бурное развитие XML-ориентированных СУБД, позволяющих управлять XML-данными. XML-ориентированные СУБД разделяются на два класса: СУБД с поддержкой XML (как правило, это реляционные СУБД), в которых поддержка XML реализована над существующей моделью данных, и прирожденные XML-СУБД, спроектированные изначально с учетом XML-модели данных. Важнейшим компонентом в этих системах является подсистема управления XML-транзакциями, которая должна обеспечивать изолированность, атомарность и надежность транзакций, выполняемых над хранимыми XML-данными. При этом в СУБД с поддержкой XML, как правило, существует некоторый механизм управления транзакциями, а в прирожденных XML-СУБД этот механизм должен разрабатываться с нуля.

Средства изоляции транзакций, имеющиеся в СУБД с поддержкой XML, плохо подходят для XML-данных, поскольку в них не учитывается иерархическая структура XML-данных и семантика XML-операций. В частности, использование средств изоляции транзакций, разработанных в рамках реляционных систем, приводит к большому количеству искусственных конфликтов между транзакциями. Иными словами, во многих ситуациях система фиксирует конфликт между конкурентными транзакциями и выполняет их последовательно, хотя на логическом уровне конфликта между транзакциями нет. Это в свою очередь негативно влияет на пропускную способность и время отклика транзакций в СУБД. Эти факторы стимулируют разработку новых средств изоляции транзакций для СУБД с поддержкой XML, учитывающих в полной мере специфику XML-модели данных. При этом в новом механизме должен учитываться уже существующий в СУБД механизм изоляции транзакций, и разумно использовать некоторые возможности существующего механизма при реализации нового.

Механизмы изоляции транзакций в большинстве прирожденных XML-СУБД

находятся, в лучшем случае, в зачаточном состоянии, что затрудняет практическое использование этих СУБД. При разработке механизмов изоляции транзакций необходимо учитывать как особенности XML-модели данных, так и то требование, что механизм изоляции транзакции в прирожденных XML-СУБД должен обеспечивать неконфликтное выполнение читающих и изменяющих транзакций. Последнее требование объясняется тем, что во многих случаях прирожденные XML-СУБД используются как среда выполнения длинных приложений, написанных на функциональном языке XQuery. Необходимо, чтобы выполнение этих приложений не приводило к блокировке изменяющих транзакций.

Другой важной проблемой прирожденных XML-СУБД является обеспечение свойств атомарности и надежности транзакций, поскольку организация внешней памяти и обработка XML-данных в оперативной памяти в этих системах существенно отличаются от аналогов, применяемых в реляционных базах данных.

Решение этих проблем и определяет актуальность диссертационной работы.

Цель и задачи работы

Целью диссертационной работы является исследование и разработка методов управления транзакциями в XML-ориентированных СУБД. Для достижения этой цели были поставлены следующие задачи:

1. Разработка универсального протокола изоляции XML-транзакций для XML-ориентированных СУБД, в полной мере учитывающего специфику XML-модели данных.

2. Разработка метода управления конкурентными XML-транзакциями в СУБД с поддержкой XML на основе универсального протокола изоляции XML-транзакций.

3. Разработка метода изоляции XML-транзакций в прирожденных XML-СУБД, учитывающего как специфику XML-модели данных, так и специфику использования прирожденных XML баз данных.

4. Разработка методов обеспечения атомарности и надежности транзакций в прирожденных XML-СУБД.

Основные результаты работы

В рамках диссертационной работы получены следующие результаты:

1. Разработан универсальный протокол XDGL изоляции XML-транзакций, который обеспечивает полную сериализацию XML-транзакций и учитывает семантику XML-модели данных при определении конфликтов между конкурентными XML-транзакциями.

2. На основе семантического протокола изоляции XML-транзакций разработан механизм управления конкурентными XML-транзакциями для реляционных СУБД с поддержкой XML.

3. Разработан версионный протокол 4VXDGL изоляции XML-транзакций для прирожденных XML-СУБД, в котором учитываются как семантика XML-модели данных, так и физические особенности организации структур внешней и оперативной памяти в прирожденных XML-СУБД. Протокол 4VXDGL также учитывает специфику использования прирожденных XML-СУБД, обеспечивая безконфликтное выполнение читающих и изменяющих транзакций.

4. Произведена экспериментальная оценка предложенных методов изоляции XML-транзакций для реляционной СУБД MS SQL Server 2005 и прирожденной XML-СУБД Седна, которая демонстрирует существенное увеличение производительности системы при наличии параллельных потоков транзакций на чтение и модификацию XML-документов.

5. Разработаны методы обеспечения атомарности и надежности транзакций в прирожденных XML-СУБД.

Научная новизна работы

Научной новизной обладают следующие результаты диссертационной работы:

1. Разработан оригинальный протокол изоляции XML-транзакций XDGL для XML-ориентированных СУБД, основанный на описывающей схеме XML-документа.

2. Предложен оригинальный семантический метод управления XML-транзакциями для реляционных СУБД с поддержкой XML на основе двухуровневой модели транзакций.

3. Разработан оригинальный версионный протокол 4VXDGL изоляции XML-транзакций для прирожденных XML-СУБД, в котором учитываются как особенности организации внешней памяти, так и методы управления оперативной памятью в этих СУБД; протокол позволяет выполнять читающие и изменяющие транзакции без взаимных блокировок.

4. Разработаны оригинальные методы обеспечения атомарности и надежности транзакций в прирожденных XML-СУБД, опирающиеся на механизм версионности данных.

Практическая значимость

Разработанные методы могут применяться в XML-ориентированных СУБД, для которых требования согласованности и надежности данных играют определяющую роль. Разработанный протокол изоляции XML-транзакций XDGL может применяться в широком классе систем управления XML-данными и позволяет существенно увеличить параллелизм транзакций в системе, обеспечивая, тем самым, более высокую производительность системы. Кроме того, разработанные методы обеспечения надежности и атомарности XML-транзакций могут служить основой для создания прирожденных XML-СУБД.

На основе предложенных методов и подходов разработаны: (1) прототип семантического менеджера управления XML-транзакциями SXTM над реляционной СУБД MS SQL Server 2005, (2) прототип менеджера управления XML-транзакциями, используемый в качестве основы для создания в ИСП РАН промышленной прирожденной XML-СУБД Седна.

Доклады и печатные публикации

Основные положения работы докладывались на девятой международной конференции в области баз данных и информационных систем (ADBIS 2005), на тринадцатом итальянском симпозиуме по базам данных (SEBD 2005), на двадцать второй британской национальной конференции по базам данных (BNCOD 2005), на симпозиуме аспирантов на двадцать третьей британской национальной конференции по базам данных (PhD Forum BNCOD 2006), на первом и втором весеннем коллоквиуме молодых исследователей в области баз данных и информационных систем (SYRCoDIS 2004, SYRCoDIS 2005), на сто третьем семинаре Московской Секции ACM SIGMOD (2005 г).

По материалам диссертации опубликовано восемь печатных работ [1, 2, 3, 4, 5, 6, 7, 8].

Структура и объем диссертации

Работа состоит из введения, пяти глав, заключения и списка литературы. Общий объем диссертации 166 старниц. Список литературы содержит 100 наименований.

Краткое содержание работы

Первая глава является обзорной и содержит изложение принципов, методов и средств, лежащих в основе разработок, которые описываются в последующих четырех главах. Глава состоит из трех разделов. В первом разделе делается обзор существующих методов управления транзакциями. Особое внимание уделяется обзору методов сериализации транзакций. Рассматриваются протоколы сериализации, основанные на блокировках, временных метках, а также версионные протоколы. Во втором разделе делается обзор технологий платформы XML в контексте задачи управления слабоструктурированными данными, в частности, рассматривается язык запросов XQuery. В третьем разделе рассматриваются два класса систем управления XML-данными: реляционные СУБД (РСУБД) с поддержкой XML и прирожденные XML-СУБД. Рассматриваются различные методы хранения XML в РСУБД, обсуждаются их достоинства и недостатки. Кроме того, делается обзор физических особенностей организации прирожденной XML-СУБД Седна.

Вторая глава посвящена рассмотрению существующих методов управления параллельными XML-транзакциями. Глава состоит из двух разделов. В первом разделе рассматриваются ограничения и недостатки существующего блокировочного механизма в РСУБД для управления XML-транзакциями при использовании различных схем хранения XML-документов. Во втором разделе рассматриваются существующие методы поддержки конкурентных транзакций, ориентированные на применение в прирожденных XML-СУБД.

Третья глава содержит описание разработанного автором протокола XDGL изоляции конкурентных XML-транзакций. Вводится набор блокировок, отражающих специфику операций чтения и изменения XML-документов, описывается специальная компактная структура данных (описывающая схема), на которой предлагается устанавливать блокировки, а также вводится формальное описание и обоснование протокола XDGL.

В четвертой главе рассматривается предложенный автором метод управления XML-транзакциями в РСУБД. Описывается двухуровневая модель XML-транзакций, при которой исходная XML-транзакция разбивается на набор независимых субтранзакций над РСУБД. Демонстрируется, что введение семантического уровня изоляции конкурентных транзакций на основе протокола XDGL приводит к существенному повышению производительности РСУБД. Кроме того, описывается метод обеспечения атомарности и надежности XML-транзакций в двухуровневой модели.

В пятой главе описывается предложенный автором метод управления транзакциями в прирожденной XML-СУБД. Вводится понятие снимка базы данных и рассматривается изоляция читающих и изменяющих XML-транзакций на основе двух снимков базы данных. Описывается версионный протокол 4VXDGL изоляции XML-транзакций,

который использует семантические XDGL-блокировки для изоляции изменяющих транзакций, и снимки базы данных для изоляции читающих и изменяющих транзакций. Обосновывается корректность протокола 4VXDGL и доказываются основные свойства этого протокола. Кроме того, рассматривается метод обеспечения атомарности и надежности XML-транзакций, основанный на использовании снимков базы данных. Использование снимков базы данных для обеспечения надежности позволяет существенно сократить размер журнала, поскольку не требуется заносить записи, требующиеся для восстановления физически согласованного состояния базы данных. Наконец, демонстрируется экспериментальная оценка предложенных методов.

В заключении перечисляются основные результаты работы.

Глава 1

Управление транзакциями и технологии XML

Настоящая глава является обзорной и содержит изложение методов и средств, которые лежат в основе разработок, описываемых в следующих главах. Глава делится на три части. В первой части дается обзор основных методов управления транзакциями в современных СУБД. Во второй части обсуждаются базовые составляющие платформы XML. Наконец, в третьей части приводится обзор основных методов поддержки XML в реляционных СУБД и прирожденных XML-СУБД.

1.1 Обзор методов управления транзакциями в СУБД

Важнейшим фундаментальным понятием в современных системах управления базами данных (СУБД) является транзакция. Транзакция - это неделимая единица работы над базой данных. Выделяются четыре основные свойства транзакций: атомарность, согласованность, изоляция и долговечность. Мы будем обозначать эти свойства аббревиатурой АСИД в соответствии с первыми буквами названий соответствующих свойств. Свойство атомарности означает, что либо результаты всех операторов, входящих в транзакцию, отображаются в базе данных (БД), либо воздействие этих операторов полностью отсутствует. Свойство согласованности означает, что транзакции переводят одно согласованное состояние базы данных в другое без обязательной поддержки согласованности во всех промежуточных точках[1]. Свойство изолированности означает, что выполняющиеся транзакции “не видят друг друга”. Это подразумевает, что, если даже будет запущено множество конкурирующих друг с другом транзакций, любое обновление определенной

[1]Мы здесь приводим это свойство транзакций для полноты и в дальнейшем мы не будем обсуждать свойство согласованности транзакций.

транзакции будет скрыто от остальных до тех пор, пока эта транзакция не завершится. Свойство долговечности означает, что когда транзакция выполнена, ее обновления сохраняются, даже если в следующий момент произойдет сбой в системе.

В соответствии с вышеописанными свойствами транзакций можно разделить на группы методы обеспечения этих свойств. Первая группа состоит из методов обеспечения изоляции параллельных транзакций, а вторая группа состоит из методов обеспечения атомарности и долговременности транзакций. К настоящему времени разработано достаточно большое количество алгоритмов, методов и подходов а рамках каждой из этих групп. Описанию этих техник и посвящены следующие несколько подразделов.

Здесь важно заметить, что идеи параллелизма, как и идеи атомарности и долговременности транзакций, в некоторой степени не зависят от того, является ли СУБД реляционной или какой-либо другой. Однако большая часть как теоретической, так и практической работы в этой области выполнялась именно в контексте реляционных СУБД.

1.1.1 Методы обеспечения изоляции параллельных транзакций

Одним из основных требований к современным СУБД является поддержка мульти режима транзакций, который означает возможность одновременной обработки в СУБД нескольких транзакций с доступом к одним и тем же данным в одно и то же время. Как известно, в такой системе для корректной обработки транзакций без возникновения конфликтных ситуаций необходимы методы контроля выполнения параллельных транзакций. Без использования таких методов в СУБД могут возникать такие ситуации, как потеря результатов обновления, грязное чтение, неповторяющееся чтение и фантомы [10].

В реализации метод управления параллельными транзакциями определяет поведение *планировщика*. Основная задача планировщика заключается в своевременном обнаружении и разрешении конфликтов между выполняющимися транзакциями. После того как конфликт был обнаружен, СУБД должна выбрать метод его разрешения. По методам разрешения конфликтов планировщики можно разделить на те, которые используют в качестве основного метода разрешения конфликтов блокировки, те, которые откатывают одну из конфликтующих транзакций и, наконец, те, которые используют версионные механизмы для разрешения конфликтов.

Далее мы рассмотрим наиболее известные и широко применяемые планировщики. Основными источниками здесь служат книги Бернштейна [11] и Вайкума [9]. Кроме того, детальный обзор протоколов синхронизации можно найти в дипломной работе П. Чардина [13].

Двухфазный протокол синхронизации

Двухфазный протокол синхронизации (Two Phase Locking, 2PL) использует блокировки для разрешения конфликтов между конкурентными транзакциями. Этот протокол достаточно широко применяется в коммерческих СУБД.

Далее мы будем использовать обозначение $W_i(x)$ для операции изменения элемента данных x, транзакцией T_i. Аналогично, $R_i(x)$ - для операции чтения x. В том случае, когда мы не желаем указывать конкретный тип операции, мы будем использовать обозначения $P_i(x)$ и $Q_i(x)$.

Каждая такая операция поступает на вход 2PL-планировщику. В зависимости от типа операции и уже обработанных запросов он определяет, выполнить ли операцию или же заблокировать транзакцию до того момента, когда конфликт будет разрешен.

В 2PL операция записи конфликтует с любой операцией другой транзакции над тем же элементом данных. Операции чтения совместимы между собой. При выборе сценария поведения планировщик руководствуется следующими правилами:

1. Когда 2PL-планировщик получает запрос на операцию $P_i(x)$, он проверяет, конфликтует ли данная операция хотя бы с одной из запланированных ранее. Если нет, то транзакция становится обладателем соответствующего захвата, и операция выполняется. Если да, то выполнение транзакции приостанавливается до тех пор, пока она не сможет получить доступ к требуемому элементу.

2. Если планировщик захватил элемент x в каком-либо режиме доступа, то он не может отпустить захват как минимум до тех пор, пока физическая операция, соответствующая запросу не завершится.

3. Если планировщик отпустил хотя бы один захват, принадлежащий транзакции T_i, то эта транзакция больше не может захватывать никаких объектов БД.

Первые два правила защищают транзакции от совместного доступа к данным в конфликтующих режимах. Третье правило принято называть фазовым. От него, кстати говоря, и происходит название протокола. Каждая транзакция логически разделяется на две фазы. Во время первой фазы она берет захваты на те объекты, которые ей требуются, а во время второй отпускает их по мере завершения работы с ними.

Именно третье правило обеспечивает логическую корректность работы параллельных транзакций. Фактически, оно обеспечивает сериализуемость допускаемых протоколом планов. Транзакции, обрабатываемые 2PL-планировщиком, сериализуются в порядке завершения. Иными словами, результат их работы эквивалентен последовательному выполнению в том порядке, в котором они завершились.

Важным частным случаем протокола 2PL является строгий двухфазный протокол синхронизации. Он отличается от 2PL только тем, что блокировки транзакциями могут сниматься только при фиксации транзакции. Как правило, строгий двухфазный протокол обозначают символом S2PL (Strict Two Phase Locking).

Протокол, основанный на временных метках

Следующий рассматриваемый нами протокол сериализует транзакции по временной метке их старта. Согласно этому протоколу, каждая транзакция получает временную метку в момент старта. Подсистема управления транзакциями присваивает эту метку каждой последующей операции этой транзакции.

Основным правилом работы планировщика, основанного на временных метках, является правило "временного упорядочивания" (Timestamp Ordering rule, TO rule): Для каждой пары конфликтующих операций $P_i(x)$ и $Q_j(x)$ операция P_i выполняется раньше Q_j тогда и только тогда когда временная метка P_i меньше временной метки Q_j.

Простейший TO-планировщик обрабатывает транзакции следующим образом:

1. Если для операции $Q_i(x)$ имеется операция $P_j(x)$, уже выполненная на этот момент, такая, что временная метка T_j (здесь и далее мы будем использовать обозначение $ts(T_j)$) больше чем $ts(Q_i)$, то для транзакции T_i должен быть выполнен откат, и затем она может быть запущена заново с новой временной меткой.

2. В противном случае операция выполняется. Информация об этом запоминается во внутренних структурах планировщика, который должен учитывать этот факт при последующей работе (например, при обработке операции другой транзакции над этим элементом данных).

Этот протокол очевидным образом допускает лишь сериализуемые планы. Однако на практике он фактически не используется из-за очень высокой степени откатов при сильной нагрузке. При этом важно отметить, что чаще всего это "воображаемые" конфликты. TO-планировщики слишком строги. Они накладывают слишком жесткие ограничения на порядок поступления операций, чтобы гарантировать сериализуемость. В реальности многие из операций, вызывающих откат согласно TO-правилу, не нарушают сериализуемости генерируемого плана.

Таким образом, протокол имеет скорее теоретический, чем практический интерес, в отличие от 2PL, который очень широко используется в индустрии. Однако различные гибридные протоколы (например многоверсионный протокол MVTO), использующие TO-планировщик в качестве базы, широко используются на практике.

Версионные методы

В этом разделе мы рассмотрим наиболее известные версионные алгоритмы. Мы будем следовать изложению, предложенному в [9]. Сначала мы рассмотрим версионную модификацию алгоритма на временных метках (MVTO), затем обсудим версионный вариант двухфазного протокола синхронизации и его упрощенную версию (в которой число версий ограничивается двумя). В конце раздела приводится комбинированный алгоритм (ROMV), предназначенный для минимизации ожидания читающих (read-only) транзакций .

Версионный протокол, основанный на временных метках (MVTO) MVTO-планировщик (Multiversion Timestamp Ordering) обрабатывает операции таким образом, чтобы суммарный результат выполнения операций был эквивалентен последовательному выполнению транзакций. При этом, как и в неверсионном TO, их порядок задается порядком временных меток, которые транзакции получают во время старта. Временные метки также используются для идентификации версий данных при чтении и модификации – каждая версия получает временную метку той транзакции, которая ее создала. Планировщик не только следит за порядком выполнения действий транзакций, но также отвечает за трансформацию операций над элементами базы данных в операции над версиями, т.е. каждая операция вида “прочитать элемент данных x”, должна быть преобразована планировщиком в операцию “прочитать версию у элемента данных x”.

Для описания алгоритма введем ряд обозначений. Временную метку, полученную транзакцией T_i в начале ее работы, будем обозначать как $ts(T_i)$. Операция чтения транзакцией T_i элемента данных x будет обозначаться как $r_i(x)$. Для обозначения того, что транзакция T_i читает версию элемента данных x, созданную транзакцией T_k, будем писать $r_i(x_k)$. Для обозначения того, что транзакция T_i записывает версию элемента данных x, будем использовать запись $w_i(x)$.

Теперь опишем алгоритм работы MVTO-планировщика:

1. Планировщик преобразует операцию $r_i(x)$ в операцию $r_i(x_k)$, где x_k – это версия элемента x, помеченная наибольшей временной меткой $ts(T_k)$, такой что $ts(T_k) \leq ts(T_i)$.

2. Операция $w_i(x)$ обрабатывается планировщиком следующим образом.

   (a) Если планировщик уже обработал действие вида $r_j(x_k)$, такое что $ts(T_k) < ts(T_i) < ts(T_j)$, то операция $w_i(x)$ отменяется, а T_i откатывается.

   (b) В противном случае $w_i(x)$ преобразуется в $w_i(x_i)$.

3. Фиксация транзакции T_i откладывается до того момента, когда завершатся все транзакции, которые записывали версии данных, прочитанные T_i.

Заметим, что последний шаг нужен только в том случае, когда мы хотим предотвратить “грязное” чтение.

Многоверсионный двухфазный протокол синхронизации В этом разделе мы рассмотрим вариант двухфазного протокола синхронизации транзакций (2PL), адаптированный для применения в версионной базе данных.

Рассматривая алгоритм работы MVTO-планировщика, мы использовали ряд терминов, которые теперь определим строго. Будем различать три типа версий элемента данных.

1. Зафиксированные версии (commited versions). Эти версии, созданные теми транзакциями, которые уже успешно закончили свою работу.

2. Текущая версия (current version). Это последняя из зафиксированных версий.

3. Незафиксированные версии (uncommited versions). Это версии, созданные теми транзакциями, которые еще находятся в работе.

В MV2PL планировщик следит за тем, чтобы в каждый момент времени существовало не более одной незафиксированной версии. В зависимости от того, позволяется ли транзакциям читать незафиксированные версии данных, различаются два варианта этого алгоритма. Мы будем одновременно рассматривать оба варианта MV2PL (с чтением незафиксированных данных и без него). Сначала рассмотрим схему работы MV2PL в предположении, что все версии элементов данных сохраняются в базе. Затем обсудим вариант этого алгоритма, в котором допускается одновременное существование не более двух версий одного и того же элемента данных.

Все операции, которые обрабатывает планировщик, разделяются на два класса: обычные операции и финальные операции. Под финальной операцией понимается последняя операция транзакции перед ее завершением или же сама операция фиксации (commit). Обе интерпретации допустимы. При этом каждая отдельная операция транзакции обрабатывается следующим образом.

1. Если операция не является финальной, то:

   (a) операция $r(x)$ выполняется незамедлительно; ей сопоставляется последняя из зафиксированных к данному моменту версий x (или последняя из незафиксированных версий x во втором варианте алгоритма);

   (b) операция $w(x)$ выполняется только после фиксации транзакции, записавшей последнюю версию x.

2. Если операция является финальной для транзакции T_i, то она откладывается до тех пор, пока не завершатся следующие транзакции:

   (a) все транзакции T_j, прочитавшие текущую версию данных, которую должна заменить версия, записанная T_i (таким образом устраняется возможность неповторяющегося чтения).

   (b) все транзакции T_j, которые записали версии, прочитанные T_i (это требуется только во втором варианте алгоритма).

Как видно, этот алгоритм ничего не говорит о числе версий одного и того же элемента, которые могут одновременно существовать в базе данных. Это вызывает проблемы с хранением версий. Во-первых, они могут занимать слишком много места (легко вообразить ситуацию, когда объем старых версий становится в несколько раз больше, чем объем всей "текущей" базы данных). Во-вторых, возникают трудности с размещением этих "старых" данных. Учитывая, что количество версий заранее не известно, сложно придумать эффективную структуру их хранения, которая бы не вызывала заметных накладных расходов. И, наконец, такая система слишком сложна в реализации.

Вероятно, именно из-за этих проблем на практике чаще используется протокол 2V2PL, предложенный впервые в работе Байера [68]. В этом протоколе возможно одновременное существование двух версий элемента данных: одной текущей версии данных и не более одной незафиксированной. Такая организация версий выгодна, прежде всего, для транзакций, выполняющих операцию чтения.

В 2V2PL используются три типа блокировок. Каждая блокировка удерживается до конца транзакции:

1. rl (read lock): эта блокировка устанавливается на текущую версию элемента данных x непосредственно перед ее прочтением;

2. wl (write lock): блокировка устанавливается перед тем, как создать новую (незавершенную) версию элемента x;

3. cl (commit lock): блокировка устанавливается перед выполнением последней операции транзакции (обычно перед операцией commit) по отношению к каждому элементу данных, который она записала. Эта блокировка играет роль монопольной блокировки для 2PL. Она необходима для корректной смены версий.

Таблица совместимости блокировок для протокола 2V2PL изображена на рисунке 1.1.

	rl(x)	wl(x)	cl(x)
rl(x)	+	+	-
wl(x)	+	-	-
cl(x)	-	-	-

Рис. 1.1: Таблица совместимости блокировок в протоколе 2V2PL

Очевидно, что использование блокировок rl и wl обеспечивает выполнение правил 1(a) и 1(b) алгоритма MV2PL (с учетом того, что одновременно позволяется поддерживать не более одной незавершенной версии). Блокировка cl, в свою очередь, обеспечивает выполнение правил 2(a) и 2(b).

Многоверсионный протокол для транзакций, не изменяющих данные (ROMV) В работе многих приложений преобладают транзакции, не изменяющие данные (read-only transactions). Такие приложения считывают и анализируют большие объемы данных. В случае наличия хотя бы небольшого числа параллельно выполняемых транзакций, производящих изменения, компонент, который отвечает за управление параллельными транзакциями, должен обеспечить согласованность прочитанных данных. В случае использования алгоритмов планирования без поддержки версий такие долговременные транзакции могут привести к чрезвычайному падению производительности системы. Например, при использовании 2PL станет крайне велика вероятность блокировки транзакций, производящих обновления данных. В результате у этих транзакций будет очень большое время отклика.

Многоверсионные алгоритмы позволяют избежать подобных проблем благодаря тому, что транзакция, вносящая изменения в базу данных, не конфликтует с транзакциями чтения. Но в то же время многоверсионные алгоритмы обычно сложны в реализации, и запросы к версионным СУБД вызывают заметные накладные расходы.

Рассматриваемый в этом разделе протокол ROMV (Multiversion Protocol for Read-Only Transactions) является гибридным, основанным на MVTO и 2PL. Он ориентирован на приложения, для которых наиболее важна скорость выполнения транзакций, не производящих изменений данных. Заметим что, в литературе отсутствует согласие по поводу названий алгоритмов. Так, Пол Боббер и Майкл Кэри в [66] называют обсуждаемый здесь протокол MV2PL. Мы следуем терминологии, принятой в книге Вайкума и Воссена [9]. В ней под термином MV2PL понимается протокол, рассмотренный в предыдущем разделе.

ROMV-планировщик разделяет все транзакции во время их создания на две группы – запросы (queries) и транзакции, изменяющие данные (update transactions). Соответственно, транзакции разных типов обрабатываются по-разному. Такой протокол оказывается проще

в реализации и позволяет получить большую часть выгод, предоставляемых версионными протоколами, которые описаны выше.

Транзакции обрабатываются следующим образом.

1. Транзакции, модифицирующие данные, выполняются в соответствием с протоколом S2PL. Этот протокол является вариантом 2PL. В нем все монопольные блокировки отпускаются лишь в конце транзакции. Каждая операция, изменяющая элемент данных, создает новую версию этого элемента. При завершении транзакции каждая такая версия помечается временной меткой, соответствующей времени завершения транзакции.

2. Запросы (read-only transactions) обрабатываются подобно тому, как это происходит в протоколе MVTO. Каждому запросу также ставится в соответствие временная метка. Но в данном случае она соответствует времени начала транзакции. При выборе версии для чтения запрос выбирает последнюю, завершенную к моменту старта запроса версию.

1.1.2 Методы обеспечения атомарности и надежности транзакций

В этом разделе мы приводим краткий обзор методов обеспечения атомарности и надежности транзакций при возможном возникновении сбоев в системе.

Далее мы будем подразумевать, что при возникновении сбоя теряются данные только из энергозависимой памяти (обычно это оперативная память), но при этом данные во внешней памяти (дисках, магнитных лентах) не теряются. Иными словами мы будем рассматривать мягкие сбои [9][2]. Рассмотрение жестких сбоев в системе, когда повреждается внешняя память, выходит за рамки нашей работы.

Основная задача процедуры восстановления после сбоев заключается в том что: (1) результаты зафиксированных транзакций должны быть сохранены в восстановленном состоянии базы данных (redo recovery); (2) результаты незафиксированных транзакций должны отсутствовать в восстановленном состоянии базы данных (undo recovery).

Основой идеей восстановления является избыточное хранение данных. Эти избыточные данные хранятся в журнале, содержащем последовательность записей об изменении базы данных. Как правило, в СУБД существует один общий журнал, в который заносятся записи об изменении базы данных всех транзакций.

Для журнала в оперативной памяти отводится специальная область для буферизации, поскольку если бы запись об изменении базы данных, которая должна поступить в

[2] Далее термин сбой без указания его типа будет обозначать мягкий сбой.

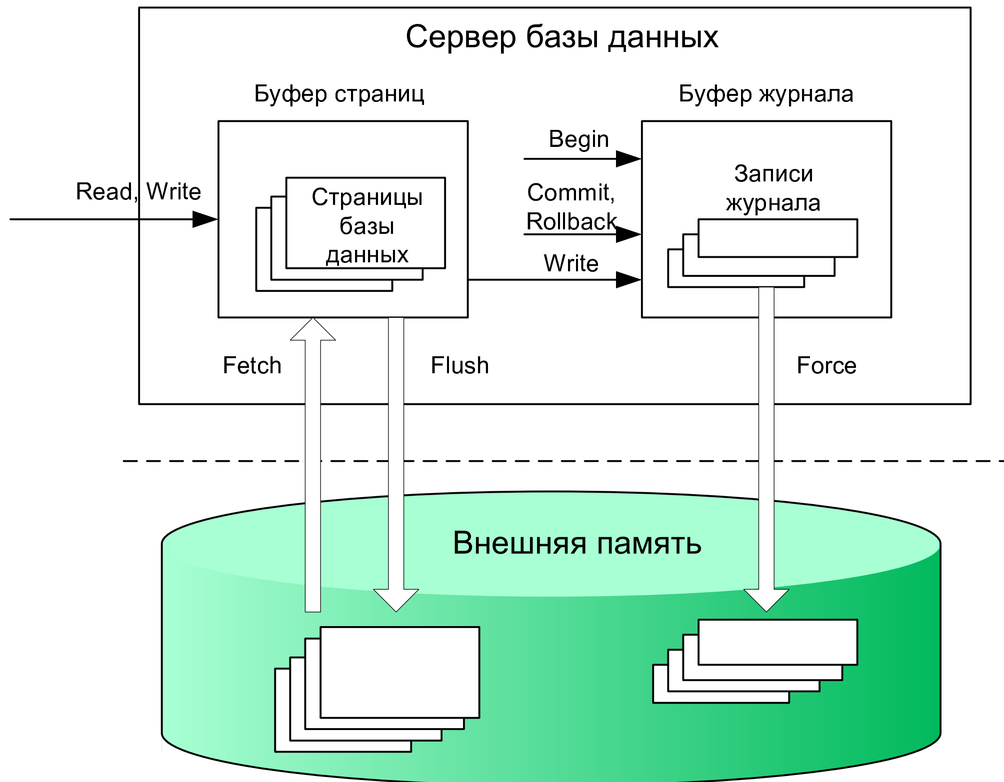


Рис. 1.2: Архитектура компонентов СУБД, относящихся к процедуре восстановления после сбоев

журнал при выполнении любой операции модификации базы данных, реально немедленно записывалась бы во внешнюю память, это привело бы к существенному замедлению работы системы. Таким образом, при нормальной работе очередная страница выталкивается во внешнюю память журнала только при полном заполнении записями.

Итак, имеются два вида буферов - буфер журнала и буфер страниц оперативной памяти, которые содержат связанную информацию. И те, и другие буфера могут выталкиваться во внешнюю память. На рисунке 1.2 графически изображены взаимосвязи между этими компонентами.

Проблема состоит в выработке некоторой общей политики выталкивания буферов, которая обеспечивала бы возможности восстановления состояния базы данных после сбоев.

Основным принципом согласованной политики выталкивания буфера журнала и буферов страниц базы данных является то, что запись об изменении элемента базы данных должна попадать во внешнюю память журнала раньше, чем измененный элемент оказывается во внешней памяти базы данных. Соответствующий протокол журнализации (и управления буферизацией) называется Write Ahead Log (WAL) [89] - “пиши сначала в журнал”, и состоит в том, что если требуется вытолкнуть во внешнюю память измененный

элемент базы данных, то перед этим нужно гарантировать выталкивание во внешнюю память журнала записи о его изменении.

Другими словами, если во внешней памяти базы данных находится некоторый элемент 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 элемента базы данных, то сам измененный элемент может отсутствовать во внешней памяти базы данных.

WAL-протокол гарантирует, что после сбоя во внешней памяти журнала есть вся необходимая информация для произведения отката всех незавершившихся транзакций (undo recovery). Однако это правило не гарантирует, что после сбоя в журнале будет необходимая информация для проведения "наката" (redo) зафиксированных транзакций. Поэтому требуется еще одно условие на политику вытеснения записей журнала: перед фиксацией транзакции во внешнюю память должны быть вытеснены все записи журнала, относящиеся к этой транзакции (включая запись о фиксации транзакции).

Индивидуальный откат транзакций

Для того, чтобы можно было выполнить по журналу индивидуальный откат транзакции, все записи в журнале от данной транзакции связываются в обратный список. Началом списка для незакончившихся транзакций является запись о последнем изменении базы данных, произведенном данной транзакцией. Для закончившихся транзакций (индивидуальные откаты которых уже невозможны) началом списка является запись о конце транзакции, которая обязательно вытолкнута во внешнюю память журнала. 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 выполняется следующим образом:

- Выбирается очередная 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 журнализуются (такие записи называются

компенсационными). Непосредственно для индивидуального отката это не нужно, но при выполнении индивидуального отката транзакции может произойти мягкий сбой, при восстановлении после которого потребуется откатить такую транзакцию, для которой не полностью выполнен индивидуальный откат.

- При успешном завершении отката в журнал заносится запись о конце транзакции. С точки зрения журнала такая транзакция является зафиксированной.

Восстановление после мягкого сбоя

К числу основных проблем восстановление после мягкого сбоя относится то, что одна логическая операция изменения базы данных может изменять несколько физических блоков базы данных, например, страницу данных и несколько страниц индексов. Страницы базы данных буферизуются в оперативной памяти и выталкиваются независимо. Несмотря на применение протокола WAL, после мягкого сбоя набор страниц внешней памяти базы данных может оказаться несогласованным, т.е. часть страниц внешней памяти соответствует объекту до изменения, часть - после изменения. К такому состоянию объекта не применимы операции логического уровня. Поэтому перед тем, как применять логические операции, необходимо некоторым образом восстановить физически целостное состояние базы данных. Состояние внешней памяти базы данных называется физически согласованным, если наборы страниц всех объектов согласованы, т.е. соответствуют состоянию объекта либо после его изменения, либо до изменения.

Существует два основных метода восстановления физической согласованности базы данных: теневой механизм и журнализация изменений внутри каждой страницы (page-oriented logging). Первый метод использовался в System R [89], а второй метод используется в очень популярном на данный момент алгоритме восстановления - ARIES [45]. Далее мы рассмотрим оба этих метода восстановления.

Восстановление на основе теневого механизма Основная идея теневого механизма заключается в следующем. При открытии файла базы данных 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 оперативной памяти) изменяется, а теневая - сохраняется 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 оперативную память теневую таблицу отображения.

В контексте базы данных теневой механизм используется следующим образом. Периодически выполняются операции установления контрольных точек (checkpoints в System R). Для этого все логические операции завершаются, все буфера оперативной памяти, содержимое которых не соответствует содержимому соответствующих страниц внешней памяти, выталкиваются. Теневая таблица отображения файлов базы данных заменяется на текущую (правильнее сказать, текущая таблица отображения записывается на место теневой).

Физическое восстановление в этом случае происходит следующим образом: текущая таблица отображения заменяется на теневую (при восстановлении просто считывается теневая таблица отображения). После этого получается состояние базы данных как на момент сбоя. Тогда по логическим операциям журнала можно откатить все транзакции, которые начались до установки контрольной точки, но не завершились до сбоя, и восстановить транзакции, которые завершились после установки контрольной точки.

Основным недостатком метода является излишний перерасход памяти. Кроме того, при восстановлении с теневой копии приходится перевыполнять большое количество изменений, которые и так присутствовали во внешней памяти. Всех этих недостатков удается избежать в ARIES.

Алгоритм ARIES В ARIES все изменения производимые над блоками базы данных при выталкивании на внешний носитель, заменяют предыдущее значение этого блока (update in-place). Поэтому для восстановления как физической, так и логической согласованности БД используется журнал. Выделяется два типа записей в журнале: (1) логическая запись, которая содержит информацию о производимой операции, (2) страничная запись, которая содержит информацию об изменениях производимых в конкретной странице. Первый тип записей используется для отката (undo), а второй тип записей - для наката (redo).

Процесс восстановления выглядит следующим образом. Сначала проходит стадия анализа, во время которой на основе журнальной информации собираются сведения, необходимые для восстановления. В основном эта работа связана с определением тех транзакций, действия которых необходимо повторить, и тех, действия которых нужно отменить. После этого следуют стадии повторения и отмены. Обычно во время повторения в базу данных вносятся все изменения, которые были внесены в нее транзакциями, успевшими к моменту сбоя завершиться, а отмена, соответственно, заключается в том, чтобы отменить действия, совершенные транзакциями, нуждающимися в откате. В ARIES имеется некоторое отличие от этой схемы, а именно то, что на стадии повторения ARIES повторяет все действия, которые были отражены в журнале к моменту сбоя. То есть после этой стадии база данных находится в том же состоянии, в каком она была в момент сбоя. После этого уже происходит

откат всех незавершенных транзакций.

Как и в System R, в ARIES время от времени выполняются контрольные точки. Однако в ARIES эта операция гораздо более эффективная, поскольку она не требует синхронного сброса всех буферов на диск и не приостанавливает на это время все текущие транзакции на изменение. Вместо этого в журнал записывается информация о грязных страницах в базе данных, а также таблица активных транзакций. На основе этих данных при восстановлении определяется точка в журнале, начиная с которой необходимо производить стадию повторения.

Отметим, что на стадии повторения необходимо определять, есть ли изменения в базе данных, относящиеся к данной записи или нет. Для этого в заголовке самой страницы хранится идентификатор записи журнала, которая описывает последнее изменение в данной страницы. В результате, сравнивая эти идентификаторы, удается узнать, есть ли рассматриваемые изменения на странице или нет.

Существуют и другие технические нюансы. Их описание можно найти в обзорной статье П. Чардина [43].

1.2 Платформа XML

В данном разделе дается обзор основных технологий платформы XML, необходимых для изложения последующего материала.

1.2.1 Расширяемый язык разметки - XML

Расширяемый язык разметки XML (Extensible Markup Language) [14] является подмножеством стандартного обобщенного языка разметки SGML (Standard Generalized Markup Language) [15]. Первоначально XML был разработан для представления информационных ресурсов в Web, в частности, как замена языка гипертекстовой разметки HTML (Hyper Text Markup Language) [16]. Одним из основных достоинств XML является его расширяемость, т.е. имеется возможность вводить новые тэги разметки, что невозможно в HTML. Однако применение XML не ограничивается только Web. В настоящее время XML широко используется в качестве средства обмена данными между различными приложениями в сети. Кроме того, XML стал стандартом де-факто для представления слабоструктурированных данных [54, 55]. В настоящей работе язык XML представляет интерес именно как формат представления слабоструктурированных данных.

На рисунке 1.3 изображен пример XML-документа. XML-документ состоит из различных элементов разметки и непосредственно содержимого документа - данных,

```
<books>
  <book isbn="1111">
     <title>Unix Network Programming</title>
     <price>38</price>
  </book>
  <book isbn="2222">
    <title>Computer Networks</title>
    <price>29</price>
  </book>
</books>
```

Рис. 1.3: Пример XML-документа

представленных в текстовой форме. Тэги предназначены для определения элементов документа, их атрибутов и других конструкций языка. XML-документ определяется в спецификации языка как объект данных, который является правильно сформулированным (well-formed) в соответствии с требованиями спецификации XML. Все эти требования достаточно просты и естественны. Например, требуется, чтобы каждому открывающему тэгу соответствовал закрывающий тэг.

На сегодняшний день консорциумом W3C выпущена первая версия языка - XML 1.0, которая имеет статус рекомендованной.

Важной частью спецификации XML является язык описания схемы XML-документов - DTD (Document Type Definition). Фактически, при помощи DTD можно накладывать ограничения на структурный вид XML-документа. Для наложения более сложных ограничений существует более мощный язык описания схем XML Schema [17, 18].

1.2.2 Язык запросов XQuery

Основным языком запросов к XML-данным является язык запросов XQuery [21]. На сегодняшний день процесс стандартизации XQuery практически завершен, и в скором времени будет выпущена первая официальная версия языка. С появлением XQuery большинство компаний, производящих коммерческие реляционные СУБД, заявило о введении дополнительных расширений для поддержки стандартов платформы XML (включая XQuery) в ближайших версиях своих продуктов.

Базовой частью языка XQuery являются путевые выражения (location path). Путевое

выражение в XQuery основывается на синтаксисе языка XPath [20] и состоит из серии шагов, разделенных символом слэш ("/"). Результатом каждого шага является последовательность узлов. Результатом вычисления путевого выражения является последовательность узлов, формируемая на последнем шаге. Каждый шаг определяется тремя компонентами: *осью* (axis), *тестом* (test) и *предикатом* (predicate). Синтаксически шаг записывается следующим образом: `<ось>::<тест>[<предикат>]`. Каждый шаг вычисляется в контексте, сформированном на предыдущем шаге. Вычисление шага для узла (будем называть этот узел текущим) осуществляется в несколько этапов, на каждом из которых строится промежуточная последовательность.

На первом этапе происходит перемещение от текущего узла по иерархии узлов в направлении, определяемом осью. В XQuery поддерживается 12 осей: `child` (последовательность дочерних узлов), `descendant` (последовательность всех узлов потомков), `parent` (отец узла), `attribute` (последовательность узлов-атрибутов), `self` (текущий узел), `descendant-or-self` (последовательность всех потомков, дополненная самим узлом), `ancestor` (последовательность всех узлов предков), `ancestor-or-self` (последовательность всех узлов предков, дополненная самим узлом), `following` (последовательность всех последующих узлов), `preceeding` (последовательность всех предшествующих узлов), `following-sibling` (последовательность всех последующих узлов братьев) и `preceeding-sibling` (последовательность всех предшествующих узлов братьев).

Второй этап состоит в исключении из последовательности, полученной на первом этапе, узлов, не удовлетворяющих тесту. Тестом в XQuery может быть имя узла или символ "*", обозначающий любое имя. Проверка на удовлетворение тесту состоит в сравнении имени узла с тестом.

На третьем этапе происходит фильтрация последовательности узлов, полученных на предыдущем этапе, на основании результата вычисления предиката для каждого узла. Приведем пример путевого выражения, результатом которого является последовательность описания всех товаров, предлагаемых к продаже Смитом:

```
document("items.xml")/child::*/
child::item[self::*/child::seller="Smith"]
/child::description
```

Существует сокращенная форма записи путевых выражений, в которой ось `child` можно не указывать; ось `descendant` опускается, но перед этим шагом ставится два слэша; переход по оси `parent` описывается через две точки; вместо оси `attribute` ставится @.

Основной конструкцией XQuery является FLWOR-выражение, обладающее следующим синтаксисом:

```
( for $var1 in expr1 | let $var2 := expr2 )+
where expr3
order by expr4
return expr5
```

FLWOR-выражение состоит из пяти составляющих: разделов `for`, `let`, `where`, `order by` и `return`. В разделе `for` определяется переменная `$var1`, которая последовательно связывается с каждым из элементов последовательности, получаемой в результате вычисления выражения `expr1`. В разделе `let` переменная `$var2` связывается с результатом вычисления выражения `expr2`. Допускается несколько разделов `for` или `let`. В разделе `where` значения переменных фильтруются на основе предиката `expr3`. Раздел `order by` позволяет отсортировать кортежи связанных переменных в соответствии с выражением `expr4`. Наконец, раздел `return` определяет результат `FLWOR`-выражения: выражение `expr4` вычисляется для каждого связывания переменных, для которого результатом вычисления предиката в разделе `where` секции является "истина". Результатом всего `FLWOR`-выражения является последовательность узлов, полученных после вычисления раздела `return` .

При помощи `FLWOR`-выражений на XQuery можно выражать операции соединения, трансформации, группировки и т. д. Особую роль играет операция трансформации, поскольку она очень часто используется в Web-приложениях при построении динамических HTML-страниц. В XQuery операция трансформации выражается при помощи конструкторов элемента и атрибута. Примером трансформации является следующий запрос, результатом которого является новое представление книг с названием "Data on the Web" и автором "Serge Abiteboul":

```
<result>
{
  for $book in //book
  where $book/author = 'Serge Abiteboul' and
        $book/title = 'Data on the Web'
  return
    <book>
    {
      $book/title,
      $book/author,
      $book/isbn,
      $book/price
    }
```

```
    </book>
}
</result>
```

XQuery является функциональным языком, поэтому в нем выражения могут быть вложены друг в друга произвольным образом. Например, в разделе `for` `FLWOR`-выражения может находиться другое `FLWOR`-выражение и т. д. с любым уровнем вложенности. Помимо этого, на XQuery пользователь может определять и использовать свои собственные функции. Также существует достаточно большое количество встроенных стандартных функций [22]. Более детальное описание XQuery можно найти в статье Чемберлина [41].

1.2.3 Язык модификаций XUpdate

К настоящему времени рабочая группа XQuery в W3C находится только в процессе разработки стандартного языка изменений частей XML-документов. В нашей работе мы используем подмножество языка изменения частей XML-документов, который был предложен в статье Татаринова [56]. В дальнейшем язык изменений мы будем называть XUpdate. Синтаксис этого языка выглядит следующим образом:

```
update := 'InsertInto' '(' constr1 ',' locpath ')' |
          'InsertBefore' '(' constr2 ',' locpath ')' |
          'InsertAfter' '(' constr2 ',' locpath ')' |
          'Delete' '(' locpath ')'
          'Rename' '(' locpath, QName ')'
constr1:= 'element' '{' QName '}'  content |
          'attribute' '{' QName '}' content
constr2 := 'element' '{' QName '}'  content
content    := '{' PCDATA '}' | '{' '}'
```

Каждая операция принимает на вход несколько аргументов (*constr1*, *constr2* и *locpath*). Выражения *constr1* и *constr2* определяют новые узлы. Выражение *locpath1* определяет путь адресации узлов в XML-документе, которые являются целевыми узлами операции изменения. Ниже приводится описание каждой операции.

- *InsertInto(constr1, locpath)*: вставляет новый узел (элемент или атрибут) в каждый целевой узел в позицию последнего дочернего узла.

- *InsertBefore(constr2, locpath)*: вставляет новый узел (только элемент) для каждого целевого узла в позицию предшествующего брата.

- *InsertAfter(constr2, locpath)*: вставляет новый узел (только элемент) для каждого целевого узла в позицию последующего брата.
- *Delete(locpath)*: осуществляет глубокое удаление узлов (вместе со всеми потомками), определяемых *locpath*.
- *Rename(locapth, QName)*: присваивает новое имя *QName* целевым узлам операции.

1.3 XML-ориентированные СУБД

С ростом популярности XML в качестве формата описания слабоструктурированных данных появились огромные массивы XML-данных, для которых необходимы развитые средства управления. Это стало толчком к развитию XML-ориентированных СУБД, позволяющих эффективным образом управлять XML-данными. XML-ориентированные СУБД делятся на два класса [42].

К первому классу относятся СУБД с поддержкой XML, в которых технологии управления XML-данными реализованы на основе существующей модели данных. На сегодняшний день среди систем этого типа наиболее развиты средства поддержки XML в реляционных СУБД (РСУБД).

Во второй класс входят СУБД, изначально построенные с учетом модели данных XML[3]. Этот тип систем на сегодняшний день переживает бурный этап развития. За счет того, что на физическом и модельном уровне эти системы учитывают особенности XML, они более эффективны для управления XML-данными, чем СУБД первого класса. Системы управления базами XML-данных, построенные с нуля и в полной мере основанные на модели данных XML мы будем назвать прирожденными XML-СУБД (или XML-СУБД).

1.3.1 РСУБД с поддержкой XML

В настоящие время практически все коммерческие реляционные СУБД [28, 36, 32] предоставляют средства управления XML-данными. В последующих разделах мы делаем краткий обзор этих средств и обсуждаем их возможности и ограничения.

XML-представления

XML-представления (XML views), которые появились в РСУБД достаточно давно, позволяют реализовать XML-оболочку между реляционным хранилищем данных и

[3]Далее в этой работе под моделью данных XML мы будем подразумевать модель данных языков XPath/XQuery [19].

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:sql="urn:schemas-microsoft-com:mapping-schema">
 <xsd:annotation>
  <xsd:appinfo>
    <sql:relationship name="CustOrders"
          parent="Customers"
          parent-key="CustomerID"
          child="Orders"
          child-key="CustomerID" />
  </xsd:appinfo>
 </xsd:annotation>
 <xsd:element name="Customer" sql:relation="Customers" type="CustomerType" />
   <xsd:complexType name="CustomerType">
     <xsd:attribute name="CustomerID" sql:field="CustomerID" type="xsd:string"/>
     <xsd:attribute name="CompanyName" sql:field="CompanyName" type="xsd:string"/>
       <xsd:sequence>
          <xsd:element name="Order"
                       sql:relation="Orders"
                       sql:relationship="CustOrders" >
             <xsd:complexType>
                <xsd:attribute name="OrderID" sql:field="OrderID" type="xsd:integer" />
                <xsd:attribute name="ProductName" sql:field="ProductName" type="xsd:string" />
             </xsd:complexType>
          </xsd:element>
       </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

Рис. 1.4: Определение XML-представления в MS SQL Server 2000/2005

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql">
    <sql:xpath-query mapping-schema="CustOrders.xml">
        /Customer[CustomerID="John Smith"]/Order[@ProductName="DVD"]
    </sql:xpath-query>
</ROOT>
```

Рис. 1.5: Пример запроса на выборку к XML-представлению

приложениями. При помощи XML-представления можно интегрировать в один общий нематериализованный XML-документ (или набор нематериализованных XML-документов) уже существующие (legacy) реляционные данные. Фактически XML-представления позволяют приложениям оперировать с реляционными данными в терминах модели данных XML. Далее мы рассмотрим, как определяются XML-представления в РСУБД, на примере SQL Server 2000/2005.

В SQL Server 2000/2005 [25] XML-представление определяется при помощи специального языка [26], который синтаксически очень похож на язык описания схем XML-

документов XML Schema [17, 18], но в нем еще есть средства определения отображения между нематериализованными XML-представлениями и реляционными отношениями. На основе этого отображения система преобразует запросы пользователя к XML-представлению в запросы к реляционным отношениям. На Рисунке 1.4 приведен пример XML-представления, определенного над двумя отношениями:

```
Customers(CustomerID int PRIMARY KEY, CompanyName varchar(20))
Orders(OrderID varchar(10) PRIMARY KEY, CustomerID int FOREIGN KEY REFERENCES,
       ProductName varchar(20))
```

Атрибуты *sql:relation*, *sql:field* и *sql:relationship* определяют отображение отношений на представление. Представление на Рисунке 1.4 отображает строки отношения *Customers* в элементы с именем *Customers*, а соответствующие им заказы *Orders* отображаются в набор вложенных элементов *Order*. Важнейшей конструкцией является аннотация *CustOrders*, которая определяет соединение отношений *Customers* и *Orders*, и за счет этого в каждом элементе *Customer* содержатся вложенные элементы *Order*, которые относятся именно к этому *Customer*. Соединение отношений определяется на основе первичного ключа *CustomerID* в отношении *Customers* и внешнего ключа *CustomerID* в отношении *Orders*. Более подробное описание отображений можно найти в работе Майкла Риса [27].

На рисунке 1.5 изображен пример запроса к XML-представлению. Значением атрибута *sql:mapping-schema* является имя файла, в котором содержится определение XML-представления (в нашем примере это файл с именем *CustOrders.xml*), а значением элемента *sql:xpath-query* служит запрос на языке XPath. Результатом этого запроса будет набор заказов *DVD* дисков, которые были сделаны Джоном Смитом.

Кроме того, в SQL Server 2000/2005 пользователь может производить элементарные операции модификации виртуальных XML-представлений на основе механизма *updategrams*. В основном этот механизм позволяет изменять значения атрибутов или текстовых эментов, но не позволяет изменять структуру XML-документа. *Updategram* представляет собой конструкцию, состоящую из двух блоков. В первом блоке *before-image* пользователь указывает узлы XML-документа, которые необходимо изменить, а во втором блоке *after-image* указывается новое значение узлов из блока *before-image*. В свою очередь, система автоматически вычисляет разницу между *before-image* и *after-image*, создает набор соответствующих операторов SQL и выполняет их. На рисунке 1.6 приведен пример *updategram*, выполнение которой приводит к замене значения элемента *CompanyName* для заказчика *"LAZYK"* на *"Corp2"* и смену идентификатора заказа *10482* на *10354*.

В Oracle [29] средства определения нематериализованных XML-представлений очень похожи на аналогичные средства в SQL Server 2000/2005. Но операции обновления

```
<root xmlns:updg="urn:schemas-microsoftcom:xml-updategram">
  <updg:sync mapping-schema="CustOrders.xml">
  <updg:before>
    <Customer CustomerID="LAZYK">
       <Order OrderID="10482"/>
    </Customer>
  </updg:before>
  <updg:after>
    <Customer CustomerID="LAZYK" CompanyName="Corp2">
      <Order OrderID="10354"/>
    </Customer>
  </updg:after>
</updg:sync>
</root>
```

Рис. 1.6: Пример запроса на изменение к XML-представлению

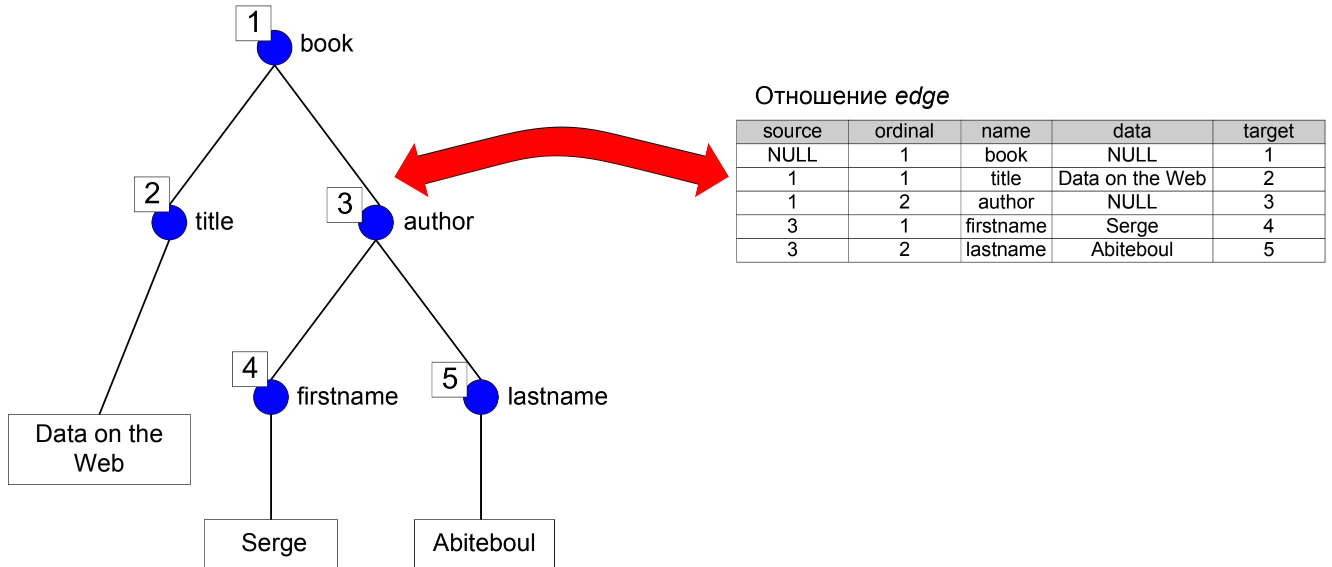
возможны только на уровне всего XML-документа - пользователь может удалять целиком XML-документ и вставлять новый, удовлетворяющий предписанной схеме. В IBM DB2 XML Extender 8.0 [31] также существуют схожие средства построения виртуальных XML-представлений. Спецификация отображения описывается в DAD (Document Access Definition) файле. При этом операции обновления можно выполнять только над отношениями и нет средств выполнять операции изменения через XML-представление.

Важно заметить, что на данный момент коммерческие РСУБД поддерживают только ограниченный вид XML-представлений, в которых соединения отношений производятся только на основе пары первичный ключ - внешний ключ, а также для избежания неопределенности при трансляции операции изменений над представлением в операции изменения реляционных отношений вложенность узлов поддерживается в ограниченном виде.

Хранение XML-документов в реляционных таблицах

Вторым направлением исследований по поддержке XML в РСУБД являются методы хранения XML-документов в реляционных СУБД. При этом основной задачей здесь является метод отображения XML-документа на набор реляционных отношений. В следующих параграфах приводится обзор базовых методов отображения XML-документов на реляционные таблицы: Edge [95], Attribute [95], Inlining [51], STORED [52]. Детальный обзор и сравнение этих методов можно найти в [53].

Метод Edge основывается на интуитивном представлении XML-документа как дерева, состоящего из набора ребер. Создается одно отношение с именем *edge*, каждый кортеж которого хранит одно ребро дерева XML-документа. Как показано на рисунке 1.7, это



source	ordinal	name	data	target
NULL	1	book	NULL	1
1	1	title	Data on the Web	2
1	2	author	NULL	3
3	1	firstname	Serge	4
3	2	lastname	Abiteboul	5

Рис. 1.7: Отображение XML-документа на отношения на основе метода Edge

отношение имеет пять атрибутов: *source*, *ordinal*, *name*, *data*, *target*. Каждому узлу в XML-документе присваивается уникальный идентификатор. В атрибутах *source* и *target* содержатся идентификаторы верхних и нижних концов ребра соответственно. Атрибут *ordinal* служит для определения порядка ребер в XML-документе. В нем содержится порядковое число ребра среди других ребер, обладающих одним и тем же верхним (*source*) узлом. Наконец, атрибут *data* определяет, является ли целевой узел ребра (*target*) листовым или внутренним. В первом случае значение *data* указывает текстовое значение узла; во втором случае значение этого атрибута равно *NULL*.

Недостатком метода *Edge* является его низкая эффективность, поскольку выполнение запросов над отношением edge требует выполнения большого количества операций соединения. Для иллюстрации этого факта рассмотрим пример.

Пример 1. *Предположим, что пользователь запустил XPath-запрос /book[1]/author[1]/firstname/text() над XML-документом, изображенным на рисунке 1.7. Этот запрос выбирает содержание элемента firstname первого автора первой книги в документе. Этот запрос соответствует следующему оператору SQL над отношением edge:*

```
SELECT data
FROM edge e1, edge e2, edge e3
WHERE    e1.name = 'book'          (1)
     and e1.ordinal = 1            (2)
     and e2.name = 'author'        (3)
```

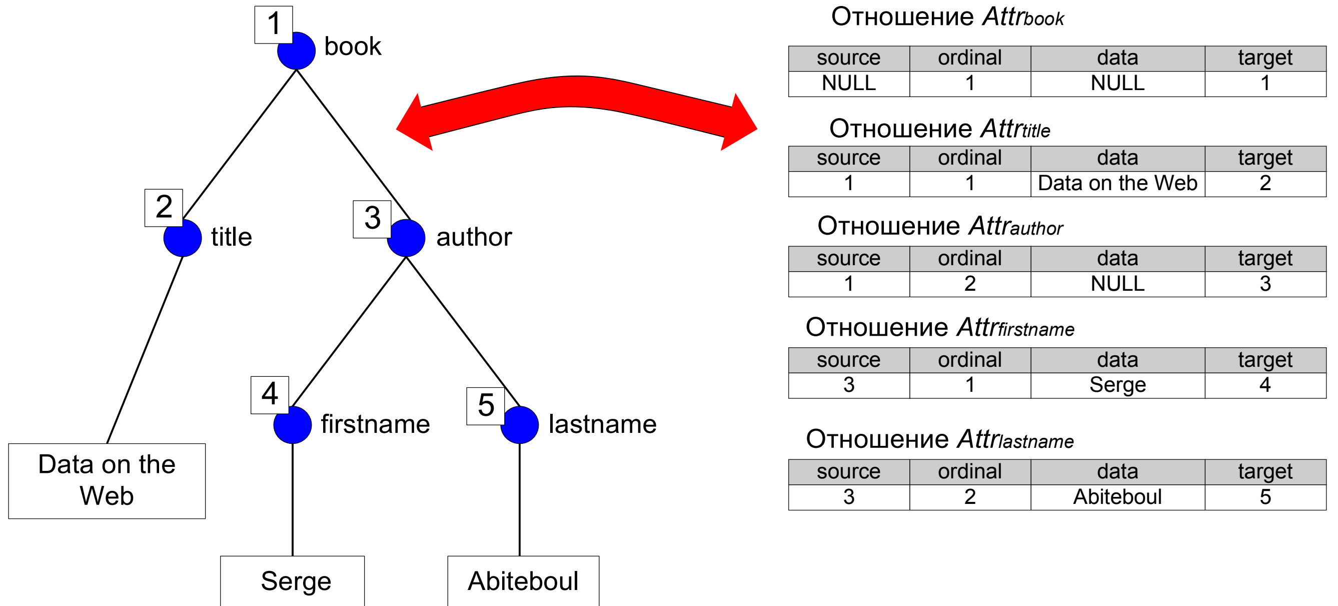


Рис. 1.8: Отображение XML-документа на реляционные таблицы на основе метода Attribute

```
and e2.ordinal = 1          (4)
and e1.target = e2.source   (5)
and e3.name = 'firstname'   (6)
and e2.target = e3.source   (7)
```

В этом запросе производится две операции соединения, выраженные при помощи условий (5) и (7).

Метод Attribute является развитием метода *Edge*. Фактически, в методе *Attribute* происходит горизонтальное разделение отношения *edge* по атрибуту *name*. В результате отношение *edge* разделяется на набор отношений, число которых определяется числом различных значений атрибута *name*. Заголовки полученных отношений отличаются от заголовка отношения *edge* отсутствием атрибута *name*. Семантика остальных атрибутов такая же, как и в методе *Edge*. На рисунке 1.8 изображен пример отображения XML-документ на отношения при помощи метода *Attribute*.

По сравнению с методом *Edge* в методе *Attribute* размер отношений меньше, чем размер отношения *edge*, но при этом количество операций соединения остается таким же. Количество предикатов на значения имен элементов сокращается. Метод *Attribute* очень хорошо подходит для запросов типа `//<node-name>`, поскольку в этом случае все необходимые узлы хранятся в одной таблице $Attr_{node-name}$.

Метод *Inlining* отличается от предыдущих методов тем, что для его использования

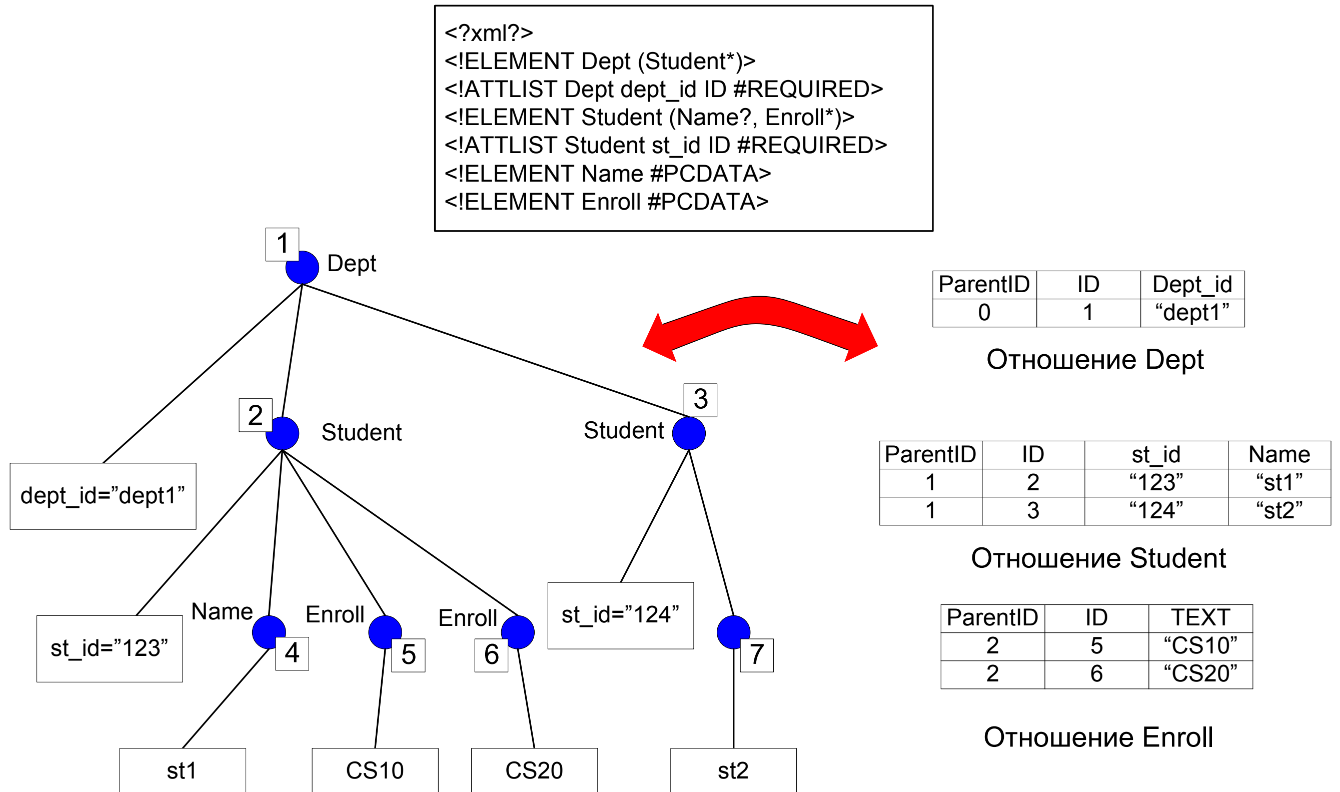


Рис. 1.9: Отображение XML-документа на реляционные таблицы на основе метода Inlining

необходимо знать схему XML-документа[4]. XML-документ распределяется по некоторому набору таблиц. Этот набор определяется исходя из схемы XML-документа. Для каждого элемента по схеме, которому соответствуют дети с множественностью “+” или “*”, заводится отдельная таблица. Элементы с множественностью не более одного *встраиваются* в кортеж родителя. Каждый кортеж в таблице обладает уникальным идентификатором *ID*, а также содержит идентификатор родителя *ParentId*. Это необходимо для восстановления отношения предок-потомок между узлами. Пример этого отображения (заимствован из [53]) изображен на рисунке 1.9. В работе [51] показывается, что метод Inlining превосходит все методы, описанные выше. Это происходит за счет того, что количество операций соединения, требуемых для выполнения XPath запросов, сокращается. Отметим, что в оригинальном методе Inlining теряется информация о порядке следования узлов в XML-документе. Однако это не означает, что это является ограничением метода. Например в работе Татаринова [57] предлагается расширение метода Inlining, в котором поддерживается порядок следования узлов, и при этом сохраняются все преимущества оригинального метода.

Основная идея метода *STORED* заключается в следующем: XML-документ хранится в большом объекте типа BLOB/CLOB, который в свою очередь принадлежит кортежу

[4]Схема XML-документа может быть описана, например, на языке DTD.

некоторой таблицы, которую мы будем называть таблицей документов (document table). Для увеличения скорости выборки данных вводятся дополнительные индексированные таблицы. Для определения частей XML-документа, которые должны храниться в индексированной таблице, проектировщик базы данных должен определить STORED-запрос. Отметим, что индексированные таблицы частично дублируют содержание таблицы документов. На рисунке 1.10 на примере изображена основная идея метода *STORED*. Индексированная таблица в соответствии с STORED-запросом хранит значения price и description. Мы предполагем, что таблица проиндексирована по атрибуту price. В результате, например запрос */site/auction[price<10]* может выполняться в два этапа: (1) на первом этапе определяется набор документов *D1, D2, DN* для которых существует элементы *auction*, у которых $price < 10$; этот этап выполняется эффективно на основе индексированной таблицы, (2) на втором этапе для выбранных идентификаторов документов выполняется исходный запрос. В результате процессору запросов не нужно производить разбор (parsing), всех XML-документов, а вместо этого требуется разбирать только часть. Кроме того, выборка целых XML-документов производится очень эффективно по сравнению с предыдущими методами. Отметим, что на базе метода STORED была реализована поддержка XML в MS SQL Server 2000 и DB2 XML Extender 7.1. Однако последние версии этих продуктов реализуют поддержку XML на основе метода, к описанию которого мы переходим.

Хранение XML-документов в атрибутах с типом XML

Третий метод хранения XML-документов заключается в использовании специального типа в SQL - `XML`. Это тип сравнительно недавно появился в языке SQL у большинства коммерческих производителей РСУБД [60]. Создав таблицу с атрибутом типа `XML`, пользователь может загружать в эту таблицу XML-документы, и затем писать к ним запросы на языке XQuery. Кроме того, для изменения XML-документов, как правило, вводится специальный язык изменения XML-документов на уровне отдельных узлов. Ниже на примере MS SQL Server 2005 мы приводим примеры того, как можно создавать таблицу с атрибутами `XML`, загружать туда XML-документы, запрашивать XML-документы и изменять XML-документы.

```
/* создание таблицы с атрибутом XML */
CREATE TABLE docs (pk INT PRIMARY KEY, xDoc XML)


/* загрузка XML документа */
INSERT INTO docs
      SELECT 1, xCol
      FROM  (SELECT * FROM OPENROWSET
```

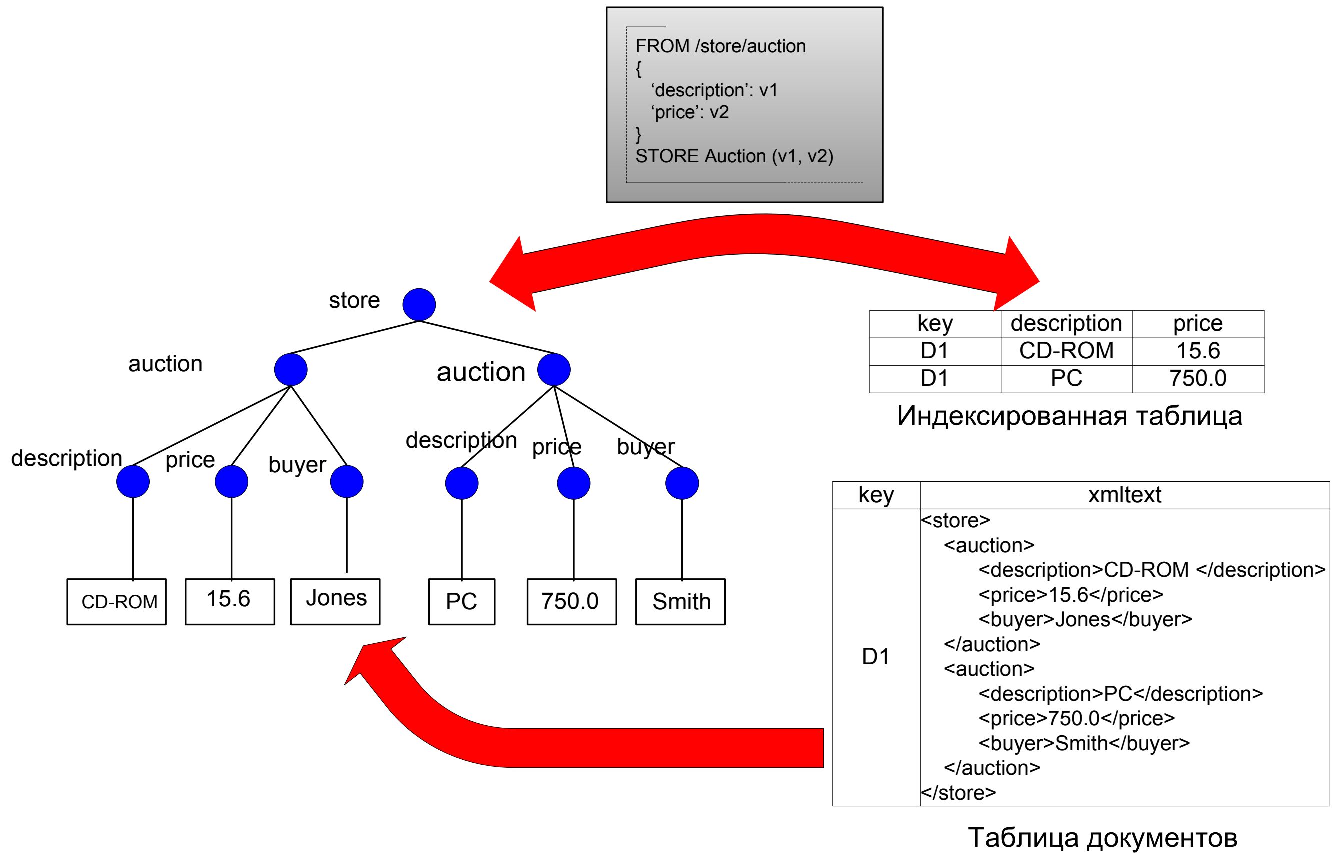


Рис. 1.10: Метод STORED хранения XML-документов

```
          (BULK 'document.xml',
           SINGLE_BLOB) AS xCol) AS R(xCol)


/* примеры выборки частей XML-документов */
SELECT pk, xDoc.query('/doc[@id = 15]//chapter') FROM docs
SELECT pk, xDoc.query('for $s in /doc[@id = 123]//chapter
                       where $s/@num >= 3
                       return <topic>{data($s/title)}</topic>')

/* пример изменения частей XML-документов */
UPDATE docs SET xCol.modify('
                insert <section num="2">
                           <title>Background</title>
                          </section>
```

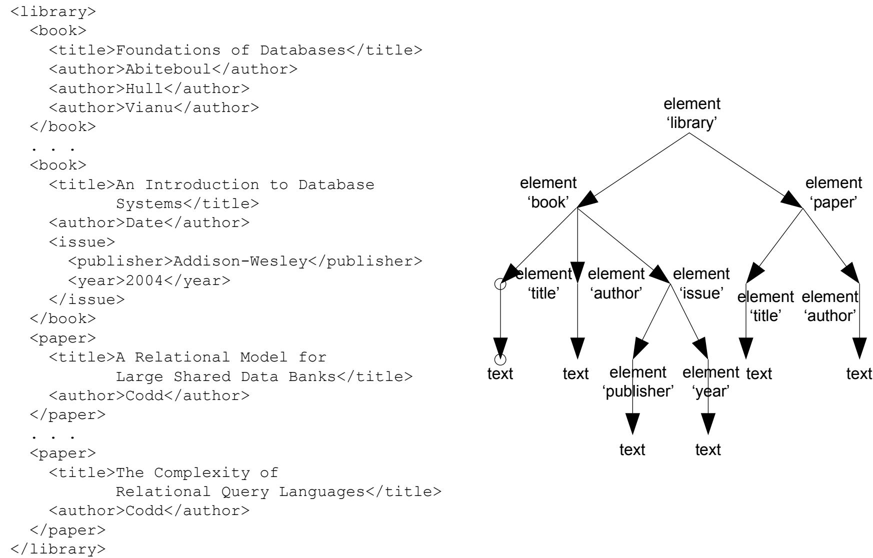
```
after (/doc//section[@num=1])[1]')
```

Надо отметить, что тип данных XML появился сравнительно недавно и предоставляет наибольшую функциональность по управлению XML-документами в РСУБД. Действительно, метод хранения в данном случае максимально приближен к прирожденным методам (например в работе [59] описан метод хранения в DB2), что положительно влияет на эффективность выполнения запросов на чтение и изменение XML-документов. Кроме того, средства выборки реализуются при помощи мощного функционального языка XQuery, на котором можно писать достаточно сложные запросы. Фактически, с появлением типа XML можно говорить, что в РСУБД появилась поддержка XML, максимально приближенная к прирожденным СУБД к обсуждению которых мы переходим.

1.3.2 Прирожденные XML-СУБД

Появление класса прирожденных XML-СУБД обусловлено требованием эффективной обработки огромных объемов XML-данных. В работах [78, 80] обсуждается ограниченность реляционных баз данных для управления древовидными, слабоструктурированными XML-данными. Авторы этих работ приходят к выводу, что построение высокоэффективной системы управления XML-данными требует пересмотра ряда важнейших принципов, включая организацию внешней памяти, организацию обработки XML-данных в основной памяти, управление транзакциями, индексацию данных и т. д. На сегодняшний день во многих из этих областей ведутся активные исследования и появляются новые прототипы систем. Среди наиболее развитых прирожденных XML-СУБД можно выделить Natix [78], Tamino [79], Excelon [37], Timber [80], Sedna [76], eXist[81] и X-Hive [82].

В этом разделе мы рассмотрим общую архитектуру и основные физические особенности прирожденной XML-СУБД Sedna (Седна), которая разрабатывается в течении последних трех лет в институте системного программирования РАН. В контексте этой системы автором были разработаны методы управления XML-транзакциями, которые представлены в главе 5. Выбор системы Седна обуславливается не только участием автора в проекте разработки этой системы, но и тем фактом, что на сегодняшний день Седна является одной из самых эффективных XML-СУБД [77]. Важно отметить, что хотя предложенные методы были разработаны в контексте Седна, область их применения не ограничивается этой системой. Далее мы переходим к обсуждению физических особенностей организации Седна. Более детальное описание можно найти в диссертационной работе Фомичева [61].



```
<library>
  <book>
    <title>Foundations of Databases</title>
    <author>Abiteboul</author>
    <author>Hull</author>
    <author>Vianu</author>
  </book>
  . . .
  <book>
    <title>An Introduction to Database
           Systems</title>
    <author>Date</author>
    <issue>
      <publisher>Addison-Wesley</publisher>
      <year>2004</year>
    </issue>
  </book>
  <paper>
    <title>A Relational Model for
           Large Shared Data Banks</title>
    <author>Codd</author>
  </paper>
  . . .
  <paper>
    <title>The Complexity of
           Relational Query Languages</title>
    <author>Codd</author>
  </paper>
</library>
```


Рис. 1.11: XML-документ и его описывающая схема

Организация хранения XML-данных во внешней памяти

Внешняя память разбивается на блоки одинакового размера, который может настраиваться при компиляции системы. XML-документ рассматривается как набор узлов. Каждому узлу XML-документа соответствует на диске дескриптор этого узла. Стратегия хранения XML-данных во внешней памяти заключается в кластеризации дескрипторов узлов XML-документа, соответствующих одному узлу описывающей схемы (пример описывающей схемы для XML-документа приведен на рисунке 1.11). Таким образом, одному узлу описывающей схемы соответствует цепочка блоков, в каждом из которых хранятся дескрипторы узлов XML-документа. В одном блоке могут храниться дескрипторы узлов, соответствующие одному узлу описывающей схемы. Напомним, что описывающая схема - это дерево, которое содержит по одному разу все пути, представленные в XML-документе. Рисунок 1.12 иллюстрирует описанную стратегию. Отметим, что дескрипторы узлов не содержат имя узла. Вместо этого в заголовке блока хранится указатель на узел схемы, к которому этот блок относится, а тот в свою очередь хранит имя узла. Это позволяет существенно сократить общий размер базы данных.

Блоки данных, соответствующие одному узлу схемы, связываются при помощи указателей `prev-block` и `next-block` в двунаправленный список. Узлы в такой цепочке частично упорядочены. Это означает, что если $block_1$ предшествует в списке $block_2$, то каждый узел из $block_1$ предшествует любому узлу из $block_2$ в соответствии с порядком следования узлов в XML-документе. Порядок узлов в одном блоке не соответствует их порядку в XML-документе, но он может быть восстановлен при помощи специальных

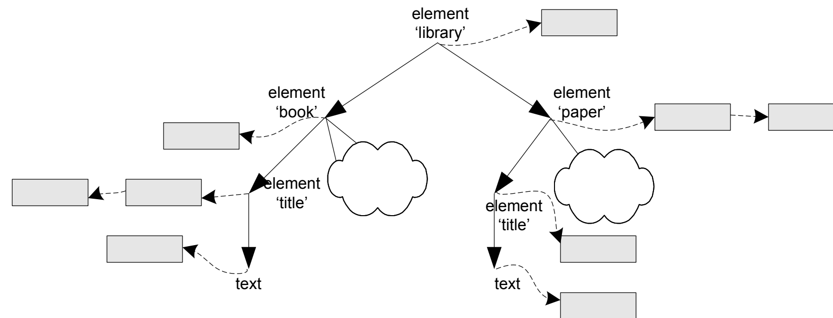


Рис. 1.12: Организация внешней памяти в Седна

указателей, которые мы обсудим ниже.

В системе хранения разделяется структурная составляющая узла и текстовое значение этого узла. Для текстовых значений узлов применяется устоявшийся и хорошо себя зарекомендовавший метод - слоттированные страницы (slotted page) [62], специально разработанный для хранения записей переменной длины. Структурная составляющая, представленная в виде дескрипторов узлов, определяет взаимосвязь (например родитель, ребенок, брат и т.д.) между другими узлами в документе. Важнейшим свойством дескрипторов узлов является то, что в каждом блоке они обладают одинаковым размером. В совокупности с тем, что дескрипторы узлов внутри блока неупорядочены, это существенно облегчает управление свободным местом в блоке и позволяет проводить операции вставки и удаления дескрипторов узлов в блоке без перемещения остальных дескрипторов в блоке.

Общая структура дескриптора узла изображена на рисунке 1.13. Дескриптор содержит несколько ссылок на соседние с ним узлы, которые мы сейчас подробно рассмотрим. Важно отметить, что все ссылки реализуются при помощи указателей, переход по которым осуществляется очень эффективно.

Указатели `left-sibling` и `right-sibling` ссылаются на дескрипторы левого и правого братьев соответственно. Эти указатели служат для поддержания связанности узлов в документе. Дескрипторы братьев могут находиться как в том же блоке, что и рассматриваемый дескриптор, так и в другом блоке в зависимости от соответствия узлу схемы.

Указатели предыдущий узел в блоке (`prev-in-block`) и следующий узел в блоке (`next-in-block`) служат для связи узлов в одном блоке в соответствии с их порядком следования в документе. Таким образом, используя эти указатели, можно эффективно восстановить порядок следования узлов в XML-документе, соответствующих одному узлу схемы. Например, ответом на запрос `/library/book/tittle` должна быть последовательность узлов `title`, отсортированная в соответствии с порядком следования узлов `title` в XML-документе. В описываемом методе хранения такой запрос можно

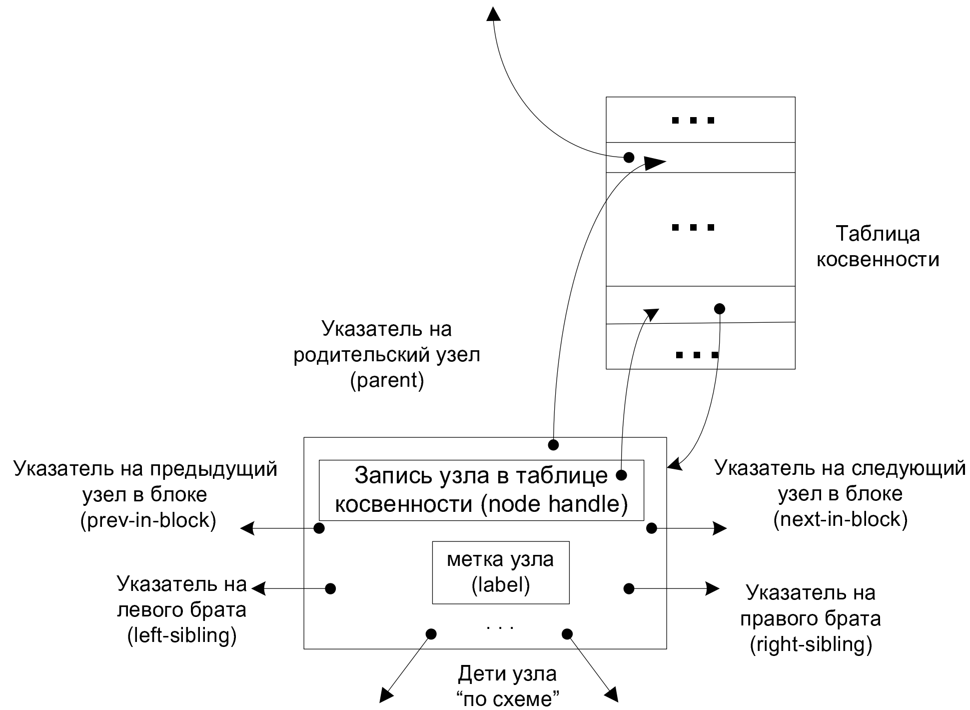


Рис. 1.13: Структура дескриптора узла

выполнить следующим образом: по описывающей схеме получить указатель на цепочку блоков `title`, а затем, используя указатель `next-in-block` и указатели между блоками `prev-block` и `next-block`, выбрать сразу в упорядоченном виде последовательность узлов `title`.

Хранение в дескрипторе узла указателей на всех детей данного узла может приводить к неограниченно большому размеру дескриптора узла, который даже возможно не будет помещаться в блок. Для избежания подобной ситуации в дескрипторе узла хранятся указатели не на всех детей данного узла, а только на первых детей узла, принадлежащих разным узлам схемы. Рассмотрим это на примере узла `library`, изображенного на рисунке 1.12. Он имеет в качестве детей два узла `book` и один узел `paper`, которые соответствуют двум узлам по схеме. Вне зависимости от количества узлов `book` и `paper` в дескрипторе узла `library` будет храниться два указателя на детей: указатель на дескриптор первого узла `book` и указатель на первый узел `paper`. Стратегия выполнения запроса `/library/book[2]` с использованием этих указателей может быть следующая: получить по описывающей схеме указатель на блок, в котором хранится дескриптор узла `library`, затем по указателю перейти на первый узел `book`, и, наконец, по указателю `next-in-block` перейти ко второму узлу `book`.

Указатель на родителя (`parent`) служит для ссылки на родителя данного узла. Важно отметить, что этот указатель является косвенным. Он указывает на некоторую ячейку таблицы косвенности, в которой содержится прямой указатель на родителя узла. Это

сделано для эффективного выполнения операции модификаций. Например, при вставке некоторого узла возможно переполнение блока, что влечет за собой перемещение части дескрипторов узлов в другой блок. В случае использования прямого указателя на родителя придется менять значение указателя `parent` для каждого узла, который является ребенком перемещаемого узла. При использовании же таблицы косвенности необходимо только соответствующим образом изменить значение в ячейке таблицы косвенности.

Запись узла в таблице косвенности (`node handle`) является указателем на запись для этого узла в таблице косвенности. Важнейшим свойством `node handle` является то, что он никогда не изменяется во время "жизни" узла в XML-документе. Поэтому `node handle` может использоваться в качестве уникального идентификатора узла.

Наконец, метка `label` является нумерующим числом узла. Нумерующие числа узлов позволяют эффективным образом определять структурную взаимосвязь между узлами. Нумерующие числа позволяют определять порядок между узлами и отношение предок-потомок. Например, нумерующие числа могут использоваться при выполнении запроса `//title` для восстановления порядка следования узлов `title`. Детальное описание нумерующей схемы в Седна можно найти в работе Азнауряна, Новака и Кузнецова [87].

Управление памятью для хранимых XML-данных

Как описывалось в предыдущей секции, связь между дескрипторами узлов реализуется при помощи указателей (прямых или косвенных). В результате при выполнении запросов над описанными структурами данных очень часто выполняется операция перехода по указателю. Таким образом, эффективность этой операции существенным образом влияет на общее время выполнение запросов в системе.

В Седна реализован оригинальный метод управления памятью, основанный на слоистой организации адресного пространства базы данных, который позволяет выполнять операции перехода по указателям очень эффективно. Кроме того, метод позволяет иметь размер базы данных, достаточный для представления всех сущностей базы данных[5]. Мы в этой работе только отметим базовые идеи организации слоистого адресного пространства (САП), а подробное описание можно найти в диссертационной работе Фомичева [61].

На рисунке 1.14 изображена архитектура слоистого адресного пространства. Далее мы описываем компоненты этой архитектуры.

Подсистема управления буферной памятью (менеджер буферной памяти) располагается в отдельном процессе операционной системы (рисунок 1.14). Для поддержки сессии пользователя на стороне сервера также запускается отдельный процесс, в котором

[5] Метод позволяет адресовать 64-х разрядные адресные пространства.

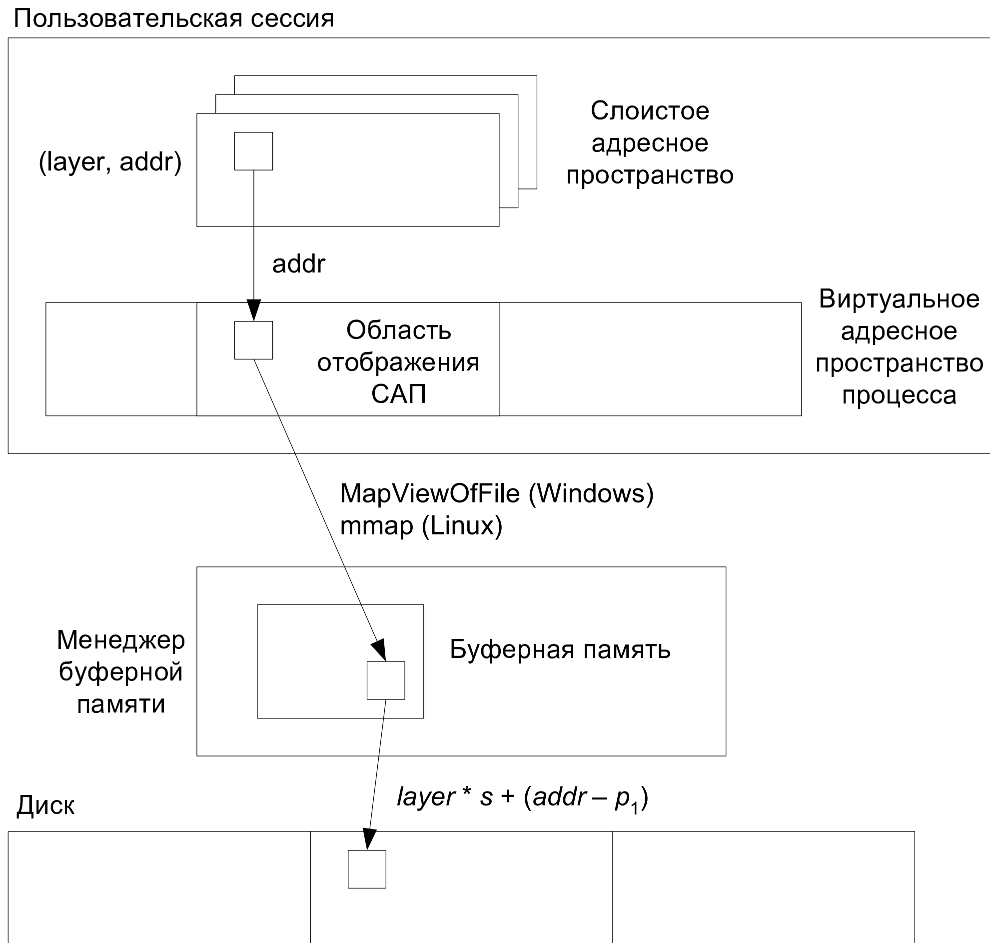


Рис. 1.14: Реализация слоистого адресного пространства

есть свои копии подсистемы выполнения запросов и подсистемы поддержки САП. В рамках этого процесса выполняются транзакции, инициированные пользователем. На рисунке 1.14 этот процесс изображен как “пользовательская сессия”. Естественно полагать, что одновременно может быть запущено несколько пользовательских сессий, в то время как менеджер буферной памяти - один на базу данных.

САП проще всего представить себе как набор слоев одинакового размера. Размер слоя фиксирован и не превышает размера адресного пространства процесса, а точнее той его части, которая может быть адресована процессом.

Адресом в слоистом адресном пространстве называется пара *(layer, addr)*, где *layer* - номер слоя (представляет собой целое число) и *addr* - адрес объекта в этом слое (представляет собой указатель). В дальнейшем, наряду с термином *адрес в слоистом адресном пространстве*, мы будем употреблять более короткое обозначение *xptr* (от *eXtended pointer*).

Важной особенностью описанной архитектуры является то, что для каждой

выполняемой в данный момент времени транзакции имеется свое адресное пространство, поскольку каждая транзакция выполняется в отдельном процессе. На рисунке 1.14 показаны четыре уровня памяти: (1) слоистое адресное пространство, (2) виртуальное адресное пространство процесса, (3) буферная память, (4) внешняя память.

Работа с САП обеспечивается через последовательное отображение САП на оставшиеся уровни памяти. При этом единицей отображения является страница[6]. Для каждой страницы САП должно быть выделено место на диске (отображение САП на внешнюю память), а во время работы с этой страницей она должна быть помещена в оперативную память (отображение САП на буферную память) и должна быть доступна по некоторому адресу в процессе транзакции (отображение САП на виртуальное адресное пространства процесса).

Отображение объекта САП с адресом *xptr=(layer, addr)* на виртуальное адресное пространство процесса, а точнее на его часть - область отображения САП, производится путем отбрасывания значения *layer*. В соответствии с таким отображением, для того чтобы обратиться к объекту по адресу *xptr=(layer, addr)* в САП, достаточно обратиться к объекту по адресу *addr* в виртуальном адресном пространстве процесса. Предложенное отображение не является однозначным, т.е нескольким адресам САП может соответствовать один адрес виртуального адресного пространства процесса. В [61] приводится эффективное решение этой проблемы.

Для отображения адреса виртуального адресного пространства процесса на буферную память используются стандартные системные вызовы: *MapViewofFile* [38] в Wndows и *mmap* [39] в Linux.

Наконец, отображение САП на внешнюю память (мы считаем, что это файл базы данных на жестком диске) осуществляется следующим образом. Для блока с адресом *xptr=(layer, adrr)* смещение в файле определяется по формуле $f = layer * s + (addr - p_1)$, где S - размер слоя САП, а p_1 - левая граница области отображения САП.

1.4 Выводы

Технологии платформы XML бурно развиваются как в научном сообществе специалистов по базам данных, так и ведущими коммерческими производителями СУБД. Разработано огромное количество методов хранения XML как в реляционных СУБД, так и в прирожденных XML-СУБД. Однако для получения зрелой и законченной технологии управления XML-данными, необходимо также иметь развитые средства управления транзакциями над XML-данными. С одной стороны, к сегодняшнему дню уже разработано

[6]Далее термины блок и страница мы будем использовать как взаимозаменяемые.

достаточно большое количество методов управления транзакциями, которые можно было бы использовать для управления XML-транзакциями. Однако применимость этих методов для XML является открытым исследовательским вопросом. Именно ответу на этот вопрос, а также обсуждению сопряженных методов управления XML-транзакциями посвящена следующая глава.

Глава 2

Существующие методы управления параллельными XML-транзакциями

Настоящая глава посвящена описанию существующих методов управления параллельными транзакциями в XML-ориентированных СУБД. Кроме того, делаются выводы о применимости этих методов для эффективного управления XML-транзакциями.

2.1 XML-транзакции в реляционных СУБД

Реляционные СУБД уже достаточно давно достигли стадии зрелости, а это подразумевает, что в них существует эффективная поддержка транзакций. Однако управление транзакциями (в частности контроль параллельного выполнения транзакций) в них реализовано с учетом реляционной модели данных, которая существенно отличается от иерархической модели XML.

Поддержка XML в РСУБД реализуется в виде некоторой надстройки над реляционной моделью. Но реальная обработка XML-запросов происходит в реляционном ядре СУБД. Когда XML-запрос доходит до реального выполнения, он уже транслирован в набор SQL-запросов, и подсистема выполнения воспринимает их как обычные запросы над реляционными данными. В результате во многом теряется информация о иерархической структуре XML и семантике XML-запросов. В частности, это приводит к тому, что реляционный менеджер блокировок во многих случаях создает искусственные конфликты между транзакциями, даже если он использует блокировки на уровне кортежей. Ниже для каждого типа хранения XML-документов в РСУБД, описанных в вводной главе, мы рассматриваем большое количество типичных ситуаций, в которых реляционный менеджер блокировок создает искусственные конфликты между транзакциями. Тем самым существенно снижается пропускная способность и время отклика транзакций в РСУБД при

наличии конкурентных транзакций, работающих с одним XML-документом на чтение и изменение.

Отметим, что ниже мы не рассматриваем нематериализованные XML-представления, поскольку средства изменения XML-документов через нематериализованные представления на данный момент либо вообще отсутствуют в РСУБД, либо развиты очень слабо.

2.1.1 Блокировки в РСУБД для XML-документов при использовании отображения XML-документов на отношения

Сценарий 1: вставка нового узла на верхнем и нижнем уровнях иерархии

Предположим, что транзакция T_1 выполняет запрос $InsertInto(< author/ >, /book)$, а вторая транзакция T_2, запущенная параллельно с T_1, намерена выполнить запрос $/book/author/firstname$. Рассмотрим, на какие кортежи будут устанавливать блокировки эти транзакциями для метода хранения *Edge*. Транзакция T_1 должна, как минимум, заблокировать на чтение кортеж, соответствующий ребру (корень документа, book) (первый кортеж на рисунке 1.7), и заблокировать на запись новый, вставленный кортеж, соответствующий новому ребру (book, author). В то же время транзакции T_2 необходимо выполнить SQL-запрос с двумя операциями соединения таблицы edge и двумя предикатами на значения имен узлов (см. пример для метода *Edge*). В результате транзакции T_2 необходимо заблокировать кортежи с именем *author*. Но один из этих кортежей уже заблокирован на запись первой транзакцией. Поэтому вторая транзакция вынуждена ожидать завершения первой. Если даже в РСУБД используются предикатные блокировки (реализованные, например, на основе блокировок интервалов ключей в индексе), то транзакция T_1 перед выполнением второго шага путевого выражения должна заблокировать на чтение предикат $source = 1 \wedge name =$ “$author''$, который запретит вставку новых элементов *author*. Фактически, получается, что вставка узла на верхнем уровне блокирует навигацию к нижним узлам. Схожая ситуация возникнет и для других методов хранения XML-документов.

Конфликт между транзакциями T_1 и T_2 является искусственным, поскольку вставка пустого элемента *author* никак не может повлиять на результат выполнения путевого выражения $/book/author/firstname$. Фактически, эти операции коммутативны. Конфликт возник бы, если первая транзакция вставляла элемент *author* с вложенным элементом *first-name*.

Далее рассмотрим новую пару конкурентных транзакций. Транзакция T_1 - $InsertInto(< firstname > John < /firstname >, /book/author)$, а T_2 - $/book//lastname$. Очевидно, что на логическом уровне между этими транзакциями нет конфликта, однако

менеджер блокировок в РСУБД вновь зафиксирует конфликт. Действительно, вставка элемента *firstname* требует установки эксклюзивной блокировки на новый кортеж, а выполнение поиска всех элементов *lastname* требует просмотра всех кортежей, которые относятся к узлам, находящимся под узлом *book*. В результате транзакция T_2 будет пытаться установить разделяемую блокировку и на кортеж, вставленный транзакцией T_1, что приведет к ее блокировке. Таким образом, даже вставка узла на нижнем уровне иерархии приводит к искусственным конфликтам с читающими транзакциями.

Сценарий 2: вставка нового узла между существующими братьями

Рассмотрим ситуацию когда транзакция намерена вставить узел, после некоторого существующего узла. Для этого она использует операцию $InsertAfter(< author/ >, /book/title)$. Поскольку эта вставка осуществляется между существующими узлами-братьями *title* и *author*, то изменяется их относительный порядок. Таким образом, необходимо изменить значение атрибута *ordinal* для существующего элемента *author*. В результате менеджер блокировок установит на него эксклюзивную блокировку. Получается, что вставка узла приводит к блокировке всех узлов-братьев, находящихся справа от него. Очевидно, что создаст множество искусственных конфликтов с параллельными транзакциями на чтение.

Дополнительная проблема связана с тем, что менеджер блокировок в РСУБД, используя блокировки на уровне кортежей, иногда не может обеспечить сериализацию транзакций. Действительно, две параллельные транзакции, запускающие один и тот же запрос $InsertInto(< author/ >, /book)$, не будут конфликтовать в РСУБД. Однако на самом деле конфликт существует, поскольку перестановка этих операций влияет на порядок вставленных узлов. Единственное возможное решение - использовать блокировки на уровне таблицы. Очевидно, что это слишком грубое решение, которое вообще заблокирует все читающие транзакции. Кроме того, как правило, в коммерческих РСУБД есть средство “сделать подсказку” (hint) менеджеру блокировок, какой уровень блокировок использовать. Но при этом совершенно необязательно, что он будет использовать эту подсказку.

Сценарий 3: удаление листового узла

При описании этого сценария мы подразумеваем, что для метода хранения используется метод Inlining (напомним, что согласно результатам сравнительного анализа, этот метод является наиболее эффективным для хранения XML-документов). Этот метод сокращает количество дорогостоящих операций соединения за счет встраивания улов-детей в кортеж родителя. Однако это приводит к возникновению дополнительных конфликтов, поскольку,

вообще говоря, разные узлы собираются в одну гранулу.

Рассмотрим пример. Пусть транзакция T_1 выполняет запрос на чтение */Dept/Student/@st_id*, а транзакция T_2 выполняет запрос на изменение *Delete (/Dept/Student[@st_id="123"]/Name)* (см. рисунок 1.9). С логической точки зрения конфликта между этими транзакциями нет, но в РСУБД он возникнет, поскольку для элемента *Student* атрибут *st_id* и элемент *Name* хранятся в одном кортеже. При этом транзакции T_1 необходимо установить на этот кортеж разделяемую блокировку, а транзакции T_2 установить на него же эксклюзивную блокировку.

Сценарий 4: удаление поддерева

В этом сценарии мы рассмотрим транзакцию T, которая удаляет поддерево *Student* с именем “st1”: *Delete(/Dept/Student[Name=“st1”])*. Мы вновь будем предполагать, что используется метод Inlining, хотя описанные ниже проблемы относятся и к другим методам.

В работе [56] показывается, что рассматриваемое удаление поддерева может быть реализовано следующими двумя операторами SQL:

```
DELETE FROM Student WHERE Name="st1"


DELETE FROM Enroll
WHERE ParentID NOT IN (SELECT ID FROM Student)
```

Однако второй оператор SQL заблокирует все кортежи в таблице `Student`, как минимум, в разделяемом режиме, что является избыточным. Фактически, удаление поддерева приводит к блокировке в разделяемом режиме всех таблиц, в которых содержатся узлы-потомки. В результате, например, транзакция, которая присваивает новое имя студенту с идентификатором “st2”, будет конфликтовать с исходной транзакцией T.

Сценарий 5: путевые выражения с предикатом

В этом сценарии мы рассмотрим транзакцию, которая выполняет путевое выражение с предикатом: */Dept/Student[@st_id=“123”]/Enroll[. !=“CS20”]/text()*. Этот запрос транслируется в SQL-запрос в котором есть предикат-соединение и два предиката на значения узлов. Например для метода *Inlining* он будет выглядеть следующим образом:

```
SELECT TEXT
FROM Student S, Enroll E
WHERE S.st_id="123"
```

```
and E.TEXT != "CS20"
and S.ID = E.ParentID
```

Вероятная стратегия выполнения этого запроса может быть такова, что сначала на таблицы `Student` и `Enroll` будут наложены предикаты `S.st_id="123"` и `E.TEXT != "CS20"` соответственно, а затем для выбранных кортежей будет произведена операция соединения. Эта стратегия соответствует плану в котором предикаты "опущены" и поэтому вполне может быть выбрана оптимизатором в качестве оптимальной.

Однако при такой стратегии на кортежи, которые не будут удовлетворять предикату-соединения, будут наложены разделяемые блокировки. Так, если бы у студента с идентификатором *"124"* были бы элементы *Enroll*, не удовлетворяющие предикату `E.TEXT != "CS20"`, то они были бы заблокированы. Причем в оригинальном путевом выражении они не адресуются.

Фактически получается, что операции соединения могут заблокировать кортежи, которые не адресуются в исходном путевом выражении. С учетом того, что операция соединения выполняется несколько раз даже для простых путевых выражений, менеджер блокировок может создавать большое количество дополнительных искусственных конфликтов.

2.1.2 Блокировки в РСУБД для XML-документов при использовании типа XML или метода STORED

При использовании для хранения XML-документа типа `XML` текущие реализации коммерческих РСУБД не используют каких-то специальных блокировок внутри иерархии XML (например, это явно указано в технической статье про MS SQL Server 2005 [28]). Используется обычный менеджер блокировок, который при обращении к XML-документу блокирует кортеж, в котором находится этот XML-документ. Таким образом, в этом случае получается гранулированность на уровне всего XML-документа.

При использовании метода STORED ситуация примерно такая же. Например, при проведении операции изменения в любом случае РСУБД устанавливает эксклюзивную блокировку на весь BLOB/CLOB объект, даже если необходимые данные присутствуют в дополнительных индексированных таблицах. Это делается по той причине, что необходимо поддерживать BLOB/CLOB объект и дополнительные таблицы в консистентном состоянии.

2.2 XML-транзакции в прирожденных XML-СУБД

В отличие от СУБД, в которых поддержка XML построена на основе уже существующих механизмов управления данными, в прирожденных XML-СУБД все базовые механизмы управления данными должны строиться с нуля. Это включает в себя и механизм управления транзакциями. С одной стороны, это требует больших усилий, но, с другой стороны, при разработке таких механизмов можно учитывать как особенности иерархической структуры XML, семантику XML-операций, организацию внешней памяти и методы обработки XML-данных в оперативной памяти, так и основные приложения прирожденных XML-СУБД.

Поэтому в этом разделе мы выделим некоторые важные приложения XML-СУБД, а затем рассмотрим существующие специальные методы управления XML-транзакциями в прирожденных XML-СУБД.

2.2.1 Основные приложения XML-СУБД

На сегодняшний день одним из самых популярных приложений XML-СУБД являются Web-приложения [88], основанные на XML. Популярность XML-СУБД в данном случае объясняется тем, что функциональный язык XQuery, являющийся важнейшим компонентом технологии управления XML-данными, может использоваться не только в качестве языка запросов к XML-данным, но и как средство для выражения логики приложения, основанного на технологии XML. Прежде всего этот подход удобен тем, что преодолевается проблема *потери соответствия* [93], которая возникает при одновременном использовании нескольких языков программирования, основанных на разных моделях данных и парадигмах. Например, если разработчик использует язык Java для выражения логики приложения и язык XSLT для визуализации данных[1], то он вынужден заботиться об отображении между объектно-ориентированной моделью данных и моделью данных XML. Обеспечение такого отображения может приводить, как к усложнению кода приложения, так и к падению эффективности приложения.

Специфика Web-приложений заключается в том, что в подавляющем большинстве запросов к XML-СУБД осуществляется лишь выборка данных. Иными словами большинство транзакций в этом случае являются читающими. Система должна минимизировать время ожидания пользователя. Это требование, в свою очередь, отражается на механизме управления транзакциями. Так, в случае использовании двухфазного протокола блокирования (2PL) время ожидания читающих транзакций может быть велико в случае наличия в системе параллельных изменяющих транзакций.

[1] Например, XSLT используется для преобразование XML-данных в HTML-страницы.

Другим важнейшим приложением XML-СУБД являются системы генерации статистических отчетов. Читающие транзакции в этом случае являются достаточно "длинными". Их время работы может составлять несколько часов или более. Кроме того, такие транзакции считывают огромные массивы данных для построения статистики. Очевидно, что на время выполнения этих транзакций должна быть возможность запускать изменяющие транзакции.

2.2.2 Обзор родственных работ по изоляции XML-транзакций

За последние несколько лет исследователями баз данных были предложены различные методы синхронизации операций над XML-данными. Ниже мы приводим обзор этих методов, а также обсуждаем их достоинства и недостатки.

Иерархический и древовидный протоколы

Первая группа методов основана на существующих иерархических [58] (метод гранулированных блокировок) и древовидных протоколах синхронизации [44]. При предварительном рассмотрении этих методов можно подумать, что они подходят для синхронизации операций над XML-данными, поскольку соответствуют иерархической структуре XML-документов. Например, протокол синхронизации XML-транзакций, предложенный в работе Грабса [48], основывается на гранулированных блокировках. Однако при детальном анализе мы пришли к выводу, что данные методы не учитывают в полной мере специфику XML-данных и операций над ними, и поэтому их применение может приводить к существенному понижению параллелизма конкурентных XML-транзакций.

Протокол гранулированных блокировок (ПГБ) был разработан для синхронизации иерархических структур, в которых один элемент включается в другой элемент, тот в свою очередь включается в третий элемент и т. д. Примером такой иерархии является структура реляционной базы данных: база данных включает в себя набор отношений, каждое отношение состоит из набора страниц[2], каждая станица в свою очередь состоит из набора кортежей. ПГБ подразумевает, что операции над данными могут выполняться на разных уровнях иерархии. Например, транзакция T_1 может удалять отношение R, а другая транзакция T_2 параллельно читать некоторые кортежи из этого отношения. Прямым решением для предотвращения конфликта между T_1 и T_2 было бы заблокировать все кортежи на изменение из R транзакцией T_1. В этом случае при попытке чтения какого-либо кортежа транзакция T_2 будет пытаться заблокировать необходимые кортежи на чтение, что приведет к конфликту между транзакциями. ПГБ предлагает более эффективное

[2]В некоторых системах появляется еще один уровень - сегмент.

Запрашиваемая блокировка	Установленная блокировка				
	S	X	IS	IX	SIX
S	+	-	+	-	-
X	-	-	-	-	-
IS	+	-	+	+	+
IX	-	-	+	+	-
SIX	-	-	+	-	-

Рис. 2.1: Таблица совместимости гранулированных блокировок

решение подобной проблемы при котором не нужно устанавливать огромное количество блокировок на запись для транзакции T_1. Для этого в протоколе вводится новый тип блокировок - блокировки намерений (intention locks). *Блокировка намерения* представляет намерение транзакции захватить общую блокировку (на чтение или изменения) вложенного элемента данных. Таким образом, перед тем, как захватить общую блокировку на некоторый элемент данных E, транзакция должна захватить блокировку намерений на каждый элемент данных, который включает в себя элемент данных E. Если транзакции необходимо читать (изменять) элемент E, то в начале необходимо установить блокировку намерения на чтение (изменение) IS (IX) для всех элементов E', включающих в себя элемент E, начиная с корня. Кроме того, для удобства вводится новая блокировка SIX (блокируется вся гранула в разделяемом режиме и устанавливается намерение на изменения внутри гранулы). Таблица совместимости блокировок S, X, IS, IX и SIX изображена на рисунке 2.1.

На Рисунке 2.1 важно отметить, что блокировки IS и IX совместимы, поскольку, установив эти блокировки на элемент E', транзакции только выразили намерение читать и изменять элементы, вложенные в него. В этом случае конфликт может выявиться при установке блокировок на более нижних уровнях иерархии. Совместимость блокировок IX и IX объясняется аналогичным образом.

Блокировки IS и X не совместимы, поскольку IS блокировка на элемент E' означает намерение транзакции читать некоторые элементы, вложенные в E', в то время как вся иерархия E' включая вложенные элементы заблокирована в эксклюзивном режиме. С другой стороны, если транзакция установила IS на элемент, то она собирается читать некоторый вложенный элемент и это не совместимо с эксклюзивным захватом всей иерархии. Несовместимость блокировок IX и S объясняется аналогичным образом.

Совместимость (или несовместимость) остальных пар блокировок очевидна. Важно заметить, что ПГБ удовлетворяет общему правилу двухфазного протокола, при котором блокировки освобождаются только после захвата всех блокировок, необходимых транзакции.

Протокол гранулированных блокировок не подходит для XML, поскольку в XML

предполагается, что на всех уровнях документа хранятся данные, и при этом типичной операцией является выборка значения узла на некотором уровне, а предки этого узла не рассматриваются. При использовании же гранулированных захватов всегда неявно захватывается все поддерево, что для XML во многих случаях является избыточным. Рассмотрим несколько примеров.

Пример 2. *Предположим, что транзакция* T_1 *читает все элементы* item, *определяемые путевым выражением* /site/regions//item, *в документе XMark [94]. В это же время другая транзакция* T_2 *намерена переименовывать элемент* africa *на* south-africa*:* Rename(/site/regions/africa, south-africa)*. Вообще говоря, транзакции* T_1 *и* T_2 *могут выполняться параллельно, поскольку конфликтов между ними нет, но применение ПГБ приведет к конфликту между ними, поскольку переименование узла* africa *требует установки блокировки* X *на этот узел транзакцией* T_2*, которая не совместима с блокировкой* IS*, которую должна установить на этот же узел транзакция* T_1*. Установка* IS *блокировки транзакцией* T_1 *необходима, поскольку она намерена прочитать узлы* item, *которые узел* africa *включает в себя.*

Пример 3. *Предположим, что транзакция* T_1 *читает элементы* name, *определяемые путевым выражением* /site/people/person/name, *в документе XMark. В это же время другая транзакция* T_2 *намерена вставить новый элемент* <person/>*:* InsertInto(<person/>, /site/people)*. Логических конфликтов между транзакциями* T_1 *и* T_2 *нет, но применение ПГБ ведет к конфликту между ними на узле* person*. Транзакция* T_1 *должна установить на него* IS *блокировку, а транзакция* T_2 *должна установить несовместимую с ней блокировку* X*.*

На примерах 2 и 3 можно видеть, что ПГБ не учитывает семантические особенности операций над XML-данными и приводит к появлению псевдоконфликтов между транзакциями.

Древовидный протокол (ДП). Мы рассматривали ПГБ для иерархических структур, в которых мелкие элементы базы данных вложены в более крупные элементы. В ДП рассматриваются древовидные структуры, образуемые за счет связывания элементов между собой. Примером такой структуры является Б-дерево. При этом предполагается, что доступ к элементу данных A осуществляется посредством обращения к некоторой вершине (как правило это корень иерархии) с последующим перемещением вниз по поддереву к элементу A. Этот факт позволяет отойти от общего правила двухфазного блокирования. ДП формулируется следующими четырьмя пунктами:

- Первый запрос на блокирование, инициируемый транзакцией, может относиться к любой вершине дерева.

- Последующие запросы на блокирование должны удовлетворяться только в том случае, если транзакция обладает блокировкой вершины, родительской по отношению к текущей.
- Операции разблокирования разрешено выполнять в любые моменты времени.
- Транзакция не имеет возможности повторного захвата блокировки элемента после ее освобождения - даже в том случае, если транзакция удерживает блокировку на родительский элемент.

Важной отличительной чертой ДП от ПГБ является то, что блокировка некоторого элемента не предполагает блокирование потомков этого элемента.

На следующем примере демонстрируется, что применение ДП для XML-данных может приводить к псевдоконфликтам.

Пример 4. *Предположим, что транзакция запрашивает все элементы* item*, используя запрос* //item. *В этом случае при использовании ДП все узлы документа заблокируются в разделяемом режиме. Это слишком ограничивающее условие, которое не позволяет производить какие-либо операции модификации над эти документом. Например, операция переименования узла* site *на* site2 *не конфликтует с запросом* //item*, в то время как при использовании ДП эти операции, выполняемые разными транзакциями, не могут выполняться параллельно.*

Кроме того, ДП накладывает слишком строгие ограничения на направление перемещения по XML-документу: при переходе от одного узла к другому можно перемещаться только сверху–вниз. В то же время в XPath существуют оси, которые позволяют перемещаться по XML-документу влево, вправо, вверх и вниз от заданного узла. Таким образом, с использованием древовидного протокола можно поддерживать только ограниченное подмножество XPath.

Протоколы синхронизации для DOM-операций

Вторая группа методов была предложена для синхронизации DOM-операций. На сегодняшний день существует несколько протоколов для синхронизации DOM-операций: Node2PL, NO2PL и taDOM. Первые два протокола были предложены в работе Свена Хелмера [73], а третий протокол (и его оптимизации) в работах Питера Хаустина [71, 72]. Сравнительный анализ всех трех протоколов приводится в работе Питера Хаустина [74]. Автор этой статьи приходит к выводу, что протокол *taDOM* значительно превосходит

протоколы Node2PL и NO2PL, и поэтому далее мы рассмотрим именно *taDOM* протокол. Но прежде чем переходить к описанию этих методов, мы кратко опишем DOM-операции.

DOM API [24] обеспечивает способ доступа к XML-документу, основанный на древовидном представлении документа. Иными словами, с точки зрения пользователя DOM API, XML-документ является ациклическим направленным графом. Для работы с ним предоставляется набор стандартных операций, с помощью которых можно обходить граф узел за узлом вдоль ребер, добавлять, удалять или модифицировать узлы. Помимо этого, есть способы выполнять простые запросы. Различаются три типа доступа к документу: навигационный, доступ на изменение и через запрос.

Навигационный доступ предоставляют такие методы DOM API, как *getAttributes()*,*getChildNodes*, *getFirstChild()*, *getNextSibling()* и *getParentNode()*. Для изменения используются методы *insertBefore()*, *removeChild()*, *setNodeValue()* или *setAttribute()*. Простейшие запросы для поиска узлов и их атрибутов предоставляются такими DOM-методами, как *getElementById()*, *getElementsByTagName()* или *hasAttribute()*.

Когда DOM API инициализируется для XML-документа, весь документ анализируется и создается DOM-дерево в основной памяти. Этот способ предоставляет быстрый доступ к узлам, однако очень дорог с точки зрения потребления памяти. Для сравнения приведем такой факт: документ размером в 20 МБ после анализа и создания DOM-дерева может занимать до 400 МБ. Поэтому различные реализации предоставляют способы оптимизации, например, такие, как отложенную обработку XML документа. Другие системы запоминают обработанный документ на диске и отображают его части в память. В любом случае встает вопрос об изоляции параллельных DOM-транзакций над XML.

Протокол taDOM В протоколе *taDOM* предполагается, что XML-документ представляется в виде специального *taDOM*-дерева. Его пример из статьи [72] приведен на рисунке 2.2, где представлен XML-документ, описывающий библиотеку.

Для синхронизации доступа к элементам документа описываемый алгоритм использует различные типы блокировок на узлах taDOM-дерева. Узел, с которым сейчас работает DOM-операция, мы будем называть рабочим. Перед тем как работать с ним, мы должны получить соответствующую блокировку, вид которой зависит от типа доступа к узлу. Блокировки taDOM следуют правилам для гранулированных захватов: операция не может получить доступ к узлу, если она не владеет захватами на все вышестоящие узлы, вплоть до корня дерева.

Теперь рассмотрим типы блокировок на taDOM-дереве:

- NR (node read). Запрашивается для получения доступа на чтение к рабочему узлу.

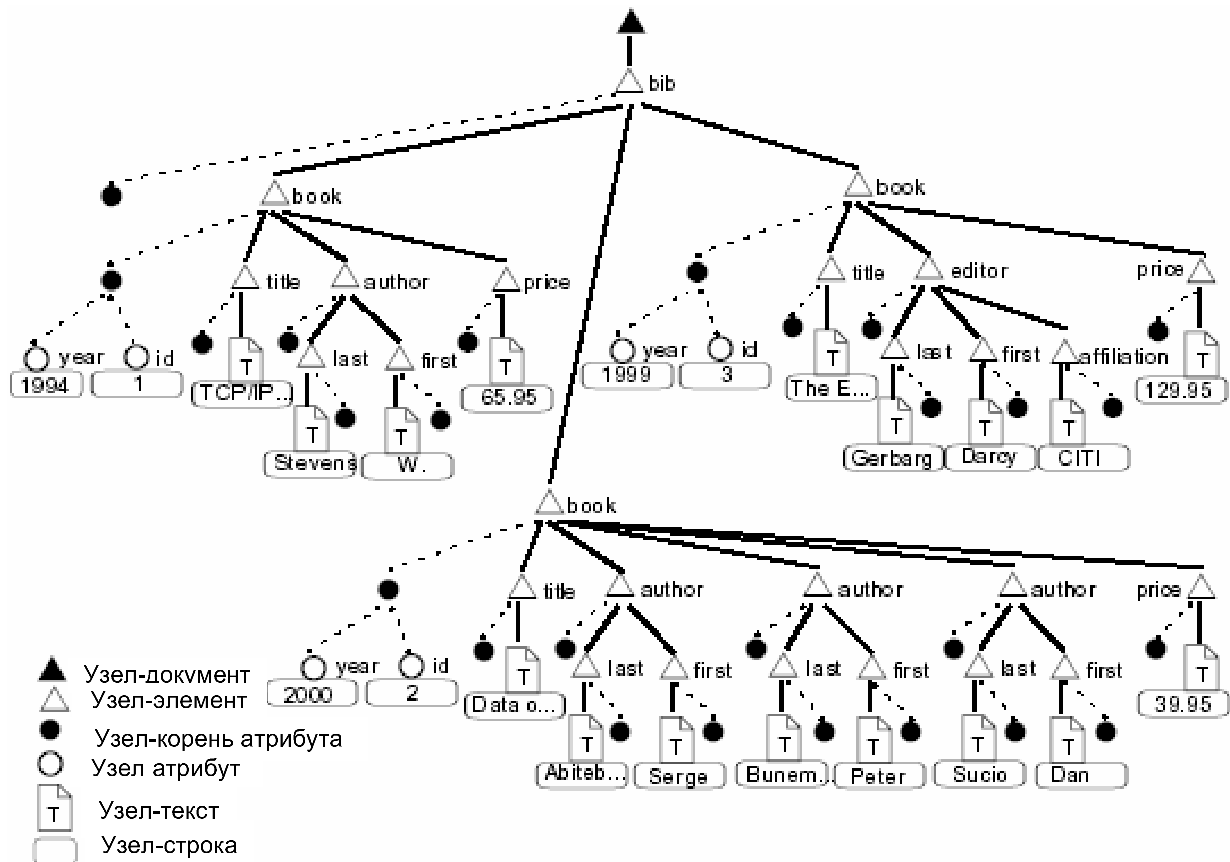


Рис. 2.2: taDOM дерево

- LR (level read). Эта блокировка блокирует целый уровень узлов. Например, такую блокировку требуется запросить для выполнения операции *getChildNodes()*. Она будет установлена на рабочий узел. При получении этого захвата, операции уже не требуется получать блокировки NR для каждого из дочерних узлов.

- Блокировка SR (sub-tree read) запрашивается для получения доступа на чтение для всего поддерева рабочего узла.

- Для изменения узла (модификации его содержания или удаления всего поддерева вместе с узлом) DOM-операции требуется блокировка X. Она также требует наличия блокировок CX (child exclusive) на родительском узле и блокировок IX (intent exclisive) на всех других узлах-предках.

- наличие блокировки CX на узле означает захват X на одном из дочерних узлов. CX не совместима с LR.

- IX означает, что где-то в поддереве узла имеется модифицируемый узел (заблокированный на X). Блокировки IX и LR совместимы.

	-	IX	NR	CX	LR	SR	U	X
IX	+	+	+	+	+	-	-	-
NR	+	+	+	+	+	+	-	-
CX	+	+	+	+	-	-	-	-
LR	+	+	+	-	+	+	-	-
SR	+	-	+	-	+	+	-	-
U	+	+	+	+	+	+	-	-
X	+	-	-	-	-	-	-	-

Рис. 2.3: Матрица совместимости taDOM блокировок

- Захват U устанавливается на поддерево, если это поддерево блокируется на чтение, и при этом возможно изменение каких-то из дочерних узлов. Он не совместим с другими блокировками на чтение и может быть усилен до X, как только стало возможно получить необходимые блокировки (IX, CX) на предков. Он также может быть ослаблен до одной из блокировок на чтение.

На рисунке 2.3 приведена матрица совместимости для описанных блокировок.

Описанный протокол хорошо подходит для синхронизации параллельных DOM-операций. Но в современных XML-СУБД доступ к данным, как правило, осуществляется при помощи высокоуровневых языков запросов XQuery/XUpdate. Протокол для синхронизация XQuery/XUpdate операций отмечается в работе [72] как направление будущих исследований.

Кроме того, описанный протокол требует непосредственный доступ к дереву XML-документа, что в некоторых случаях может быть невозможно (например, при синхронизации доступа к XML-документам, хранимым в реляционных СУБД).

Протокол, основанный на блокировании путей

В работе Яна Хидерса [75] предложен протокол управления параллельными транзакциями, основанный на захвате путевых выражений. Прежде чем рассматривать протокол, мы введем некоторую нотацию, используемую в протоколе.

Мы предполагаем, что пользователь задает запросы с помощью подмножества языка XPath. Ниже приведена грамматика такого языка.

```
P::= F|F/P|F//P
```

```
F::=.|T|*|@A|@*|t|s
```

Множество F задает элементарные запросы над узлом. В P они комбинируются друг с другом в более сложные путевые выражения. Здесь T означает набор имен узлов, A-множество имен атрибутов, t обозначает функцию $text()$, возвращающую все дочерние текстовые узлы, а s – функцию $value()$, возвращающую строковое значение атрибута или текстового узла.

В данных обозначениях запрос Q удовлетворяет грамматике:

```
Q::= /P|//P|X/P|X//P
```

Здесь X-некоторое множество узлов, полученное на предыдущем шаге транзакции.

Операции, изменяющие данные, разбиваются на три группы: операции, изменяющие атрибуты, операции, изменяющие элементы, и операции манипулирующие значениями текстовых узлов. Перечислим все операции, принадлежащие этим группам.

Операции модификации атрибутов:

1. *create-attribute(e-id, a-name, string)* создает атрибут с именем *a-name* и строковым значением *string* в узле с идентификатором *e-id*.

2. *delete-attribute(a-id)* удаляет атрибут с идентификатором *a-id*.

3. *update-attribute(a-id, string)* изменяет значение атрибута с идентификатором *a-id* на строку *string*.

Операции модификации элементов:

1. *create-element-under(e-id, tagname)* создает пустой элемент с именем *tagname*, следующий за последним дочерним узлом с идентификатором *e-id*.

2. *delete-leaf-element(e-id)* удаляет листовой элемент с идентификатором *e-id*. Операция применима лишь к листовым узлам.

Операции для работы с текстовыми узлами:

1. *create-text-under(e-id, string)* создает текстовый узел, следующий за последним дочерним узлом с идентификатором *e-id*.

2. *delete-text(t-id)* удаляет текстовый узел с идентификатором *t-id*.

3. *update-text(t-id, string)* изменяет значение текстового узла с идентификатором *t-id* на строку *string*.

Рассматриваемый протокол использует для управления параллельными транзакциями так называемые путевые захваты. Как и в 2PL, есть два базовых типа блокировок – на чтение и на запись. Однако, помимо типа захвата, они снабжаются путевым выражением и типом операции. Благодаря этой информации в ряде случаев удается избежать синхронизационных конфликтов.

Итак, блокировка на чтение представляет из себя кортеж (n, p), где n – идентификатор узла, на который ставится блокировка, а p – путевое выражение. Такой захват будет затребован транзакцией, читающей данные с помощью путевого запроса с началом в узле n.

Блокировки на запись выглядят по-другому. Такая блокировка имеет вид: (n, f), где n – идентификатор узла, а f – один из элементов множества элементарных запросов F (оно описано выше). Соответствие описанным выше операциям приводится ниже.

Операции модификации атрибутов:

1. *create-attribute(n, a, v)* : операция требует блокировки $@a$ на n.
2. *delete-attribute(n)* : требуется блокировка $@a$ на родительский узел n, где a – имя атрибута соответствующего n.
3. *update-attribute(n, v)* : операция требует блокировки s на n.

Операции модификации элементов:

1. *create-element-under(n, tname)* : для создания узла требуется блокировка *tname* на n.
2. *delete-leaf-element(n)* : требуется установить блокировку на *tname* в родительском узле n, где *tname* - имя удаляемого узла.

Операции для работы с текстовыми узлами:

1. *create-text-under(n, v)* : для создания текстового узла требуется блокировка t на родительский узел.
2. *delete-text(n)* : для удаления узла нужна такая же блокировка.
3. *update-text(n, v)* : для модификации текстового значения требуется блокировка s на n.

Теперь опишем правила совместимости блокировок:

1. Две блокировки на чтение всегда совместимы.

2. Блокировка на чтение *(n, p)* конфликтует с блокировкой на запись *(n', f)* тогда и только тогда, когда *n* является предком *n'* и результат путевого запроса *path(n,n')/f* входит в число целевых узлов путевого выражения *p*, аннотирующего читающий захват.

3. Блокировка на запись *(n, f)* всегда конфликтует с другой блокировкой на запись *(n, f')* в том же узле.

Этот протокол ориентирован на путевые выражения, однако в нем рассматривается очень ограниченное подмножество XPath, в котором нет предикатов, а оси ограничиваются самыми базовыми (*child* и *descendant-or-self*). Кроме того, операция проверки блокировок на совместимость является дорогой операцией по времени. Наконец, авторы не приводят какую-либо экспериментальную оценку предложенного протокола.

2.3 Выводы

Мы рассмотрели существующие методы управления параллельными XML-транзакциями, как в РСУБД, так и в прирожденных XML-СУБД.

В реляционных СУБД поддержка XML-транзакций строится на основе существующих методов, которые были разработаны для реляционных данных. Проанализировано большое количество различных сценариев, в которых несколько XML-транзакций запускаются параллельно, и сделан вывод, что менеджер блокировок в РСУБД порождает огромное количество псевдоконфликтов, что негативно влияет на общую производительность системы. В частности, время отклика транзакций может существенно увеличиться. Поэтому для управления параллельными XML-транзакциями в РСУБД нужна дополнительная надстройка над существующим механизмом управления транзакциями, которая позволяла бы учитывать иерархическую структуру XML и семантику XML-операций при определении конфликтов между ними.

В прирожденных XML-СУБД методы управления параллельными XML-транзакциями находятся в зачаточном состоянии. Существует сравнительно небольшое количество работ, посвященных этой проблеме. Самые интересные и законченные методы предлагаются для синхронизации DOM-операций. Однако в развитых прирожденных XML-СУБД доступ к данным, как правило, осуществляется при помощи декларативных языков XQuery/XUpdate, и поэтому методы синхронизации должны ориентироваться именно на эти языки и учитывать именно их специфику. Кроме того, все предложенные методы основываются на 2PL протоколе, который плохо подходит для классических приложений прирожденных XML-СУБД - Web приложений, в которых большинство транзакций являются читающими.

Действительно, одна достаточно “длинная” читающая транзакция может заблокировать доступ на изменения к базе данных, что во многих случаях является недопустимым.

Основным выводом этой главы является то, что в XML-ориентированных СУБД необходима разработка новых методов управления XML-транзакциями, учитывающих специфику иерархической структуры XML, операций над XML-данными и XML-приложений.

Глава 3

Протокол изоляции XML-транзакций XDGL

Настоящая глава посвящена описанию и обоснованию предлагаемого автором протокола изоляции XML-транзакций XDGL, который учитывает иерархическую структуру XML-данных и семантические особенности операций над XML-данными при определении конфликтов между конкурентными XML-транзакциями.

3.1 Введение

В вводной главе были описаны основные методы изоляции транзакций, предложенные на данный момент в литературе. Самым популярным и хорошо зарекомендовавшим себя является метод, основанный на блокировках. В наиболее популярных СУБД, таких как MS SQL Server, IBM DB2, Sybase используется строгий двухфазный протокол синхронизационных блокировок (S2PL)[1], в соответствии с которым все блокировки, установленные транзакцией, освобождаются при ее фиксации. Автор диссертационной работы также основывается на протоколе S2PL, как основном правиле установки и освобождения блокировок в XDGL [2].

При разработке протокола XDGL (XPath-based DataGuide Locking) автор учитывал тот факт, что при его реализации внутреннее представление XML-документа в базе данных может быть недоступно. Поэтому в XDGL блокировки устанавливаются не на узлах самого XML-документа, а на узлах дополнительной структуры, которая является структурным обобщением XML-документа. В литературе [85] эту структуру принято

[1]Исключением является Oracle, который использует версионный протокол.

[2]При этом XDGL-блокировки могут использоваться также и в многоверсионных протоколах (например в ROMV) для изоляции изменяющих транзакций.

называть описывающей схемой[3] XML-документа. В описывающей схеме представлены все пути в XML-документе, и только они. Поскольку одному пути, как правило, соответствуют много узлов в XML-документе, то размер описывающей схемы гораздо меньше размера соответствующего ей XML-документа. Поэтому дополнительные издержки, связанные с поиском необходимых узлов в описывающей схеме, а также установкой блокировок на схеме, сравнительно небольшие.

Кроме того, использование описывающей схемы XML-документа позволяет абстрагироваться от системы хранения XML-документа, что, в свою очередь, ведет к универсальности XDGL: он может быть реализован над любой системой хранения XML-документов. В главах 4 и 5 мы рассматриваем реализацию XDGL над реляционной и прирожденной системами хранения XML-документов.

В XDGL выделяется несколько типов ограничений, накладываемых на XML-документ. Для каждого типа ограничений существуют соответствующий тип блокировки.

К первому типу относятся ограничения на структуру XML-документа. Например, ограничение такого типа присутствует в запросе `/doc/person/name`, который выбирает все элементы `name`, являющиеся детьми элемента `person`, который, в свою очередь, является ребенком корневого элемента `doc`. Здесь отношение родитель-ребенок узлов определяет структурное ограничение. Этому типу ограничений соответствуют *структурные блокировки*.

Ко второму типу относятся ограничения на значения узлов в XML-документе. Эти ограничений выражаются при помощи предикатов. Например, в запросе `/doc/person[@age > 30]` предикат `@age > 30` определяет ограничение на значение атрибута `@age`. Этому типу ограничений соответствуют *предикатные блокировки*.

Кроме того, в XDGL вводятся *логические блокировки* для предотвращения фантомов [86]. Логические блокировки являются комбинацией структурных и предикатных блокировок.

Глава организована следующим образом. В разделе 3.2 вводятся основные определения и обозначения, используемые далее в этой главе. Раздел 3.3 посвящен рассмотрению семантических особенностей операций над XML-данными (выраженных на языках XQuery и XUpdate), которые влияют на выбор типов блокировок, необходимых в протоколе изоляции XML-транзакций. В разделе 3.4 формально вводятся все типы блокировок, используемые в XDGL. Раздел 3.5 посвящен описанию алгоритма работы XDGL-планировщика. Наконец, в разделе 3.6 приводится формальное обоснование протокола XDGL. В разделе 3.7 рассматриваются дополнительные оптимизации протокола XDGL при наличии DTD. Раздел

[3]Далее в диссертационной работе неуточняемый термин “схема” будет обозначать описывающую схему XML-документа.

3.8 посвящен обсуждению примеров использования XDGL.

3.2 Основные определения и обозначения

В этом разделе мы вводим базовые определения и обозначения, которые мы будем использовать далее в этой главе.

Определение 1. *Путем для некоторго узла n в XML-документе DOC называется последовательность пар (тип_узла, имя_узла)*[4]*, таких что (1) первая пара последовательности представляет собой описание корневого узла XML-документа DOC, (2) последняя пара последовательности представляет собой описание узла n, (3) для любых двух подряд идущих пар в последовательности первая описывает узел, являющийся родительским для узла, который описывает вторая.*

Используя понятие пути в XML-документе, мы вводим понятие описывающей схемы XML-документа [5].

Определение 2. *Описывающей схемой XML-документа DOC называется XML-документ DG, который обладает двумя свойствами: (1) для каждого пути в DOC существует и, причем единственный, путь в DG, (2) для каждого пути в DG существует путь в DOC.*

На рисунке 3.1 изображен XML-документ *GTree* (слева), его описывающая схема (в центре) и предписывающая схема DTD (справа). Далее в этой главе все примеры обсуждаются для этого XML-документа. Отметим также, что далее предписывающую схему мы будем изображать графически в виде дерева.

Ниже перечислены ряд обозначений и сокращений, используемых в этой главе:

- Символ LP обозначает путевое выражение (locpath), а символ Q обозначает произвольное выражение на XQuery;
- Символ R обозначает произвольную операцию чтения;
- Символы II, IA и IB обозначают операции $InsertInto$, $InsertAfter$ и $InsertBefore$ соответственно. Символ $I*$ обозначает операцию вставки узла произвольного типа;
- Символ D обозначает операцию удаления узла $Delete$;
- Символ RN обозначает операцию переименования узла $Rename$;

[4] Здесь *тип_узла* соответствует типу узла в модели данных языка XQuery [19].

[5] В англоязычной литературе [85] термин описывающая схема иногда называют путеводителем по данным (DataGuide).

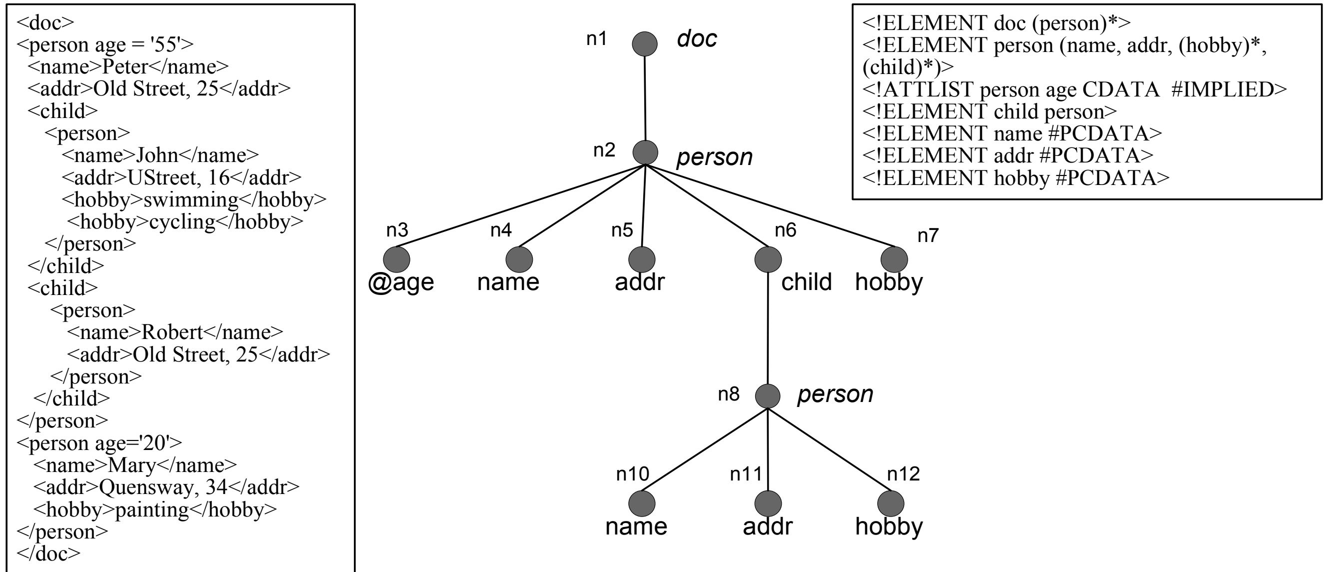


Рис. 3.1: XML-документ *GTree* и его описывающая и предписывающая схемы

- Символ W обозначает произвольную операцию изменения;
- Символ C обозначает фиксацию транзакции. Напомним, что при фиксации транзакции освобождаются все блокировки, установленные транзакцией;
- Символ n_i^{DOC} (n_i^{DG}) обозначает некоторый узел в исходном XML-документе (описывающей схеме XML-документа);
- Символ $PATH(n_i^{DOC})$ обозначает путь до узла n_i^{DOC};
- Операция $DST(LP)$ определяет последовательность узлов, выбираемых LP на последнем шаге. Фактически операция DST определяет результат выражения LP;
- Операция $PREFIX(p)$ определяет множество путей, которые являются префиксами пути p;
- Операция $ITM(LP)$ определяет последовательность узлов, выбираемых LP на промежуточных шагах (всех шагах за исключением последнего). Кроме того, каждый из полученных узлов n_i^{DOC} удовлетворяет условию: $\exists n_j^{DOC} \in DST(LP)$: $PATH(n_i^{DOC}) \in PREFIX(PATH(n_j^{DOC}))$;
- Операция $DGN(\{n_i^{DOC}\})$ определяет множество узлов описывающей схемы, которым соответствуют узлы из множества узлов $\{n_i^{DOC}\}$ в исходном XML-документе;
- Операция $TARG(op)$ обозначает целевые узлы операции op;

- $NEWN(op)$ обозначает новые узлы создаваемые операцией op. В нашем наборе операций изменения XML-документа это операции RN и $I*$. Поэтому $NEWN$ определена только для этих операций;

- Операция $DSTND(\{n_i^{DOC}\})$ возвращает множество всех узлов XML-документа, которые являются потомками узлов из множества $\{n_i^{DOC}\}$;

- Предикат $INTERNAL(n_i^{DG})$ выдает истину, если узел схемы n_i^{DG} является внутренним, и выдает ложь, если узел n_i^{DG} является листовым.

3.3 Семантические особенности языков XQuery/XUpdate

В этом разделе рассматриваются семантические особенности операций XQuery и XUpdate, которые необходимо учитывать при разработке протокола изоляции XML-транзакций. Эти особенности влияют на выбор типов синхронизационных блокировок, требуемых в протоколе изоляции XML-транзакций. Учет этих особенностей позволяет уменьшить гранулированность синхронизационных захватов и увеличить параллелизм конкурентных транзакций.

3.3.1 Путевые выражения

Путевое выражение (locpath) является базовой конструкцией, используемой как в XQuery, так и в XUpdate. Поэтому для получения наилучшего параллелизма транзакций в протоколе должна максимально учитываться семантика путевых выражений.

Путевое выражение состоит из последовательности шагов, на каждом из которых требуются различные типы блокировок. Например, на последнем шаге выбираются узлы, содержимое которых сериализуется [21] и передается пользователю. Поэтому для таких узлов необходимо установить блокировку на чтение для поддеревьев (глубокую блокировку) с корнями в этих узлах. С другой стороны, для узлов, выбираемых на промежуточных шагах, достаточно установить блокировку на чтение только на сам узел (неглубокую блокировку).

В дополнение к этому, необходимо гарантировать, что не удерживаются глубокие блокировки на запись на поддеревьях, включающих узлы или поддеревья, которые мы хотим заблокировать на чтение. Эта проблема решается при помощи блокировок намерения (intention locks) [58]. Перед тем, как устанавливать блокировку на узел n, необходимо установить блокировку намерения на каждого предка узла n, начиная с корня. Кроме того, если на очередном шаге путевого выражения осуществляется фильтрация узлов при помощи предикатов, то для некоторых узлов выполняется операция атомизации [21]. На узлы, для

которых выполняется операция атомизации необходимо устанавливать глубокую блокировку, поскольку процедура атомизации обходит все поддерево, соотвествующее атомизируемому узлу.

3.3.2 Запросы на XQuery

В XQuery-запросах обращение к данным осуществляется при помощи путевых выражений. Поэтому и блокировки в XQuery-запросах должны устанавливаться на основе анализа путевых выражений, записанных в XQuery-запросе. Однако путевые выражения в XQuery-запросе не должны рассматриваться изолированно от всего XQuery-запроса, поскольку учет контекста путевого выражения в XQuery-запросе позволяет во многих случаях существенно улучшить гранулированность блокировок. Дело в том, что в выражении XQuery не для всех узлов, выбираемых locpath, требуется блокировать все поддерево. Например, в выражении `fn:count(//person)` необходимо гарантировать неизменяемость количества узлов `person`, но при этом потомки элементов person могут изменяться произвольным образом другими транзакциями. В качестве еще одного примера рассмотрим FLWR-выражение: `for $v in //person return $a/name`. Здесь блокировочный механизм не должен обеспечивать неизменяемость всех поддеревьев `person`. Вместо этого достаточно гарантировать, что сами элементы `person` не будут изменяться другими транзакциями (при этом потомки могут изменяться), и что не будут изменены поддеревья `name`.

3.3.3 Операция вставки новых узлов

Анализ блокировок для операции вставки нового узла должен осуществляться на основе двух аргументов: выражения locpath, которое специфицирует целевые узлы операции, а также выражения constr2, которое определяет новые узлы. Анализ блокировок для locpath мы обсуждали выше. Для новых узлов необходимо установить неглубокую блокировку на запись. Например, для нового узла `<person/>` требуется установить монопольную блокировку только на узел `person`, но при этом не нужно блокировать узлы, находящиеся ниже по иерархии. Для узла `<name>John</name>` возможны два варианта блокировок: можно установить монопольную неглубокую блокировку на узлы `name` и `text` (ребенок узла `name`) либо монопольную глубокую блокировку на узел `name`. Дополнительной семантической особенностью операции вставки (II) является следующее: при вставке нового узла в позицию последнего дочернего узла целевого узла необходимо гарантировать, что другие транзакции не будут вставлять другие узлы в это же место. Очевидно, что можно установить глубокую блокировку (на чтение) на узел, в который вставляется новый узел, и тем самым гарантировать, что в него не будут вставлены новые узлы, но это слишком грубое решение.

Здесь лучше ввести дополнительную структурную блокировку: неглубокую блокировку на чтение, которая дополнительно предотвращает вставку новых узлы в блокируемый узел. Аналогично, для операций IA и IB требуется ввести неглубокие блокировки на чтение, которые дополнительно предотвращают вставку новых узлов после и перед блокируемым узлом соответственно. В остальном к операциям IA и IB применимы те же рассуждения, что и к операции II.

3.3.4 Операция удаления узлов

Операция D выполняет глубокое удаление целевых узлов. Поэтому необходимо запретить чтение или модификацию удаляемых поддеревьев другими транзакциями. Следовательно, на удаляемые узлы необходимо установить монопольную глубокую блокировку.

3.3.5 Операция переименования узлов

Операция переименования присваивает новое имя узлу, но при этом не изменяет его потомков. Поэтому нужно установить монопольную неглубокую блокировку на изменяемый узел, а также на узел с новым именем. При этом не требуется устанавливать блокировки на потомков переименовываемого узла. В результате, например, транзакции Rename (/doc/person, person2) и /doc//name не будут конфликтовать по блокировкам.

3.4 XDGL-блокировки

3.4.1 Структурные блокировки

В XDGL выделяется два типа структурных блокировок: узловые и древовидные. Узловые блокировки накладывают ограничения только на выполнение операций над узлами XML-документа, соответствующих блокируемому узлу схемы, а древовидная блокировка неявно накладывает ограничения на выполнение операций над поддеревьями в XML-документе, соответствующих блокируемому узлу. Ниже определяется семантика всех структурных блокировок в XDGL.

- Блокировка P *(pass)*. Эта узловая блокировка используется в путевых выражениях LP. P блокировка устанавливается на узлы $DGN(ITM(LP))$ - узлы схемы, которые соответствуют промежуточным узлам путевого выражения LP. Предположим, что P блокировка установлена на множество узлов $\{n_j^{DG}\}$. Это приводит к запрету выполнения каких либо модификаций узлов n_i^{DOC}, которые в нем существовали до

установки P блокировки и удовлетворяют условию $DGN(n_i^{DOC}) \subseteq \{n_j^{DG}\}$. Но P блокировка не запрещает выполнение W операций, которые приводят к появлению *новых* узлов n_i^{DOC}, удовлетворяющих условию $DGN(n_i^{DOC}) \subseteq \{n_j^{DG}\}$. Также P блокировка не накладывает какие-либо ограничения на выполнение W операций, которые производят произвольные модификации узлов n_i^{DOC}, не удовлетворяющих условию $DGN(n_i^{DOC}) \subseteq \{n_j^{DG}\}$. Наконец P блокировка не накладывает какие либо ограничения на чтение (выполнение R операций) произвольных узлов n_i^{DOC}.

- Блокировка *S (share).* Эта узловая блокировка также используется в путевых выражениях LP. S блокировка устанавливается на узлы $DGN(DST(LP))$ - узлы схемы, которые соответствуют конечным узлам путевого выражения LP. Предположим, что S блокировка установлена на множество узлов $\{n_j^{DG}\}$. Это приводит к запрету выполнения каких-либо модификаций узлов n_i^{DOC}, которые удовлетворяют условию $DGN(n_i^{DOC}) \subseteq \{n_j^{DG}\}$. Также S блокировка не накладывает какие-либо ограничения на выполнение W операций, которые производят произвольные модификации узлов n_i^{DOC}, не удовлетворяющих условию $DGN(n_i^{DOC}) \subseteq \{n_j^{DG}\}$. Наконец, S блокировка не накладывает какие либо ограничения на чтение (выполнение R операций) произвольных узлов n_i^{DOC}.

- Блокировка *SI (shared into).* Эта узловая блокировка используется в операции I_I. SI блокировка устанавливается на узлы $DGN(TARG(I_I))$ - узлы схемы, которые соответствуют целевым узлам операции I_I. Предположим, что SI блокировка установлена на множество узлов $\{n_j^{DG}\}$. Семантика блокировки SI совпадает с семантикой блокировки S за исключением одного дополнительного условия. SI блокировка запрещает выполнение операций вставки новых узлов n_i^{DOC} *в* узлы, удовлетворяющие условию $DGN(n_i^{DOC}) \subseteq \{n_j^{DG}\}$. Иными словами, запрещается выполнение конкурентных операций I_I над целевыми узлами рассматриваемой операции I_I. Это позволяет предотвратить конфликты, связанные с порядком следования узлов в XML-документе.

  Аналогично определяются блокировки *SA (shared after)* и *SB (shared before)*, которые запрещают вставку новых узлов *после* и *перед* целевыми узлами операций I_A и I_B соответственно.

- Блокировка *X (exclusive).* Эта узловая блокировка используется в W операциях, которые изменяют *внутренние* узлы в XML-документе. В нашем наборе операций модификации это операция RN. Блокировка X устанавливается на узлы $DGN(\{TARG(RN) \cup NEWN(RN)\})$. Предположим, что X блокировка установлена

на множество узлов $\{n_j^{DG}\}$. Это приводит к запрету выполнения каких-либо чтений или модификаций узлов n_i^{DOC}, которые удовлетворяют условию $DGN(n_i^{DOC}) \subseteq \{n_j^{DG}\}$. Но X блокировка не накладывает какие-либо ограничения на выполнение R или W операций, которые производят произвольные чтения или модификации узлов n_i^{DOC}, не удовлетворяющих условию $DGN(n_i^{DOC}) \subseteq \{n_j^{DG}\}$.

- Блокировка *XN (exclusive new).* Эта узловая блокировка используется в $I*$ операциях, которые создают новые узлы в XML-документе. Блокировка XN устанавливается на узлы схемы $DGN(NEWN(I*))$. Предположим, что XN блокировка установлена на множество узлов $\{n_j^{DG}\}$. Семантика блокировки XN совпадает с семантикой блокировки X за исключением одного ослабляющего условия. XN блокировка не запрещает выполнение операций *вставки новых узлов* и *тестовое чтение* [6] узлов n_i^{DOC}, которые удовлетворяют условию $DGN(n_i^{DOC}) \subseteq \{n_j^{DG}\}$.

- Блокировка *ST (shared tree).* Эта древовидная блокировка используется как в R, так и в W операциях. Предположим, что ST блокировка установлена на множество узлов $\{n_j^{DG}\}$. Это приводит к запрету выполнения каких либо операций модификации узлов n_i^{DOC}, которые удовлетворяют условию $DGN(n_i^{DOC} \cup DSTND(n_i^{DOC})) \subseteq \{n_j^{DG}\}$. Операции модификации, не удовлетворяющие этому условию, и операции чтения произвольных узлов XML-документа блокировкой ST не запрещаются.

- Блокировка *XT (shared tree).* Эта древовидная блокировка используется операцией D. Блокировка XT устанавливается на узлы $DGN(TARG(D))$ - узлы схемы, которые соответствуют целевым узлам операции D. Предположим, что XT блокировка установлена на множество узлов $\{n_j^{DG}\}$. Это приводит к запрету выполнения каких либо операций чтения или модификации узлов n_i^{DOC}, которые удовлетворяют условию: $(DGN(n_i^{DOC} \cup DSTND(n_i^{DOC})) \subseteq \{n_j^{DG}\}$. Операции чтения и модификации, не удовлетворяющие этому условию, блокировкой XT не запрещаются.

- Блокировки *IS (intention shared)* и *IX (intention exclusive).* Блокировку IS (IX) необходимо установить для каждого предка узла (начиная с корня), на который должна быть установлена одна из разделяемых (эксклюзивных) блокировок. Семантика блокировок намерений такая же, как и в стандартном DAG [58] протоколе. IS (IX) блокировка гарантирует отсутствие эксклюзивных (разделяемых и эксклюзивных)

[6] Мы будем считать, что в путевом выражении LP для множества узлов $ITM(LP)$ производится тестовое чтение, а для множества узлов $DST(LP)$ производится целевое чтение. Термин "чтение" без указание его типа обозначает любой из вышеуказанных типов чтения.

	Установленная блокировка										
Запрашиваемая блокировка	S	P	SI	SA	SB	XN	X	ST	XT	IS	IX
S	+	+	+	+	+	ср	ср	+	ср	+	+
P	+	+	+	+	+	+	ср	+	ср	+	+
SI	+	+	ср	+	+	ср	ср	+	ср	+	+
SA	+	+	+	ср	+	ср	ср	+	ср	+	+
SB	+	+	+	+	ср	ср	ср	+	ср	+	+
XN	ср	+	ср	ср	ср	+	ср	ср	ср	+	+
X	ср	ср	ср	ср	ср	ср	ср	ср	ср	+	+
ST	+	+	+	+	+	ср	ср	+	ср	+	ср
XT	ср	ср	ср	ср	ср	ср	ср	ср	ср	ср	ср
IS	+	+	+	+	+	+	+	+	ср	+	+
IX	+	+	+	+	+	+	+	ср	ср	+	+

Рис. 3.2: Матрица совместимости структурных блокировок XDGL

блокировок на поддеревьях на верхних уровнях схемы. В нашем протоколе IS (IX) гарантирует отсутствие ST (ST или XT) блокировок на узлах-предках схемы.

Таблица совместимости структурных блокировок показана на Рисунке 3.2. Символ "+" на рисунке обозначает совместимость блокировок. Символ "ср" обозначает условную несовместимость типов блокировок, и решение о наличии или отсутствии конфликта должно приниматься на основе предикатов, которые мы обсудим в следующем подразделе.

Ниже мы обсуждаем матрицу совместимости структурных блокировок в XDGL.

Рассмотрим столбец S матрицы. Совместимость с блокировками S, SI, SA и SB очевидна, поскольку установка этих блокировок на узел схемы n^{DG} только предполагает чтение узлов в XML-документе, соответствующих узлу n^{DG}. Условная несовместимость S с XN объясняется тем, что, установив блокировку S на n^{DG}, транзакция запрещает появление новых узлов в документе, соответствующих узлу n^{DG}. Условная несовместимость S и X блокировки объясняется тем, что S блокировка на узле n^{DG} запрещает какие-либо модификации узлов в XML-документе, соответствующих узлу n^{DG}. Аналогично объясняется условная несовместимость блокировки S с XT и XN. Отметим, что блокировка S совместима с блокировкой IX, поскольку IX блокировка предполагает модификацию узлов на более низких уровнях иерархии, а S блокировка не распространяется на более низкие уровни.

Рассмотрим столбец P матрицы. Единственное отличие от столбца S заключается в том, что P блокировка совместима с XN блокировкой. Это объясняется тем, что P блокировка на узел n^{DG} не запрещает вставку в XML-документ *новых* узлов n_i^{DOC}, удовлетворяющих условию $n^{DG} \subseteq DGN(n_i^{DOC})$.

Блокировка SI отличается от блокировки S только тем, что она несовместима сама с собой тем самым предотвращая вставки в один узел конкурентными транзакциями.

Аналогично, блокировки SA и SB, запрещают какие-либо вставки другими транзакциями перед и после рассматриваемого узла соответственно.

Рассмотрим столбец XN матрицы. Совместимость с блокировкой P мы уже обсуждали при рассмотрении столбца P. Совместимость с блокировками IS и IX также очевидна, поскольку блокировка XN является узловой. Совместимость XN блокировки с XN блокировкой объясняется тем, что, установив блокировку XN на n^{DG}, транзакция не запрещает вставку новых узлов n_i^{DOC}, которые соответствуют узлу n^{DG}.

Совместимость блокировок для столбцов X, ST, XT, IS, IX очевидна из вышесказанного.

3.4.2 Предикатные блокировки

Блокировка узла схемы в каком-либо режиме неявно приводит к блокировке в этом режиме всех узлов в XML-документе, которые соответствуют данному узлу схемы. Очевидно, что во многих случаях это будет приводить к блокировке лишних узлов XML-документа. Например, для запроса */price[. = 100]* не нужно блокировать все узлы *price*, а достаточно заблокировать узлы *price*, значение которых равно *100*.

Для решения указанной проблемы мы с каждой структурной блокировкой ассоциируем предикат. Этот предикат накладывает ограничение на значение узла в XML-документе. Таким образом, блокировка в XDGL состоит из двух частей: типа блокировки и предиката. Две блокировки *(lock-type1, pred1)* и *(lock-type2, pred2)* совместимы, если выполняется одно из условий:

1. типы блокировок *lock-type1* и *lock-type2* совместимы согласно таблице совместимости;

2. предикаты *pred1* и *pred2* совместимы (т.е конъюнкция предикатов *pred1* и *pred2* тождественно равна *false*).

Вообще говоря, проверка совместимости предикатов является NP-трудной задачей [90], и поэтому в XDGL мы проверяем на совместимость только простые предикаты (сравнения с константами), для которых существуют полиномиальные алгоритмы проверки на совместимость (например такой алгоритм описан в работе Ханта [91]), а остальные предикаты отождествляем с *true*(для краткости мы будем обозначать эту константу символом #t).

3.4.3 Логические блокировки

В этом подразделе мы описываем логические блокировки, которые в XDGL используются для предотвращения фантомов. Прежде всего рассмотрим, в каких ситуациях могут появляться

фантомы [86].

Предположим, что транзакция T_1 читает все атрибуты *age* в XML-документе $GTree$ (см. рисунок 3.1), используя запрос *//@age*. В это же время транзакция T_2 вставляет новый атрибут *age*, используя операцию *InsertInto(attribute age {30}, /doc/person/child/person)*. При повторном чтении всех атрибутов *age* транзакция T_1 прочитает новый атрибут-фантом *age*, вставленный второй транзакцией. В общем случае фантомы могут появляться в случае, если выполняются два следующих условия:

- Операция модификации расширяет описывающую схему XML-документа (добавляет новый путь).
- Эта модификация приводит к изменению целевых узлов предыдущих операций чтения или модификации других транзакций.

Основная сложность предотвращения фантомов заключается в том, что нельзя заранее установить блокировку на узел, которого еще не существует.

В контексте XML наиболее вероятной операцией, повторное выполнение которой может привести к появлению фантомов, является путевое выражение с осями $descendant/descendant-or-self$. Также фантом может появиться в путевом выражении, в одном из шагов которого используется '*', поскольку вставка нового узла по схеме также может приводить к изменению целевых узлов операций. Наконец, фантом может появляться в случае, если в путевом выражении LP конструкция $nameTest$ содержит имя узла, которое не присутствует в описывающей схеме XML-документа. В этом случае, если узел с таким именем затем появится, то повторное выполнения выражения LP приведет к появлению фантома.

Чтобы избежать появление узлов-фантомов, мы вводим две дополнительные блокировки: L и IN. Логическая блокировка L устанавливается на узел схемы для предотвращения появления фантомов в поддереве, соответствующему этому узлу. При установке L-блокировки необходимо специфицировать набор свойств для узлов в документе. Свойства узлов документа выражаются при помощи условий на имя узла и его значение. L-блокировка запрещает вставку новых узлов в поддерево, если вставка приводит к расширению схемы XML-документа, и вставляется узел со свойствами, указанными в L-блокировке. В свою очередь, транзакция, которая расширяет схему, должна установить блокировку IN (insert new node) для каждого предка вставляемого узла. При требовании установки IN-блокировки также специфицируется набор свойств вставляемого узла.

Ниже мы рассматриваем все комбинации для свойств L-блокировки, которые мы рассматриваем в протоколе XDGL:

1. *node-name='name1'* (например, для запроса $//person$);
2. *node-name='name1'*, *node-value relop 'val1'* (например, для запроса $//name[. \neq' John']$);
3. *node-name='name1'*, *child-name='name2'*, *child-value relop 'val1'* (например, для запроса $//person[name \neq' John']$).

Здесь *node-name* обозначает имя узла, *node-value* - значение узла, *child-name* - имя дочернего узла, *child-value* - значение дочернего узла, и, наконец, *relop* обозначает операцию сравнения.

Для проверки того, что свойства вставляемого узла не конфликтуют со свойствами *L*-блокировки, установленной другой транзакцией, в *IN*-блокировке необходимо указать три свойства вставляемого узла: *new-node-parent-name* - имя родительского узла, *new-node-name* - имя вставляемого узла и *new-node-value* - значение вставляемого узла.

L- и *IN*-блокировки не совместимы в каждом из следующих случаев:

- Свойство *new-node-name* *IN*-блокировки равно свойству *node-name* *L*-блокировки, и при этом *L*-блокировка не содержит других свойств (случай 1).
- Свойства *new-node-name* и *new-node-value* *IN*-блокировки соответствуют значениям свойств *L*-блокировки (которых должно быть два); иными словами, *node-name=new-node-name* и *new-node-value relop 'val1'* $\neq \#f$ (случай 2).
- Все три свойства *IN*-блокировки соответствуют трем свойствам *L*-блокировки; иными словами, *node-name=new-node-parent-name*, *child-name=new-node-name* и *new-node-value relop 'val1'* $\neq \#f$ (случай 3).

Заметим, что если в *L*-блокировке вместо имени узла указано '*', то это имя узла считается соответствующим произвольному имени.

3.5 XDGL-планировщик

В этом разделе мы описываем алгоритм работы XDGL-планировщика. Мы предполагаем, что XDGL-планировщику на вход дается пара (op_i, tid_k) (где op_i - это операция на языке XQuery/XUpdate, а tid_k - идентификатор транзакции, которая намерена выполнить операцию op_i), а на выходе XDGL-планировщик либо приостанавливает выполнение транзакции с идентификатором tid_k, либо разрешает выполняться операции op_i. При этом побочным эффектом работы планировщика является установка блокировок на узлы описывающей схемы XML-документа. Итак, шаги XDGL-планировщика выглядят следующим образом.

1. Вычислить множество всех путей в схеме, приводящих к данным, которые читаются или изменяются операцией op_i. Обозначим это множество символом DP (data-path-set);

2. Вычислить множество узлов $\{n_j^{DG}\}$ схемы, которые соответствуют конечным узлам путей из DP. Далее, если на узел n_j^{DOC} операция op_i накладывает предикат p_j, то ассоциировать этот предикат с узлом схемы $DGN(n_j^{DOC})$. Обозначим полученное множество узлов, соответствующих конечным узлам из DP с ассоциированными предикатами, символом $NP = \{(n_j^{DG}, p_j)\}$ (node-predicate-set);

3. Вычислить множество всех узлов n_j^{DG} схемы (и их свойства $properties_j$) таких, что в соответсвующих поддеревьях XML-документа могут появиться фантомы (см. раздел 3.4.3). Обозначим полученное множество символом $PH = \{(n_j^{DG}, properties_j)\}$ (phantom-set);

4. Если op_i расширяет схему, то вычислить свойства нового узла. Обозначим эти свойства символом $properties_i$;

5. Вычислить множество $\{LP_i\}$, которое содержит все путевые выражения (как абсолютные, так и относительные) из операции op_i.

6. Для каждого n_j^{DG}, удовлетворяющего условию $n_j^{DG} \in \{ITM(LP_i)\} \wedge INTERNAL(n_j^{DG})\}$, установить блокировку *(P, #t)*, а также на каждого предка узла n_j установить блокировку *(IS, #t)*.

7. По следующим правилам установить структурные блокировки для операции op_i:

   - Пусть op_i – это операция Q. Для каждого $n_j^{DG} \in NP$ выполнить:
     (a) если n_j^{DG} соответствует узлу, атомизируемому при выполнении op_i, то установить на узел n_j^{DG} блокировку (ST, p_j);
     (b) если n_j^{DG} соответствует узлу, не атомизируемому при выполнении op_i, то установить на узел n_j^{DG} блокировку (S, p_j);
     (c) для каждого предка узла n_j^{DG} установить блокировку *(IS, #t)*.
   - Пусть op_i – это операция II. Для каждого $n_j^{DG} \in NP$ выполнить:
     (a) если n_j^{DG} соответствует целевым узлам операции II, то установить блокировку (SI, p_j) на узел n_j^{DG} и блокировку *(IS, #t)* на всех его предков;
     (b) если n_j^{DG} соответствует дополнительным ветвям путевых выражений LP (необходимых для указания предиката) в операции II, то установить на узел n_j^{DG} блокировку (ST, p_j) (либо блокировку (S, p_j), если для проверки предиката не требуется атомизация узла) и блокировку *(IS, #t)* на его предков;

(c) если n_j^{DG} соответствует новому узлу, который вставляется операцией II, то установить на узел n_j^{DG} блокировку (XN, p_j) и блокировку *(IX, #t)* на его предков.

- Если op_i - это IA или IB операция, то выполнить шаги, аналогичные предыдущему пункту.
- Пусть op_i – это операция D. Для каждого $n_j^{DG} \in NP$ выполнить:

  (a) если n_j^{DG} соответствует целевым узлам операции D, то установить на узел n_j^{DG} блокировку (XT, p_j) и блокировку *(IX, #t)* на его предков;

  (b) если n_j^{DG} соответствует дополнительным ветвям путевых выражений LP (необходимых для указания предиката) в операции D, то установить на узел n_j^{DG} блокировку (ST, p_j) (либо блокировку (S, p_j), если для проверки предиката не требуется атомизация узла) и блокировку *(IS, #t)* на его предков.
- Пусть op_i – это операция RN. Для каждого $n_j^{DG} \in NP$ выполнить:

  (a) если n_j^{DG} соответствует целевым или новым узлам операции RN, то установить на узел n_j^{DG} блокировку (X, p_j) и блокировку *(IX, #t)* на его предков;

  (b) если n_j^{DG} соответствует дополнительным ветвям путевых выражений LP (необходимых для указания предиката) в операции RN, то установить на узел n_j^{DG} блокировку (ST, p_j) (либо (S, p_j), если для проверки предиката не требуется атомизация узла) и блокировку *(IS, #t)* на его предков.

8. Для каждого $n_j^{DG} \in PH$ установить блокировку $(L, properties_j)$.
9. Если op_i расширяет схему, то установить блокировку $(IN, properties_i)$ для всех предков узла, расширяющего схему.
10. Если требуемую блокировку установить невозможно (она не совместима с уже установленной блокировкой), то блокировать выполнение операции op_i до тех пор, пока не будет освобождена конфликтующая блокировка.
11. Освободить все блокировки, установленные транзакцией, при ее фиксации.

Замечание 1. *Важно также заметить, что мы предполагаем, что ни одна операция не удаляет узлы схемы. Вместо этого периодически запускается процедура удаления узлов схемы, на которые не установлена ни одна блокировка и которым не соответствует ни один узел в XML-документе.*

3.6 Обоснование корректности протокола XDGL

В этом разделе мы приводим формальное обоснование корректности протокола XDGL. Доказывается теорема о сериализуемости планов, генерируемых XDGL-планировщиком. Но прежде, чем переходить к доказательству, введем несколько определений.

Определение 3. *Транзакцией T_i мы будем называть последовательность пар (op_j, T_i). Причем последней в этой последовательности всегда является операция C.*

Определение 4. *Планом S множества транзакций $T = \{T_1, T_2, ..., T_n\}$ мы будем называть перемешанную последовательность операций транзакций из T (при этом порядок операций в транзакциях не меняется).*

Определение 5. *Мы будем обозначать символом L_i^S множество всех блокировок, установленных транзакциями после выполнения i-го шага в плане S.*

Определение 6. *Мы будем говорить, что план S является допустимым, тогда и только тогда, когда на каждом шаге i в плане S множество всех установленных блокировок L_i^S содержит только совместимые блокировки (разных транзакций).*

Определение 7. *Планы S и S' мы будем называть эквивалентными, если (1) план S является перестановкой плана S' (при этом порядок операций в транзакции сохраняется), (2) полученный документ (документы), после выполнения планов S и S' одинаковый (одинаковые), и (3) все запросы в плане S возвращают те же ответы, что и соответствующие запросы в плане S'.*

Определение 8. *План S называется сериальным, если для произвольных двух транзакций T_i и T_j из плана S все операции транзакции T_i предшествуют (или следуют) всем операциям транзакции T_j.*

Определение 9. *План S является сериализуемым если он эквивалентен какому-нибудь сериальному плану.*

Для того, чтобы доказать, что XDGL-планировщик корректен, мы должны показать, что все планы, которые он может сгенерировать, являются сериализуемыми. Доказательство состоит из двух шагов. На первом шаге в леммах 1, 2 и 3 мы докажем ряд свойств XDGL-планировщика, а затем на втором шаге, используя эти свойства, мы докажем сериализуемость планов, генерируемых XDGL-планировщиком.

Мы подразумеваем, что транзакции в плане S упорядочены следующим образом: $T_i < T_j$, если операция фиксации транзакции T_j следует за операцией фиксации транзакции T_i в

плане S, либо в плане S присутствует операция фиксации для транзакции T_i и отсутствует операция фиксации для транзакции T_j. Согласно этому порядку мы будем сериализовать план S, сгенерированный XDGL-планировщиком.

Введем несколько обозначений:

Определение 10. *Мы будем обозначать символом $l_i^S(op_i)$ множество всех блокировок, установленных (или освобожденных) в ходе выполнения операции op_i в плане S.*

Определение 11. *Мы будем обозначать символом D_i^S документ (или набор документов), полученный после выполнения i-ой операции в плане S.*

Далее в этом разделе мы будем считать, что план S' отличается от плана S перестановкой двух смежных пар (op_i, T_i) и (op_{i+1}, T_j) $(T_j < T_i)$.

Лемма 1. *Если план S является допустимым, то и план S' также является допустимым.*

Доказательство. *Очевидно, что S' является допустимым планом, если выполняются следующие условия: (1) все блокировки $L_i^{S'}$ совместимы, (2) $L_{i+1}^{S'} \subseteq L_{i+1}^S$.*

- *Пусть $op_i, op_{i+1} \in \{Q, I*, D, RN\}$. Поскольку Q, $I*$, D и RN операции не снимают блокировки, то $L_{i+1}^S = L_{i-1}^S \cup l_i^S(op_i) \cup l_{i+1}^S(op_{i+1})$ и $L_i^{S'} = L_{i-1}^{S'} \cup l_i^{S'}(op_{i+1}) \cup l_{i+1}^{S'}(op_i) = L_{i-1}^S \cup l_i^{S'}(op_{i+1}) \cup l_{i+1}^{S'}(op_i)$. Покажем, что $l_{i+1}^{S'}(op_i) = l_i^S(op_i)$. Факт, что $l_i^{S'}(op_{i+1}) = l_{i+1}^S(op_{i+1})$ доказывается аналогично.*

  Предположим, что $op_i, op_{i+1} \in \{Q, D\}$. Поскольку операции Q и D не изменяют схему, то $l_{i+1}^{S'}(op_i) = l_i^S(op_i)$.

  Добавим в рассматриваемое множество операцию $I*$, т.е. $op_i, op_{i+1} \in \{Q, I*, D\}$. Условие $l_{i+1}^{S'}(op_i) \neq l_i^S(op_i)$ может быть верным только в случае, если op_i затрагивает новые узлы схемы, вставленные операцией op_{i+1}. Иными словами, операция $op_{i+1} \in \{I*\}$, и при этом op_{i+1} расширяет схему. При этом согласно XDGL-планировщику операция op_{i+1} должна установить на новый узел схемы блокировку XN. Однако напомним, что операция $I*$ всегда вставляет листовые узлы. А на листовые узлы никогда не устанавливаются блокировки P, IS и IX. Поэтому операция op_i на новый узел схемы, вставленный операцией op_{i+1} может установить только блокировки S, SI, SA, SB, XN, X, ST или XT. Однако все эти блокировки, за исключением XN, не совместимы с блокировкой XN (при этом совместимости за счет предикатов нельзя добиться, поскольку точно есть пересечение по узлам документа), поэтому такого случая быть не может. А установка операцией op_i блокировки XN означает, что $op_i \in \{I*\}$ и при этом расширяет схему тем же самым новым узлом. Таким образом,

от перестановки операций op_i и op_{i+1} количество установленных этими операциями блокировками не изменяется, т.е. $l_{i+1}^{S'}(op_i) = l_i^S(op_i)$.

Наконец рассмотрим полное множество операций, т.е. $op_i, op_{i+1} \in \{Q, I*, D, RN\}$. Условие $l_{i+1}^{S'}(op_i) \neq l_i^S(op_i)$ $(l_i^{S'}(op_{i+1}) \neq l_{i+1}^S(op_{i+1}))$ может быть верным только в случае, если op_i затрагивает новые узлы схемы, вставленные операцией op_{i+1}. Иными словами, операция $op_{i+1} \in \{I*, RN\}$, и при этом op_{i+1} расширяет схему. Случай, когда $op_{i+1} \in \{I*\}$, рассматривается аналогично предыдущему пункту, поэтому осталось рассмотреть случай, когда $op_{i+1} \in \{RN\}$. В этом случае схема может расшириться не отдельным листовым узлом, а целым поддеревом. Согласно XDGL-планировщику, на каждый узел нового поддерева (и их предков) должна быть установлена логическая блокировка $(IN, new_schema_node_name)$. В свою очередь, если операция op_i читает узлы, соответствующие новому узлу поддерева, после выполнения операции op_{i+1}, и при этом она их не читает до выполнения op_{i+1}, то это потенциальные узлы-фантомы, и поэтому на более обширное поддерево, чем вставляемое поддерево, должна быть установлена логическая блокировка, которая будет конфликтовать с установленными IN блокировками операцией op_{i+1}. Таким образом, $l_{i+1}^{S'}(op_i) = l_i^S(op_i)$.

Мы показали, что $L_i^{S'} = L_i^S$ и $L_{i+1}^{S'} = L_{i+1}^S$, и, тем самым доказан факт (1).

- *Пусть $op_{i+1} \in \{C\}$ (если $op_i \in \{C\}$, то перестановка op_i и op_{i+1} не требуется). Поскольку операция C освобождает блокировки, то $L_{i+1}^S = L_{i-1}^S \cup l_i^S(op_i) \setminus l_{i+1}^S(op_{i+1})$ и $L_i^{S'} = L_{i-1}^{S'} \setminus l_i^{S'}(op_{i+1})$. Поскольку $l_{i+1}^S(op_{i+1}) = l_i^{S'}(op_{i+1})$, мы получаем, что $L_i^{S'} \subseteq L_{i+1}^S$. Таким образом доказаны факты (1) и (2).*

  Свойство $L_{i+1}^{S'} = L_{i+1}^S$, следует из того, что операция C не изменяет схему.

Таким образом, для всех операций мы доказали свойства (1) и (2). Лемма доказана.

□

Лемма 2. *Если план S является допустимым, и, по крайней мере, одна из операций op_i или op_{i+1} в плане S является запросом на выборку, то результаты выполнения этого (этих) запроса (запросов) на выборку в планах S и S' одинаковые.*

Доказательство. *Поскольку запрос на выборку не изменяет документ, то мы должны рассмотреть все комбинации запросов на выборку с операциями изменения XML-документов.*

- *$(I*, Q) \rightarrow (Q, I*)$. Операция $I*$ может изменить результат Q, в двух случаях: (1) $I*$ вставляет узлы, которые становятся целевыми для операции Q, (2) $I*$*

вставляет новые узлы, что косвенно приводит к изменению целевых узлов (например, изменяется значение предиката в каком-нибудь путевом выражении из Q *). Первый случай невозможен, поскольку* XN *блокировка не совместима с* S *или* ST *блокировками. Второй случай невозможен по той же причинам, поскольку для всех узлов, участвующих в вычислении предиката, устанавливаются* S *или* ST *блокировки. Кроме того,* L *и* IN *блокировки предотвращают появление фантомов.*

- *Рассуждения для перестановок* $(Q, I*) \rightarrow (I*, Q)$, $(D, Q) \rightarrow (Q, D)$, $(Q, D) \rightarrow (D, Q)$, $(Q, RN) \rightarrow (RN, Q)$, $(RN, Q) \rightarrow (Q, RN)$ *аналогичны предыдущему пункту.*

- $(Q, C) \rightarrow (C, Q)$. *Поскольку операция* C *не изменяет документ (она только освобождает блокировки), то перестановка операций* C *и* Q *не может изменить результат* Q.

- $(C, Q) \rightarrow (Q, C)$. *Эту перестановку не требуется делать для сведения плана* S *к сериальному.*

Лемма доказана.

□

Лемма 3. *Если план* S *является допустимым, то* $D^{S}_{i+1} = D^{S'}_{i+1}$.

Доказательство. *Поскольку операции* Q *не изменяют документ, то условие леммы выполнено, когда, как минимум, одна из операций* op_i *или* op_{i+1} *является запросом на чтение.*

- $(II, II) \rightarrow (II, II)$. *Существуют два случая, когда перестановка двух* II *операций может привести к изменению полученного документа: (1) операции* op_i *и* op_{i+1} *вставляют в один и тот же узел; в этом случае перестановка этих двух операций приведет к изменению порядка узлов в документе; (2)* op_{i+1} *вставляет новый узел в узел, созданный операцией* op_i*; в этом случае перестановка этих двух операций приведет к тому, что операция* op_{i+1} *не вставит какие либо узлы в документ, и поэтому документ, полученный после выполнения переставленных операций, изменится.*

  Однако первый случай невозможен, поскольку SI *блокировка не совместима с* SI *блокировкой (с учетом предикатов), а план* S *является допустимым. Второй случай невозможен, поскольку* op_{i+1} *требует* SI *блокировку на целевые узлы, но* SI *блокировка не совместима с* XN *блокировкой (с учетом предикатов), которая требуется* op_i *операцией на узел схемы, соответствующий вставляемому узлу.*

Аналогичным образом можно получить, что перестановки (IA, IA) *и* (IB, IB) *также не приводят к изменению полученного документа после выполнения* $i + 1$*-го шага.*

- $(II, IA) \rightarrow (IA, II)$. *Единственная спорная ситуация по сохранению порядка узлов в документе возникает, когда* IA *вставляет узел* n^{DOC_x} *как правого ребенка некоторого узла* n_y^{DOC}, *и операция* II *вставляет новый узел узел* n_z^{DOC} *в родительский узел для узла* n_y^{DOC}. *Однако очевидно, что такая вставка не приводит даже к нарушению порядка узлов в XML-документе.*

  Аналогично, перестановки остальных комбинаций II, IA *и* IB *операций не изменяют документ, полученный после выполнения* $i + 1$*-го шага.*

- $(I*, D) \rightarrow (D, I*)$. *Перестановка* $I*$ *и* D *операций может привести к изменению полученного документа на* $i + 1$*-м шаге только в случаях, если (1) операция* D *удаляет узлы, вставленные операцией* $I*$, *или (2) приводит к изменению целевых узлов операции* $I*$. *Однако для создаваемого узла на соответствующие узлы схемы устанавливается* XN *блокировка, и* IX *блокировка устанавливается на всех предков. А перед удалением узлов операция* D *должна установить* XT *блокировку на соответствующий узел схемы. Поскольку* XT *блокировка не совместима ни с* XN *блокировкой, ни с* IX *блокировкой (предикат не может сделать блокировки совместимыми поскольку есть пересечение по узлу, который вставляется операцией* $I*$ *и удаляется операцией* D*), случай (1) невозможен. Случай (2) также невозможен, поскольку для всех узлов, для которых делается тестовое или целевое чтение, устанавливается какая-нибудь блокировка на соответствующий узел схемы, а* XT *блокировка не совместима ни с одной другой блокировкой (с учетом предиката аналогично (1)).*

- $(D, I*) \rightarrow (I*, D)$, $(D, D) \rightarrow (D, D)$. *Этот случай аналогичен предыдущему.*

- $(I*, RN) \rightarrow (RN, I*)$. *Перестановка* $I*$ *и* RN *операций может привести к изменению полученного документа на* $i + 1$*-м шаге только в случаях, если (1) операция* RN *переименовывает узлы, вставленные операцией* $I*$, *или (2) приводит к изменению целевых узлов операции* $I*$. *Однако для создаваемого узла на соответствующие узлы схемы устанавливается* XN *блокировка. А перед переименованием узлов операция* RN *должна установить* X *блокировку на соответствующий узел схемы. Поскольку* X *блокировка не совместима с* XN *блокировкой (предикат не может сделать блокировки совместимыми, поскольку есть пересечение по узлу, который вставляется операцией* $I*$ *и переименовывается операцией* RN*). Случай (2) также невозможен, поскольку для всех узлов, для которых делается тестовое или целевое*

чтение устанавливается какая-нибудь блокировка на соответствующий узел схемы, а X блокировка не совместима ни с одной другой блокировкой (с учетом предиката аналогично (1)) за исключением блокировок IS и IX. Однако узлы, на которые устанавливаются блокировки IS и IX, могут изменяться как угодно (единственное предназначение этих блокировок - это не допустить, что будет удаляться целое поддерево на более высоких уровнях документа), и это не повлияет на целевые узлы II операции. Появление фантомов также невозможно из-за наличия L и IN блокировок. Таким образом случаи (1) и (2) невозможны.

- $(RN, I*) \rightarrow (I*, RN)$, $(D, RN) \rightarrow (RN, D)$, $(RN, D) \rightarrow (D, RN)$, $(RN, RN) \rightarrow (RN, RN)$. *Для всех этих перестановок доказательство проводится по схеме, аналогичной предыдущему пункту.*

- *Следующие перестановки:* $(C, I*) \rightarrow (I*, C)$, $(C, D) \rightarrow (D, C)$ *и* $(C, RN) \rightarrow (RN, C)$ *не требуются для сведения плана S к сериальному.*

- $(D, C) \rightarrow (C, D)$, $(I*, C) \rightarrow (C, I*)$, $(RN, C) \rightarrow (C, RN)$. *Очевидно, что эти перестановки не могут привести к изменению документа, полученного после выполнения $i+1$-го шага.*

Мы рассмотрели все возможные комбинации двух смежных операций op_i, op_{i+1} и доказали, что их перестановка не приводит к изменению документа после выполнения $i+1$-го шага, т.е $D^S_{i+1} = D^{S'}_{i+1}$. Лемма доказана.

□

Теорема 1. *Все планы S, сгенерированные XDGL-планировщиком, являются сериализуемыми.*

Доказательство. *Для доказательства этого факта мы будем последовательно приводить допустимый план S к сериальному плану S^{serial} посредством перестановки двух смежных пар (op_i, T_i) и (op_{i+1}, T_j), где $(T_j < T_i)$. Очевидно, что план является сериальным, если больше нет таких пар.*

Учитывая леммы 2 и 3, мы получаем, что планы S и S' эквивалентны. Здесь план S' отличается от плана S перестановкой двух смежных операций. В лемме 1 мы доказали, что S' является допустимым планом. Таким образом перемещение двух смежных пар в S' также приведет к эквивалентному плану S''. Поскольку план - это ограниченный набор пар, а две пары должны быть переставлены максимум один раз, мы заключаем, что S может быть приведен к S^{serial} за фиксированное количество перестановок. Теорема доказана.

□

3.7 Дополнительные оптимизации в XDGL

В этом разделе обсуждаются дополнительные оптимизации в XDGL, которые возможны при наличии предписывающей схемы XML-документа. Мы будем считать что предписывающая схема представляется в виде DTD, но на самом деле оптимизация возможна и при использовании других языков описания схемы.

В описывающей схеме теряется очень важная информация о документе – порядок следования узлов, в то время как из DTD такую информацию можно получить. Например, следующий фрагмент DTD:

```
<!ELEMENT person (name, age?, sex, salary?)>
```

показывает, что все дочерние узлы элемента *person* строго упорядочены: *age* всегда следует после *name*, *sex* – строго после *age* (или *name*, если *age* отсутствует), а необязательный элемент *salary* всегда завершает список дочерних узлов элемента *person*.

Информацию о порядке дочерних узлов можно использовать для оптимизации блокировок для осей *preceding-sibling* и *following-sibling*. Дело в том, что для пути *node-name/preceding-sibling::** необходимо установить блокировки S для всех "братьев" узла *node-name*, поскольку описывающая схема не обеспечивает информацию о том, какие узлы следуют за *node-name*. При наличии же информации о порядке узлов можно блокировать не всех "братьев" *node-name*. Например, для запроса *count(/person/sex/preceding-sibling::*)* на последнем шаге достаточно заблокировать в режиме S только *sex* и *salary*, а *name* и *age* можно не блокировать.

Другой вид оптимизации основан на том, что конфликты между блокировками на одном узле схемы могут разрешаться за счет предикатов на других узлах схемы. Рассмотрим пример. Предположим, что транзакция T_1 выполняет запрос */doc/person[name='Peter']/addr*, а в то же время другая транзакция T_2 выполняет запрос на изменение: *Delete /doc/person[name='Mary']/addr*. Транзакция T_1 должна установить блокировку $(ST, \#t)$ на узел n_5 ($addr$), а транзакция T_2 должна установить блокировку $(XT, \#t)$ на тот же узел. Поскольку ST и XT блокировки несовместимы, то транзакции T_1 и T_2 не могут выполняться параллельно. На самом деле, конфликта между транзакциями T_1 и T_2 нет, поскольку предикаты на узле *name* "разъединяют" узлы *person* (а следовательно и узлы $addr$). Действительно, транзакция T_1 запрашивает все элемены *addr* у персон с именем 'John', а транзакция T_2 удаляет узлы *addr* у персон с именем 'Mary'. Поскольку не может быть персон одновременно с именем, 'John' и 'Mary', то элементы *addr*, с которыми работают транзакции T_1 и T_2, различные.

Таким образом, при установке блокировок для путевых выражений, необходимо помечать все блокировки, соответствующие одному путевому выражению. Причем метки

для различных путевых выражений должны быть различными. Тогда, если на одном узле схемы блокировки конфликтуют, то конфликт может быть разрешен за счет проверки предикатов на других узлах (с теми же метками). Если предикаты на других узлах "разъединяют" узлы XML-документа, соответствующие узлу схемы, на котором произошел конфликт, то конфликт разрешается. Важно отметить, что при проверке того, что предикаты p_1 и p_2 "разъединяют" узлы n_1 и n_2, недостаточно проверить, что $p_1 \cap p_2 = \emptyset$. Важно еще убедиться, что предикаты p_1 и p_2 проверяются на узлах с множественностью не более одного. Поясним это на примере.

Пусть транзакция T_1 запускает запрос *Delete /doc/person/child/person[hobby='swimming'] /addr*, а транзакция T_2 выполняет модификацию *Delete /doc/person/child/ person[hobby='cycling']/addr*. В этом случае конфликт на узле n_{11} (*addr*) не может быть разрешен за счет предикатов на узле n_{12} (*hobby*), поскольку у человека может быть несколько увлечений (хобби), и предикат по увлечениям не "разъединяет" узлы *person*.

Информацию о множественности узла можно извлекать из схемы XML-документа. Таким образом, разрешение конфликтов за счет предикатов на других узлах может также проводиться только при наличии предписывающей схемы XML-документа.

3.8 Примеры использования протокола XDGL

В этом разделе мы рассмотрим несколько примеров, на которых демонстрируется протокол XDGL. Все примеры основываются на XML-документе GTree (см. рисунок 3.1). На рисунках для примеров мы показываем только часть узлов схемы, на которых транзакциями устанавливаются блокировки.

Пример 5 (чтение и удаление узлов-братьев)**.** *Предположим, что транзакция T_1 намерена прочитать все элементы name, выбираемые путевым выражением /doc/person/name. В это же время транзакция T_2 хочет удалить все элементы hobby, определяемые путем доступа: /doc/person/hobby: Delete(/doc/person/hobby). Очевидно, что логического конфликта между транзакциями нет, поскольку удаление узлов второй транзакцией не влияет на результат первой. Таким образом, транзакции T_1 и T_2, вообще говоря, могут выполняться одновременно. На рисунке 3.3 мы показываем, что протокол XDGL действительно позволяет выполнять транзакции T_1 и T_2 одновременно, поскольку между блокировками, необходимыми для выполнения T_1 и T_2, нет конфликтов.*

Пример 6 (конфликт двух операций вставки)**.** *Предположим, что транзакция T_1 намерена вставить новый элемент: InsertInto (<child/>, /doc/person). В это же время транзакция*

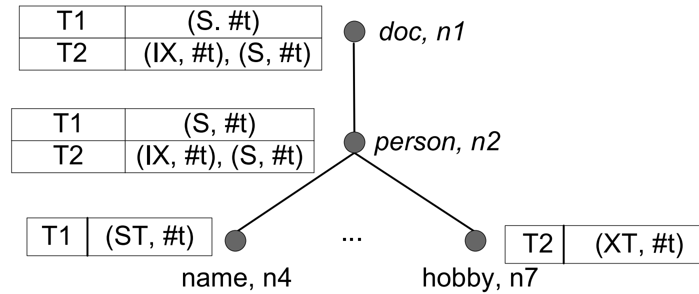


Рис. 3.3: Блокировки на схеме для примера 5

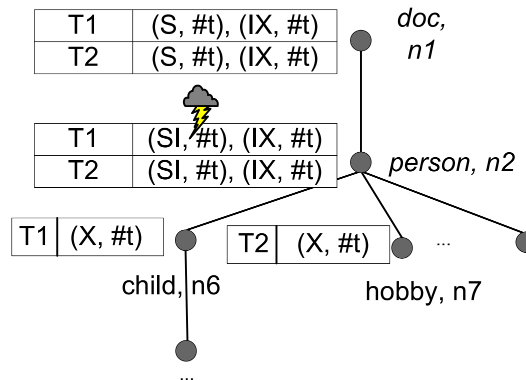


Рис. 3.4: Блокировки на схеме для примера 6

T_2 хочет вставить другой элемент: InsertInto(<hobby/> , /doc/person). Транзакции T_1 и T_2 не могут выполняться одновременно, поскольку они обе вставляют узлы в позицию последнего дочернего узла узлов person. Таким образом, возникает конфликт, связанный с упорядоченностью узлов в XML-документе. На рисунке 3.4 мы демонстрируем, что XDGL справляется с подобными конфликтами.

Пример 7 (вставка узла-фантома)**.** *Предположим, что транзакция T_1 намерена прочитать все атрибуты age, определяемые путевым выражением /doc/person//@age. В это же время другая транзакция T2 намерена вставить новый атрибут age: InsertInto(attribute age '54', /doc/person/ child/person). При одновременном выполнении транзакций возможно появление узла-фантома age для T_1. На рисунке 3.5 мы демонстрируем, что XDGL запрещает вставку узла-фантома age транзакцией T_2*

3.9 Выводы

В настоящей главе формально описан протокол XDGL изоляции конкурентных XML-транзакций, основанный на описывающей схеме XML-документа. Предложено

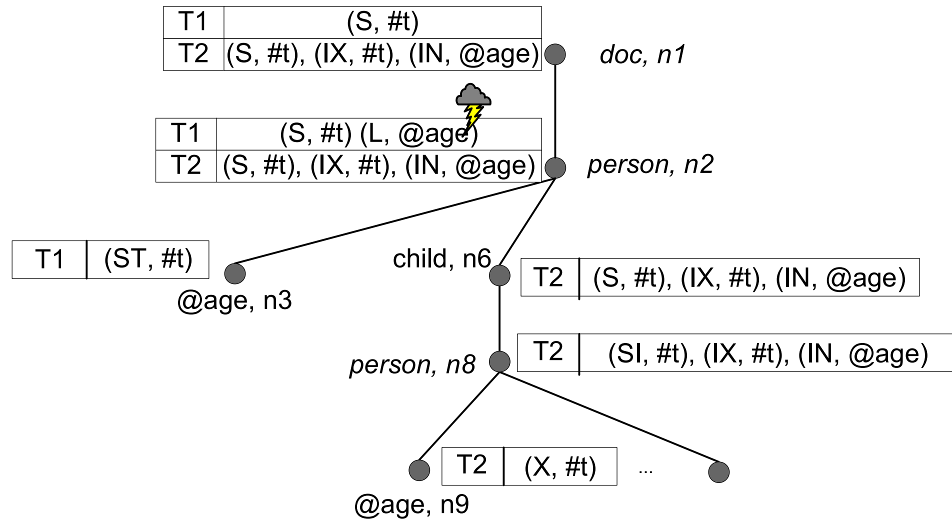


Рис. 3.5: Блокировки на схеме для примера 7

доказательство сериализуемости планов, генерируемых XDGL-планировщиком. Основными преимуществами XDGL является, то, что при определении конфликтов он учитывает иерархическую структуру XML-документов, а также семантику XML-операций. Кроме того, протокол не зависит от физических особенностей построения XML-ориентированных СУБД, что позволяет его использовать в широком классе систем.

Глава 4

Управление XML-транзакциями в реляционных СУБД

В этой главе описывается предложенный автором метод управления транзакциями в реляционных СУБД с поддержкой XML. В методе используется двухуровневая модель управления XML-транзакциями. На втором уровне для изоляции конкурентных XML-транзакций используется семантический протокол XDGL, учитывающий семантические особенности XML-модели данных. Это позволяет существенно уменьшить время отклика транзакций, а также повысить пропускную способность транзакций в РСУБД.

4.1 Многоуровневые модели транзакций и их применение для управления XML-транзакциями в РСУБД

В главе 2 говорилось, что управление XML-транзакциями можно осуществлять на основе уже существующих транзакционных механизмов в РСУБД. Действительно, прикладной программист может сгруппировать в одну единую транзакцию набор XML-запросов к базе данных, которые должны обладать свойствами атомарности и изолированности. В результате РСУБД обеспечивает транзакционные свойства для этой группы запросов.

Однако мы показали, что применение этого подхода приводит к большому количеству искусственных конфликтов между XML-транзакциями. Это связано с тем, что менеджер транзакций в РСУБД не учитывает иерархическую структуру XML-данных, а также семантические особенности XML-операций. Фактически, в случае работы нескольких параллельных транзакций на чтение и изменение одного XML-документа, планировщик выполняет их последовательно, что во многих случаях является слишком сильным ограничением. Далее в этой главе подход, при котором управление XML-транзакциями

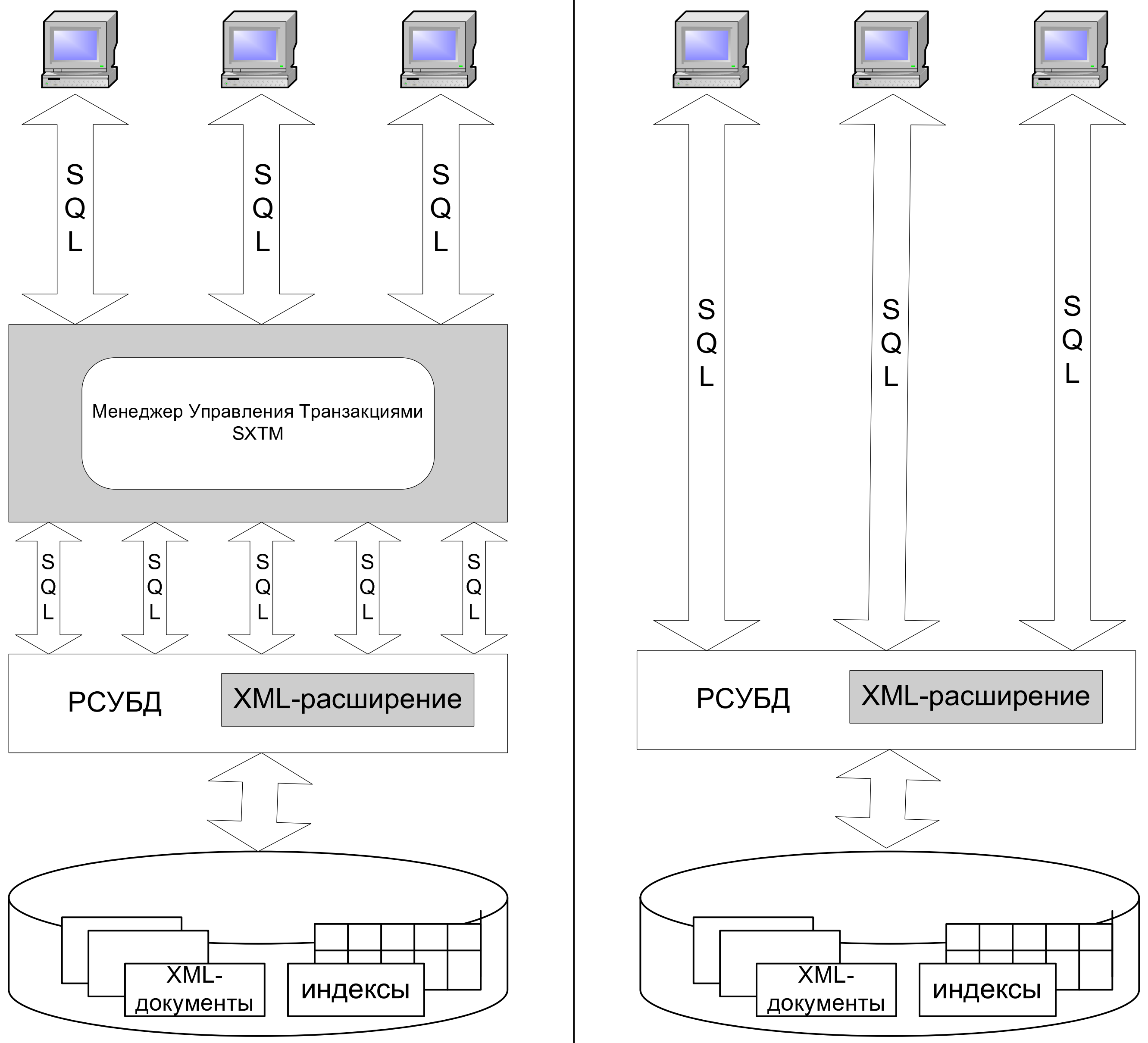


Рис. 4.1: Двухуровневая (слева) и одноуровневая (справа) модели управления XML-транзакциями в РСУБД

реализуется только за счет средств РСУБД, мы будем называть *плоской моделью транзакций* (см. рисунок 4.1 справа).

В данной работе предлагается новый способ управления XML-транзакциями в РСУБД, основанный на *двухуровневой модели транзакций* [92]. В этой модели набор запросов пользователя, для которых требуется наличие свойств атомарности и изолированности, образует глобальную транзакцию. Дополнительный менеджер транзакций, построенный над РСУБД, декомпозирует исходную глобальную транзакцию на набор субтранзакций (DB-

транзакций) к РСУБД.

При использовании этого метода мы получаем очень важное преимущество. На втором уровне менеджер XML-транзакций при принятии решения о конфликте между транзакциями может учитывать семантику XML-операций и устанавливать "семантические" блокировки до конца транзакции. В свою очередь, DB-транзакция завершается и освобождает "грубые" блокировки гораздо раньше завершения глобальной транзакции. За счет этого можно увеличить параллелизм конкурентных транзакций. Общая архитектура предложенного подхода, изображена на рисунке 4.1 (слева).

Для оценки вышеописанного подхода в контексте его применения для управления XML-транзакциями мы разработали и реализовали программный прототип SXTM (Semantic XML Transaction Manager) – дополнительный семантический менеджер управления XML-транзакциями. В SXTM глобальными транзакциями являются XML-транзакции, которые состоят из набора XML-запросов. SXTM перехватывает все запросы клиента и обращает их в субтранзакции к РСУБД. DB-транзакции могут запускаться на пониженных уровнях изоляции за счет того, что SXTM гарантирует сериализацию XML-транзакций. Эта оптимизация, в совокупности с быстрым освобождением блокировок в РСУБД, приводит к существенному повышению параллелизма XML-операций.

4.2 Применение XDGL для изоляции транзакций в РСУБД

Как мы обсуждали выше, для изоляции транзакций на втором ("семантическом") уровне необходимо применять протокол, который бы учитывал иерархическую структуру XML-данных и семантику операций. При этом протокол не должен основываться на самом XML-документе, поскольку физически в SXTM нет доступа к узлам XML-документа. Очевидно, что на эту роль подходит протокол XDGL.

Для использования протокола XDGL для изоляции XML-транзакций в РСУБД необходимо реализовать менеджер блокировок, который будет реализовывать логику XDGL-планировщика. Мы не будем в работе фокусироваться на деталях реализации менеджера блокировок, поскольку эта задача достаточно хорошо изучена и детально описана в книге Грея [10]. Здесь только отметим, что для разрешения тупиков мы используем граф ожидания транзакций [10], который периодически проверяется на наличие циклов. В случае, если цикл найден, выбирается транзакция-жертва, которая откатывается и освобождает свои блокировки. Эта процедура выполняется до тех пор, пока граф ожидания не будет содержать циклов.

Далее мы сосредоточимся на более специфичной задаче, а именно, каким образом в SXTM поддерживать описывающую схему, необходимую для работы XDGL-планировщика?

4.2.1 Поддержка описывающей схемы в SXTM

Предлагаются два способа поддержки схемы в SXTM, описываемых в двух следующих секциях.

Получение описывающей схемы на основе предписывающей схемы

В этом методе мы предполагаем, что для XML-документа существует предписывающая схема, описанная на DTD (или на любом другом языке описания схем XML-документов). В этом случае при получении DTD можно статически (один раз перед загрузкой документа в базу данных) вычислить все возможные пути, которые могут присутствовать в XML-документе. Ниже описывается алгоритм получения всех возможных путей в XML-документе на основе DTD[1].

1. Извлечь из DTD определение первого элемента. Эта строчка описывает содержание корневого элемента (набор элементов-детей и их множественность). Также извлечь определения всех атрибутов для корневого элемента. Обозначим имя корневого элемента, имена всех дочерних элементов и имена всех атрибутов корневого элемента символами e_r, $e_1, e_2, ..., e_n$ и $a_1, a_2, ..., a_m$ соответственно[2]. Инициализировать множество путей $PATH = \{(e_r, e_1), (e_r, e_2), ...(e_r, e_n), (e_r, a_1), (e_r, a_2), ..., (e_r, a_m)\}$. Далее пути мы будем обозначать символом p_i, а функцию, возвращающую последний элемент пути p_i - символом $last(p_i)$.

2. Для каждого пути $p_i \in PATH$ найти определение в DTD для $last(p_i)$. Рассмотреть следующие варианты:

   - Если $last(p_i)$ - это атрибут, то добавить в путь p_i компонент $\#text\#$.
   - Если $last(p_i)$ - это элемент, то извлечь имена всех дочерних элементов $e_{1'}, e_{2'}, ..., e_{n'}$ и атрибутов $a_{1'}, a_{2'}, ..., a_{m'}$ для элемента $last(p_i)$ из определения этого элемента в DTD. Присвоить $PATH$ новое значение: $PATH = \{(PATH \setminus p_i) \cup (p_i, e_{1'}), \cup(p_i, e_{2'}), ... \cup (p_i, e_{n'}), \cup(p_i, a_{1'}), \cup(p_i, a_{2'}), \cup(p_i, a_{m'})\}$.

---

[1] Для простоты, но без потери общности алгоритма, мы подразумеваем, что в DTD присутствуют описания только для двух типов узлов: элементов и атрибутов.

[2] PCDATA мы будем обозначать символом $\#text\#$.

3. Повторять шаг 2 до тех пор, пока ни для одного пути $p_i \in PATH$ не будут найдены дочерние узлы для элемента $last(p_i)$.

Основной недостаток приведенного алгоритма заключается в том, что он может зациклиться, если ему на вход дать *рекурсивное* DTD, когда один и тот же элемент может определяться через самого себя. Пример такого DTD изображен на рисунке 3.1. Однако в коммерческих реляционных системах, как правило, бывают ограничения на максимальную глубину иерархии XML-документа. Например, в последней версии MS SQL Server 2005 максимальная глубина иерархии XML-документа ограничена 128-ю уровнями [28]. Поэтому в описанный выше алгоритм можно добавить ограничение на максимальную длину пути и тем самым избежать зацикливания.

Однако в случае, если заранее DTD для XML-документа не известно, приведенный выше алгоритм не может использоваться. В этом случае необходимо применять второй метод.

Получение схемы на основе анализа операций изменения

В этом методе перед загрузкой исходного XML-документа в РСУБД SXTM извлекает из самого документа описывающую схему. Алгоритм получения описывающей схемы из XML-документа является достаточно хорошо изученным вопросом в сообществе исследователей баз данных, и поэтому мы здесь не приводим этот алгоритм. Можно обратиться к статье Видом [85], в которой приводится такой алгоритм.

Таким образом, после загрузки XML-документа в РСУБД через SXTM мы извлекаем схему этого документа. Однако необходимо ее поддерживать в актуальном состоянии при последующих изменениях XML-документа. При этом заметим, что XDGL протокол по-прежнему будет работать, если в схеме будут некоторые лишние пути, которые на данный момент на самом деле не присутствуют в XML-документе. Главное, чтобы таких узлов было не слишком много. Но при этом поддерживаемая схема должна отражать *все* пути, которые есть в XML-документе.

В результате мы получаем, что нам необходимо анализировать все операции op_i изменения XML-документов, которые потенциально могут расширять описывающую схему. В нашем наборе операций это $I*$ и RN.

Таким образом, при анализе этих операций необходимо определить, к каким узлам схемы относятся узлы, вставляемые этими операциями. Для этого необходимо определить множество узлов схемы, которые соответствуют целевым узлам этих операций, а затем, исходя из типа операции и имен новых узлов, определить новые узлы в схеме.

Алгоритм получения всех узлов схемы, соответствующих целевым узлам операции

изменения (напомним, что целевые узлы определяются при помощи путевого выражения LP), фактически соответствует вычислению этого выражения на схеме. Это обосновывается тем, что предписывающая схема является XML-документом. Для этих целей можно использовать любой доступный XPath-процессор (например SAXON [40]). Обозначим полученное множество символом $TARG = \{t_j\}$. По этому множеству легко получить множество всех путей $PATH = \{p_j\}$, соответствующих этим узлам.

Для получения потенциально новых путей в схеме рассмотрим следующие варианты:

- Пусть $op_i = II$. Тогда путь, соответствующий новым узлам определяется следующим образом: $p = (p_j, new_name)$ (или $(p = (p_j, new_name, \#text\#))$, если вставляется узел с текстовым значением), где new_name определяет имя нового узла (его мы извлекаем из конструктора).

- Пусть $op_i \in \{IA, IB\}$. Тогда путь, соответствующий новым узлам определяется следующим образом: $p = (p_j \setminus last(p_j), new_name)$ (или $(p = (p_j \setminus last(p_j), new_name, \#text\#))$, если вставляется узел с текстовым значением), где new_name определяет имя нового узла (его мы извлекаем из конструктора).

- Пусть $op_i = RN$. Тогда потенциально новые пути вычисляются следующим образом:

  – Если p_j соответствует листовому узлу в схеме, то $p = (p_j \setminus last(p_j), new_name)$, где new_name - это новое имя переименовываемых узлов

  – Если p_j соответствует внутреннему узлу в схеме, то новых путей может быть множество $\{(p_j \setminus last(p_j), new_name) \cup (p_j \setminus last(p_j), new_name, desc_1(p_j)) \cup ... \cup (p_j \setminus last(p_j), new_name, desc_n(p_j))\}$, где $desc_i(p_j)$ - это имя узла в схеме, который является потомком узла t_j (соответствующего пути p_j).

Очевидно, что операция удаления D может привести к тому, что в XML-документе какой-то путь будет удален, а в схеме он по-прежнему будет присутствовать. Это может привести к тому, что схема может сильно разрастись. Для предотвращения этого мы предлагаем установить некоторый порог для размера схемы, при превышении которого необходимо выполнить специальный запрос к РСУБД, который на основе существующего XML-документа и его предписывающей схемы возвращает актуальную схему, путем “отрезания” лишних путей. Ниже изображен такой запрос:

```
declare function create-actual-dg($doc as node()*, $dg as node()* ) as node()*
{
  for $n in $dg/node()
```

```
  return
    if (not (empty($doc/node()[name(.)=name($n)])))
      then
        element {name($n)} {create-actual-dg($doc/node()[name(.)=name($n)], $n)}
      else
       ()
};

create-actual-dg(document("doc-name"), document("dg-name"))
```

Здесь мы предполагаем, что перед выполнением этого запроса описывающая схема сериализуется в XML-представление на стороне SXTM и после этого загружается в РСУБД с именем `dg-name`.

Замечание 2. *Для простоты изложения, но не ограничивая общности, при определении функции* `create-actual-dg` *мы считаем, что в документе присутствуют только узлы типа элемент.*

4.3 Атомарность XML-транзакций в двухуровневой модели

Свойство атомарности транзакций означает, что все изменения, произведенные транзакцией над базой данных, либо будут полностью присутствовать в базе данных после фиксации транзакции и будут видны другим пользователям, либо ни одно изменение не будет внесено в базу данных. При декомпозиции глобальной XML-транзакции на субтранзакции SXTM необходимо гарантировать ее атомарность.

Если в РСУБД поддерживается модель вложенных транзакций ONT (Open Nested Transactions) [12], то атомарность глобальной XML-транзакции можно реализовать средствами РСУБД. Основной характеристикой модели транзакций ONT является то, что глобальная транзакция состоит из субтранзакций нижнего уровня, и при фиксации каждой субтранзакции полностью освобождаются ее блокировки. Тогда XML-транзакцию можно запускать в РСУБД как глобальную ONT-транзакцию, а отдельные XML-операции выполнять как субтранзакции. В результате при откате XML-транзакции (или ее повторном выполнении при восстановлении базы данных после сбоя) РСУБД по журналу может произвести все необходимые действия для обеспечения атомарности XML-транзакции.

Если РСУБД не поддерживает модель ONT, то SXTM разбивает исходную транзакцию на набор DB-транзакций, которые являются независимыми с точки зрения РСУБД. В этом

случае в SXTM должен присутствовать дополнительный менеджер восстановления XML-транзакций, поскольку в РСУБД ничего не известно о глобальной транзакции, и средствами РСУБД нельзя ее откатить (или повторно выполнить после сбоя). Менеджер восстановления в SXTM можно реализовать следующим образом.

SXTM должен заносить в журнал все изменения, производимые операциями модификации XML-документа. При фиксации каждой глобальной транзакции в журнал должна попадать соответствующая запись COMMIT. Журнал можно реализовать в виде дополнительной таблицы реляционной базы данных со следующей схемой:

```
SXTM-Log(LSN INT PRIMARY KEY, Transaction-ID INT,
         Operation VARCHAR(10), Parameters CLOB)
```

Первичным ключом таблицы *SXTM-Log* является столбец *LSN* (Log Sequence Number), значения которого уникально идентифицируют записи в журнале. В SXTM *LSN* можно реализовать на основе обычного счетчика, значение которого увеличивается на единицу при вставке очередной записи в журнал. По *LSN* можно восстановить порядок следования операций модификации в глобальной транзакции. Столбец *Transaction-ID* содержит идентификатор глобальной XML-транзакции, к которой относится операция. В столбце *Operation* хранится тип операции модификации. Наконец, столбец *Parameters* хранит все параметры операции изменения.

Вставка в журнал *SXTM-Log* записи об операции модификации op_i должна выполняться в рамках DB-транзакции, в которой выполняется эта операция. Таким образом, в случае мягкого сбоя системы (теряется содержимое основной памяти, но не повреждаются данные на диске) после восстановления базы данных средствами РСУБД в журнале *SXTM-Log* окажутся только записи для зафиксированных DB-транзакций. Других записей не будет, поскольку РСУБД гарантирует атомарность DB-транзакций.

Важным следствием этого свойства является то, что менеджеру восстановления SXTM никогда не требуется производить повторное выполнение (REDO) XML-операций, поскольку при выполнении операции фиксации XML-транзакции, когда производится вставка в журнал записи COMMIT для этой транзакции, происходит фиксация последней DB-транзакции. Таким образом, тот факт, что SXMT успешно зафиксировал XML-транзакцию, означает, что РСУБД успешно зафиксировала все субтранзакции, и поэтому при восстановлении после сбоя повторное выполнение всех субтранзакций будет выполнять РСУБД.

Другим важным обстоятельством является то, что для последней DB-транзакции в журнал не нужно помещать запись об XML-операции, выполняемой в этой транзакции, а достаточно вставить запись COMMIT. Это связано с тем, что SXTM никогда не откатывает последнюю DB-транзакцию (в случае сбоя эта транзакция всегда откатывается РСУБД).

Рассмотрим, какую информацию необходимо сохранять в столбце *Parameters*, чтобы можно было корректно выполнить откат (UNDO) для любой XML-операции модификации op_i.

- Пусть op_i – это операция II. Для операции вставки узла обратной является операция удаления этого узла. В соответствии с семантикой операции II новый узел вставляется на место последнего ребенка целевого узла. Следовательно, обратная операция должна удалять последнего ребенка целевого узла. Например, для операции вставки *InsertInto(<name/>, /doc/person)* в столбец *Parameters* нужно записать информацию об операции *Delete/doc/person/name[position()=last()].* При выполнении UNDO необходимо выполнить эту операцию удаления.

- Пусть op_i – это операция IB. Аналогично, в столбец *Parameters* нужно записать информацию об операции удаления вставленного узла. Например, для операции вставки *InsertBefore(<name/>, /doc/person/hobby)* в столбец *Parameters* необходимо записать информацию об операции *Delete(/doc/person/hobby::preceding-sibling[position()=1]).* Для операции IA в последнем выражении необходимо заменить ось *preceding-sibling* на *following-sibling.*

- Пусть op_i – это операция D. Для операции удаления узла обратной является операция вставки этого узла. Поэтому удаляемый узел необходимо сохранить в журнале, чтобы при откате можно было его вставить. При этом требуется запомнить не только сам удаляемый узел, но и его позицию в XML-документе. Для этого перед выполнением операции удаления нужно выполнить предварительный запрос, результатом которого будут удаляемые узлы и их позиции в XML-документе. Например, для операции *Delete(/doc/person[name='John'])* предварительный запрос будет выглядеть так:

  ```
  for $node at $i in /doc/person
  where $node/name = 'John'
  return ($node, $i)
  ```

  Этот запрос выбирает все узлы person с именем “John”, а также их позиции среди всех узлов *person.* Результат запроса сохраняется в столбце *Parameters* таблицы журнала. Кроме того, сохраняется путь в XML-документе, по которому нужно будет вставлять узлы при выполнении отката. Для рассматриваемого примера таким путем является */doc.*

  При выполнении обратной операции (UNDO) выполняются следующие действия: (1) из столбца *Parameters* выбираются путь *path*, по которому нужно вставлять узлы, а

также сами узлы $node$ и их позиции pos в XML-документе, (2) для каждого узла $node_i$ выполняется операция вставки $InsertAfter(node_i, path/node()[position() = pos_i - 1])$. Для корректного выполнения этих действий необходимо гарантировать, что позиции удаленных и повторно вставляемых узлов в XML-документе не изменятся. Иначе после повторной вставки узел может попасть не на свое место в XML-документе. Для гарантии этого вводятся две дополнительные блокировки CD и LM, которые не совместимы одна с другой и обладают следующей семантикой.

Блокировка CD (child delete) используется в операции D. Эта блокировка устанавливается на узел схемы, соответствующий родителю удаляемого узла. CD-блокировка, установленная на узел n, предотвращает какие-либо вставки или удаления детей узла n. Это гарантирует, что при откате операции D удаленные узлы будут вставляться в точности в те же позиции, в которых находились до операции удаления.

Блокировка LM (level modified) используется в операциях $I*$, D и RN. Она устанавливается на узел схемы, состав дочерних узлов которого будет изменяться. Так, если транзакция вставляет новый узел в узел n, то на узел n, помимо блокировки SI необходимо установить еще и блокировку LM. Если транзакция вставляет новый узел перед узлом n, то на узел n должна быть установлена блокировка SB, а на родителя узла n – блокировка LM. При этом блокировка LM конфликтует с блокировкой CD.

- Пусть op_i – это операция RN. При выполнении этой операции происходит как удаление, так и вставка узла. Поэтому в журнал необходимо заносить информацию для повторной вставки переименовываемого узла и для удаления нового узла.

В заключение раздела отметим, что при использовании описанного подхода к реализации в SXTM дополнительного менеджера восстановления XML-транзакций (если в РСУБД не поддерживается модель транзакций ONT) для операций D требуется выполнять дополнительный XQuery-запрос, а также приходится устанавливать дополнительные блокировки. Этих накладных расходов можно избежать, если использовать отложенную стратегию выполнения D операций.

Основная идея заключается в том, что SXTM может отложить реальное выполнение операции удаления до тех пор, пока в этой же транзакции не понадобится выполнить какую-либо операцию, зависящую от результатов операции удаления, либо не произойдет фиксация транзакции. Если операция удаления выполняется в конце транзакции (при выполнении последней DB-транзакции), то в журнал уже не нужно писать какую-либо информацию об этой операции, поскольку UNDO и REDO для последней DB-транзакции всегда выполняет РСУБД.

При отложенном выполнении операции D SXTM должен только установить обычные XDGL-блокировки для операции D в момент ее появления в транзакции.

4.4 Индивидуальные откаты транзакций и восстановление базы данных после сбоев

В этом разделе мы обсуждаем восстановление базы данных после сбоев при использовании многоуровневой модели управления XML-транзакциями. Но начнем мы с более простого случая - индивидуального отката транзакции.

Предположим, что транзакцию с идентификатором *tid* необходимо откатить. В этом случае необходимо последовательно в обратном порядке выполнить обратные операции для выполненных XML-операций. Для этого необходимо воспользоваться информацией из таблицы *SXTM-log*. Для получения обратного итератора (т.е при первом вызове метода *next()* возвращается последняя запись транзакции, далее возвращается предпоследняя запись и т.д. до первой записи) по записям таблицы, относящимся к рассматриваемой транзакции, необходимо выполнить следующий SQL-запрос:

```
SELECT  Operation, Parameters
FROM SXTM-log
WHERE Transaction-ID = trid
ORDER BY LSN DESC;
```

При этом откат каждой XML-операции выполняется в отдельной DB-транзакции, и в этой же транзакции в таблицу журнала *SXTM-log* заносится *компенсационная запись*. Компенсационная запись имеет тип *CLR* (Compensation Log Record) и в качестве параметров содержит LSN записи, которую следующей необходимо откатывать. Так, если при прямом выполнении некоторой транзакции в журнал были занесены записями со значениями LSN равным 1, 2 и 3, то компенсационная запись для записи 3 (обозначим ее 3’) будет содержать указатель на предыдущую запись 2, в свою очередь компенсационная запись 2’ для записи 2 будет содержать указатель на запись 1 и т.д. В DB-транзакции, выполняющей откат последней операции, необходимо в таблицу занести также запись ROLLBACK для этой транзакции. Таким образом, при завершении отката транзакции с тремя прямыми записями в журнале должны появиться записи 1, 2, 3, 3’, 2’, 1’, ROLLBACK. Причем, если необходимо откатить компенсационную запись (например во время восстановления после сбоя), то все, что нужно сделать, - это извлечь из атрибута *Parameters* значение LSN записи, которую следующей необходимо откатывать. После этого нужно вызывать метод *next()* обратного итератора до тех пор, пока не встретится запись с этим LSN.

Отметим, что без использования компенсационных записей может возникнуть ситуация, когда одна XML-операция будет откачена дважды - первый раз во время отката, а затем во время восстановления после сбоя (если сбой случился во время отката).

Замечание 3. *Отметим, что указанный выше запрос выполняется в РСУБД с уровнем изоляции COMMITTED READ, поскольку гарантируется, что параллельно не будут вставляться записи с таким же идентификатором транзакции. Это позволяет избежать возможной блокировки конкурентных транзакций при попытке вставить новую запись в таблицу* SXTM-log.

Далее рассмотрим шаги, которые необходимо совершить для восстановления базы данных после сбоя.

1. Запустить РСУБД и дождаться окончания ее восстановления после сбоя. Заметим, что после этого может оказаться, что набор XML-транзакций, находятся в промежуточном состоянии, т.е. для них нет ни записи COMMIT, ни записи ROLLBACK. Именно для таких транзакций затем необходимо выполнить операцию UNDO.

2. Выполнить следующий запрос, который выбирает все идентификаторы транзакций, которые не были зафиксированы или полностью откачены:

```
SELECT DISTINCT a.Transaction-ID
FROM SXTM-log AS a
WHERE not
       (exists(SELECT b.Transaction-ID
               FROM SXTM-log AS b
                WHERE a.Transaction-ID = b.Transaction-ID AND
                     (b.Operation = 'COMMIT' OR b.Operation = 'ROLLBACK')
            ))
```

   Предположим, что результатом этого запроса будет последовательность идентификаторов $(trid1, trid2, ...tridn)$

3. Выполнить следующий запрос:

```
SELECT  Operation, Parameters
FROM SXTM-log
WHERE Transaction-ID = trid1 OR
```

```
        Transaction-ID = trid1 ... Transaction-ID = tridn
    ORDER BY LSN DESC;
```

И затем, при помощи метода *next()* обратного итератора, полученного в результате выполнения вышеуказанного запроса, последовательно выполнять откат требуемых транзакций. И в этом случае в журнал заносятся компенсационные записи на случай повторного сбоя. Причем гарантируется, что для каждой транзакции в журнале окажется ровно столько компенсационных записей, сколько существует записей, выполненных при прямом выполнении транзакции.

Размер журнала может значительно вырасти через некоторое время. Поэтому мы предусматриваем процедуру отсечения журнала (*truncate log*), которую можно выполнять периодически. Эта процедура может запускаться также на низких уровнях изоляции (например COMMITTED READ). Очевидно, что для корректного восстановления транзакций достаточно присутствие записей только для незавершившихся транзакций. Поэтому необходимо в SXTM поддерживать переменную $MinLSN$, которая содержит значение минимального LSN записи, относящейся к незавершившимся транзакциям. Для этого необходимо в SXTM поддерживать таблицу[3] всех активных транзакций, с указанием LSN первой записи в журнале для каждой транзакции (или NULL, если транзакция не занесла в журнал ни одной записи). Очевидно, что при завершении какой-либо транзакции необходимо заново сканировать эту таблицу и определять значение $MinLSN$. Все записи в журнале с LSN меньшим, чем $MimLSN$, могут удаляться процедурой *truncate-log*. Для этого достаточно выполнить следующий оператор SQL:

```
DELETE FROM SXTM-log
WHERE LSN < MinLSN;
```

4.5 Повышение параллелизма внутри XML-транзакций

За счет применения семантического протокола изоляции XML-транзакций удается избежать искусственных конфликтов между транзакциями, что приводит к повышению параллелизма конкурентных XML-транзакций. Однако можно уменьшить время выполнения XML-транзакций также за счет использования параллелизма *внутри* транзакции (intra-transaction parallelism) [49].

[3]Заметим, что эта таблица не должна хранится в РСУБД, а достаточно ее поддерживать в актуальном состоянии в оперативной памяти SXTM.

В применении к XML параллелизм внутри транзакций может использоваться в следующих случаях. Очень часто XML-приложению необходимо хранить много небольших XML-документов, удовлетворяющих одной схеме. Как правило, множество таких схожих XML-документов называют *коллекцией* XML-документов. В РСУБД для хранения такой коллекции удобнее всего использовать таблицу (назовем ее `XMLTable`), и один из атрибутов будет обладать типом `XML`. В этом атрибуте будет храниться XML-документ. Другие атрибуты таблицы могут использоваться для описания свойств XML-документа (например уникальный идентификатор документа). Таким образом, каждому отдельному XML-документу в коллекции соответствует отдельный кортеж в реляционной таблице. Схема реляционной таблицы представляет собой следующее:

```
XMLTable(DocId INT PRIMARY KEY, ..., XMLType XCol)
```

Пример запроса к таблице `XMLTable` выглядит следующим образом: *SELECT XCol.query('Здесь записывается произвольный XQuery запрос') FROM XMLTable";.* Результатом этого запроса будет набор частей XML-документов, удовлетворяющих XQuery запросу. Этот XQuery запрос выполняется для каждого кортежа в таблице `XMLTable`. Для выборки идентификаторов документов, удовлетворяющих некоторому XQuery запросу, можно использовать следующее выражение: *SELECT DocId FROM XMLTable WHERE XCol.exist ('условие на XQuery') = 1.* Наконец, для модификации набора XML-документов можно использовать выражение: *UPDATE XMLTable SET XCol.modify('delete XPath выражение')*[4].

Напомним, что, как правило, для ускорения выборки частей XML-документов применяются индексы. Детальное описание индексов, специфичных для XML можно найти в работе Пола [34]. Например, применение индекса по пути и значениям (path-value index) позволяют очень эффективно выполнять запросы следующего типа: */book/section[title="Web"].*

Далее мы приводим шаги, которые описывают, как можно выполнять XPath выражения с использованием внутреннего параллелизма.

1. *Синтаксический разбор запроса.* На этой стадии строится синтаксическое дерево исходного запроса и извлекаются предикаты (например *title="Web"* из предыдущего примера).

2. *Выделение подзапроса.* Этот шаг определяет подзапрос SQ (subsuming query) исходного XPath-запроса, который обладает двумя свойствами: (1) этот запрос эффективно

[4]Приведенный синтаксис выражений был взят из СУБД MS SQL Server 2005 [28]

выполняется на основе индексов (см. [34]), и (2) он выбирает набор кортежей, которые содержат все кортежи исходного запроса. Вторая проблема известна как *containment query* и подробно изучается в [50]. На этом шаге используются предикаты, полученные на первом шаге, для которых создан индекс.

3. *Вычисление промежуточного результата.* SXTM посылает выделенный подзапрос SQ в РСУБД и получает множество идентификаторов *документов-кандидатов*, в которых содержится результат исходного запроса

4. *Вычисление фильтрующих запросов.* На этом шаге для каждого документа, попавшего в результат, полученного на предыдущем шаге, выполняется исходный запрос. Причем эти запросы запускаются SXTM *параллельно.*

5. *Формирование результата.* На этом шаге SXTM формирует окончательный результат путем "склеивания" результатов от запушенных запросов к РСУБД.

Схематичное изображение описанного алгоритма для запроса *SELECT XCol.query('//book/section[title='Web']/chapter') FROM XMLTable* изображено на рисунке 4.2. Предполагается, что на элементах *title* существует индекс.

Заметим, что приведенный алгоритм аналогичным образом работает и для запросов на изменение.

За счет того, что подзапросы к РСУБД выполняются параллельно, общее время их выполнения может существенно сокращаться при использовании многопроцессорных систем. Кроме того, каждая транзакция к отдельному документу блокирует один кортеж, в котором хранится XML-документ[5], но не блокирует всю таблицу `XMLTable`, что увеличивает параллелизм конкурентных DB-транзакций. Причем за счет "измельчения" транзакции блокировка на отдельный XML-документ в РСУБД устанавливается на совсем небольшое время.

Выполнение SQ-запроса предполагается выполнять на основе индексов, для которых в РСУБД реализованы эффективные методы изоляции (такие техники были разработаны в работах Мохана [46, 47]). Поэтому выполнение SQ не приводит к созданию искусственных конфликтов между транзакциями.

[5]Здесь мы предполагаем, что в РСУБД используются блокировки на уровне кортежей.

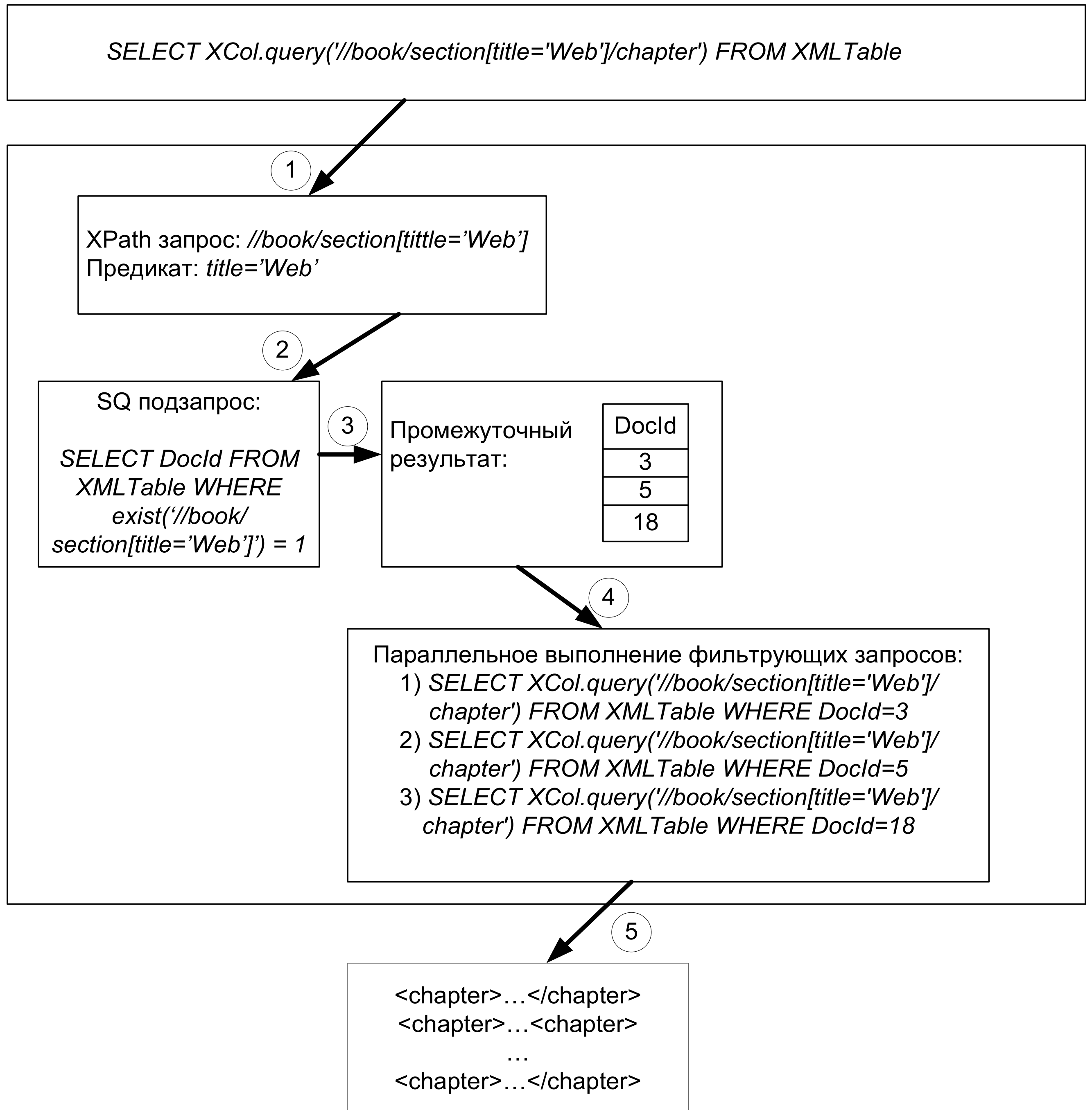


Рис. 4.2: Обработка XML-запроса с распараллеливанием

4.6 Экспериментальная оценка семантического менеджера управления XML-транзакциями

Для экспериментальной оценки семантического менеджера управления XML-транзакциями был реализовал программный прототип на языке C++ в виде расширения к реляционной СУБД MS SQL Server 2005. При помощи экспериментов была измерена производительность РСУБД для управления XML-данными с использованием SXTM и без него.

Ниже мы будем рассматривать результаты экспериментов, направленные на измерение следующих характеристик РСУБД:

- Накладные расходы, накладываемые SXTM.
- Пропускная способность РСУБД при наличии параллельных потоков транзакций на чтение и изменение XML-документов.
- Среднее время отклика для транзакций при наличии конкурентных транзакций на чтение и изменение XML-документов.
- Среднее время отклика для транзакций с использованием параллелизма внутри транзакций при наличии конкурентных транзакций на чтение и изменение XML-документов в коллекции.

4.6.1 Экспериментальная установка

В этом разделе мы рассмотрим инфраструктуру, на основе которой проводились эксперименты. Рассматриваемая инфраструктура базируется на тестах измерения производительности, предложенных в проекте XMark [94]. В рамках этого проекта был предложен генератор XML-документов xml-gen, позволяющий создавать XML-документы произвольного размера, а также набор тестовых запросов на выборку XML-документов. Генерируемые XML-документы описывают классическую систему аукционов, где описываются текущие и завершившиеся аукционы, данные о потенциальных покупателях и клиентах, совершивших покупку, описание продаваемых предметах и.т.д. Поскольку в тестовом наборе XMark нет запросов на изменение XML-документов, мы разработали набор таких запросов, которые отражают основные операции работы с аукционами: операции преобразования открытого аукциона в закрытый, добавления на аукцион нового участника, изменения базовой цены продаваемого предмета, изменения способа доставки предмета и т.д. Описывающая схема XML-документов XMark изображена на рисунке 4.3[6]. Автор проводил

[6] С целью небольшого упрощения мы опустили ряд узлов схемы. Кроме того, с целью получения более компактного рисунка мы все регионы изобразили в виде одного дерева.

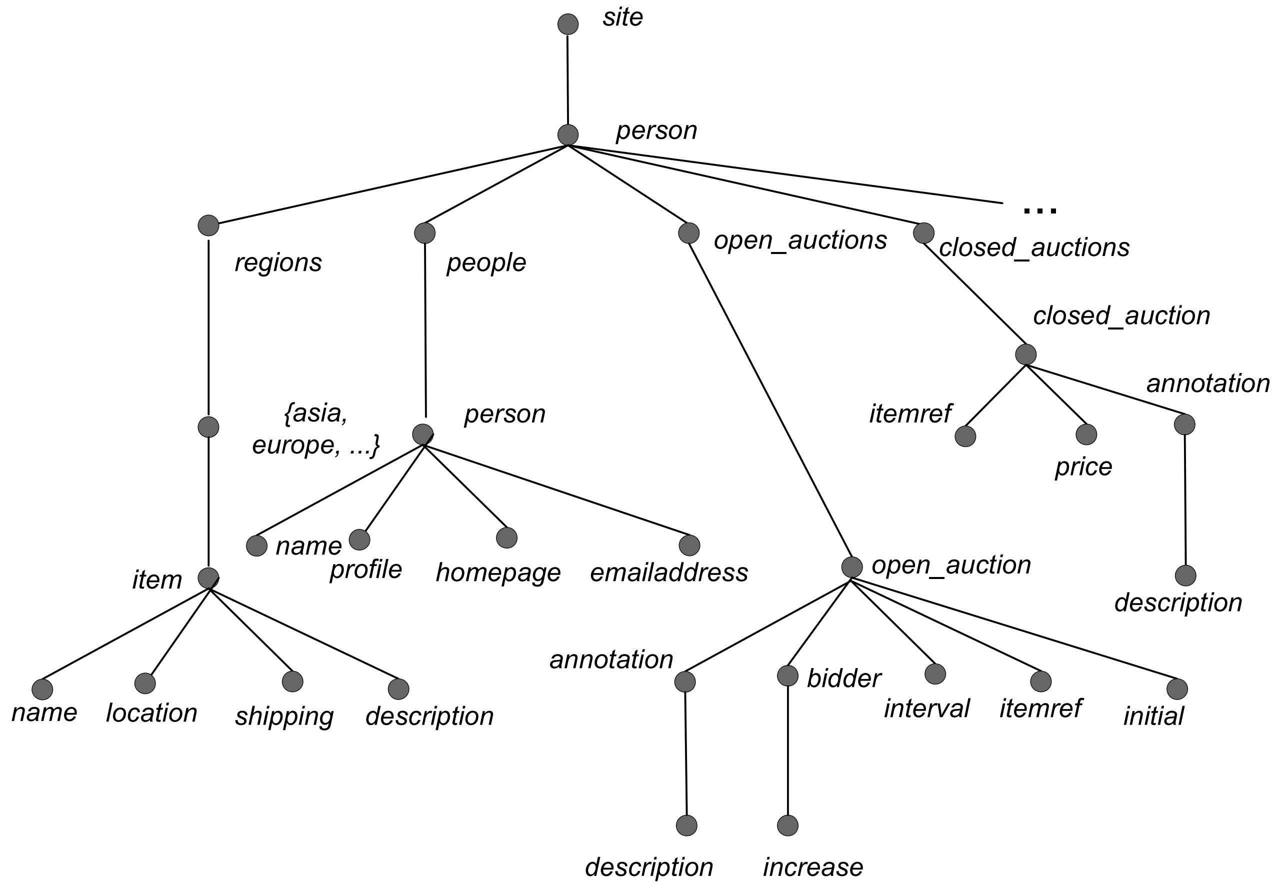


Рис. 4.3: Описывающая схема XML-документов XMark

эксперименты для документов различного размера от 10Мб до 1Гб.

Эксперименты проводились над РСУБД MS SQL Server 2005, установленный на компьютере Intel Pentium(R) 4 CPU 2.80ГГц, 512Мб ОЗУ с ОС MS Windows XP.

4.6.2 Эксперимент 1: накладные расходы

Первая группа наших экспериментов направлена на выявление накладных расходов, накладываемых SXTM. Для этого мы запускали последовательно два потока транзакций. Первый поток состоит только из транзакций на чтение, а второй поток содержит как транзакции на чтение (25%), так и транзакции на изменение. В каждый поток входит по 100 транзакций. Каждая транзакция состоит из различного количества запросов в пределах от 1 до 5 штук. Результаты измерений изображены на рисунке 4.4.

На рисунке 4.4 изображены результаты замеров для XML-документов размером 1Мб, 10Мб, 50Мб и 100Мб. Левая шкала каждой группы изображает время работы потока запросов на чтение при использовании плоской модели транзакций. За ней следует шкала, обозначающая, время работы потока запросов на чтение с использованием

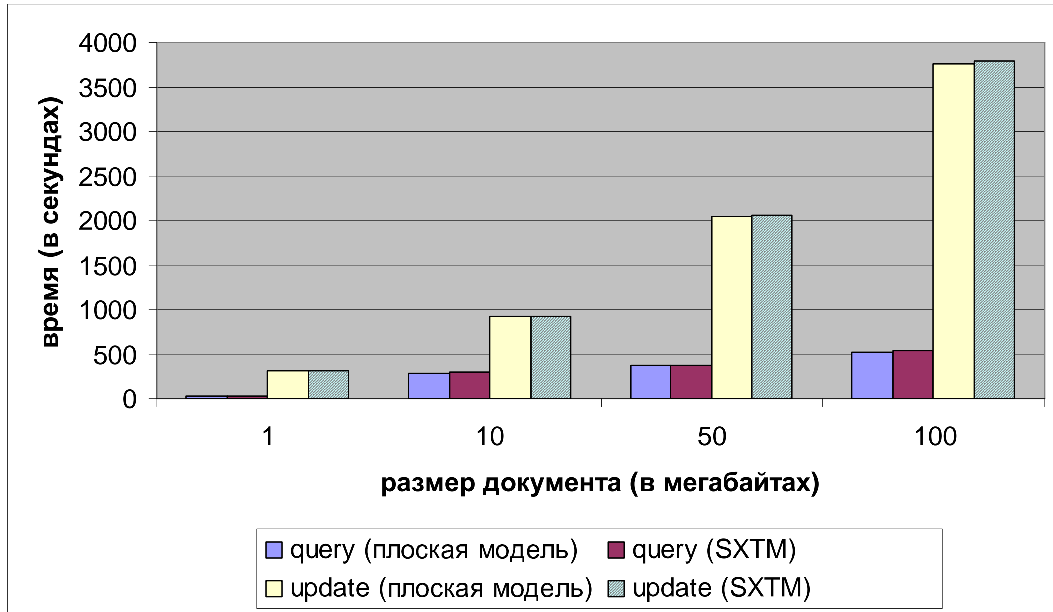


Рис. 4.4: Измерение накладных расходов SXTM

двухуровневой модели транзакций SXTM. Третья и четвертая шкалы каждой группы аналогичны предыдущим двум, но вместо потоков транзакций на чтение запускаются потоки транзакций на изменение.

Из рисунка 4.4 следует, что накладные расходы SXTM не существенны по сравнению с общим временем выполнения потоков.

Заметим, что для читающих транзакций дополнительные издержки практически не растут с ростом размера документа. Это связано с тем, что количество блокировок не растет с ростом размера документа (описывающая схема не изменяется), и операции на чтение не нужно заносить в журнал. Однако небольшой прирост дают издержки, связанные с фиксацией DB-транзакций.

Дополнительные издержки для потоков транзакций на изменение несколько больше. Это связано с тем, что необходимо заносить дополнительные записи в таблицу *SXTM-log*. Причем с ростом документа может расти и размер записей, которые необходимо заносить в журнал. С этим связано то, что с ростом размера документа издержки на журнализацию также возрастают. Прежде всего, это относится к операции D, для которой необходимо заносить в журнал удаляемые узлы. Однако общее время работы транзакции увеличивается примерно на несколько процентов.

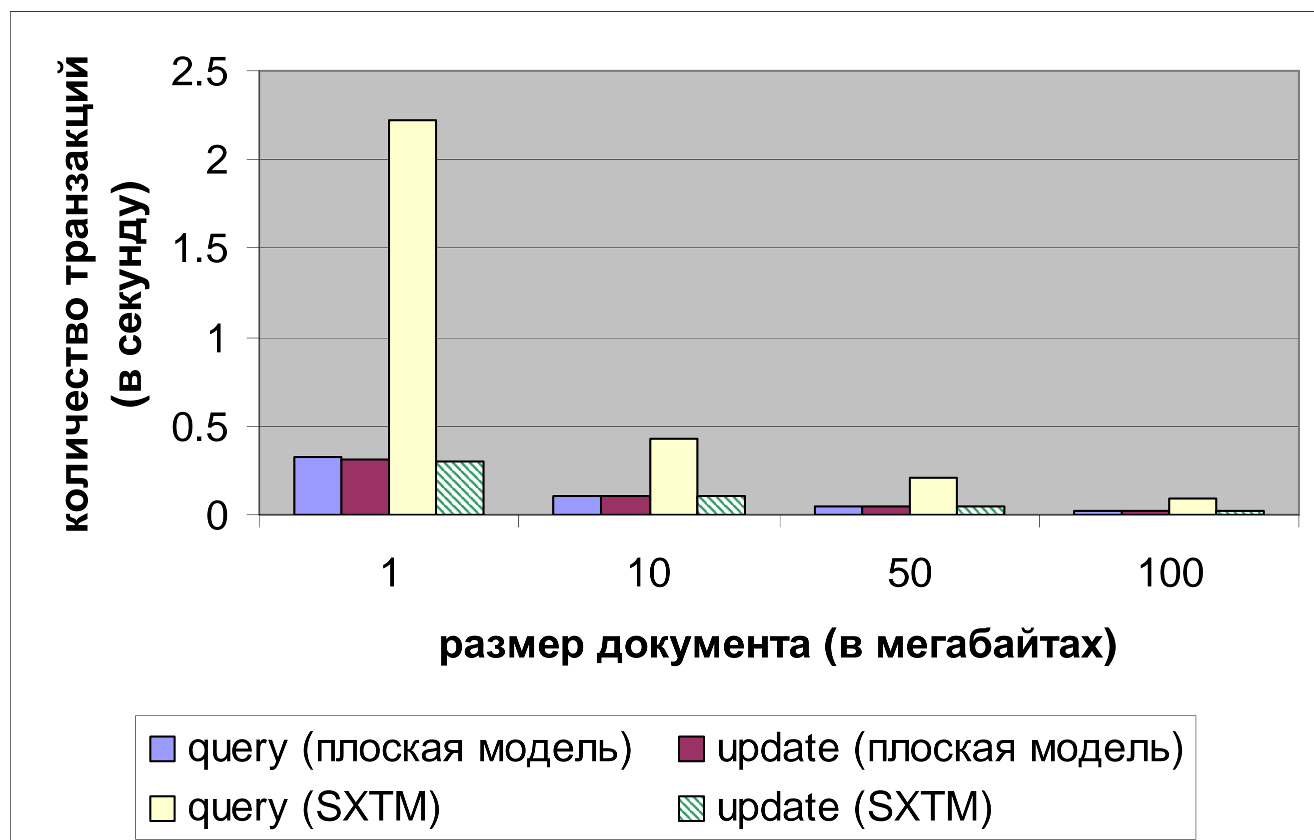


Рис. 4.5: Измерение пропукной способности РСУБД

4.6.3 Эксперимент 2: пропускная способность

Второй эксперимент направлен на измерение пропускной способности РСУБД при наличии конкурентных потоков транзакций на чтение и изменение XML-документов. Используемые потоки транзакций мы описывали в ходе обсуждения предыдущего эксперимента. Однако отличие заключается в том, что потоки запускали параллельно.

На рисунке 4.5 изображены полученные результаты измерения для XML-документов размером 1Мб, 10Мб, 50Мб и 100Мб. Левая шкала каждой группы изображает среднее количество выполненных транзакций в секунду для потока запросов на чтение при использовании плоской модели транзакций. За ней следует шкала, обозначающая, среднее количество выполненных транзакций на изменение за одну секунду с использованием плоской модели. Третья (четвертая) шкала изображает среднее количество выполненных транзакций на чтение (изменение) в секунду при использовании двухуровневой модели транзакций SXTM.

На диаграмме видно, что SXTM позволяет существенно увеличить пропускную способность РСУБД. Наибольший прирост производительности получается для читающих транзакций. Это связано с тем, что в MS SQL Server 2005 существуют специальные путевые

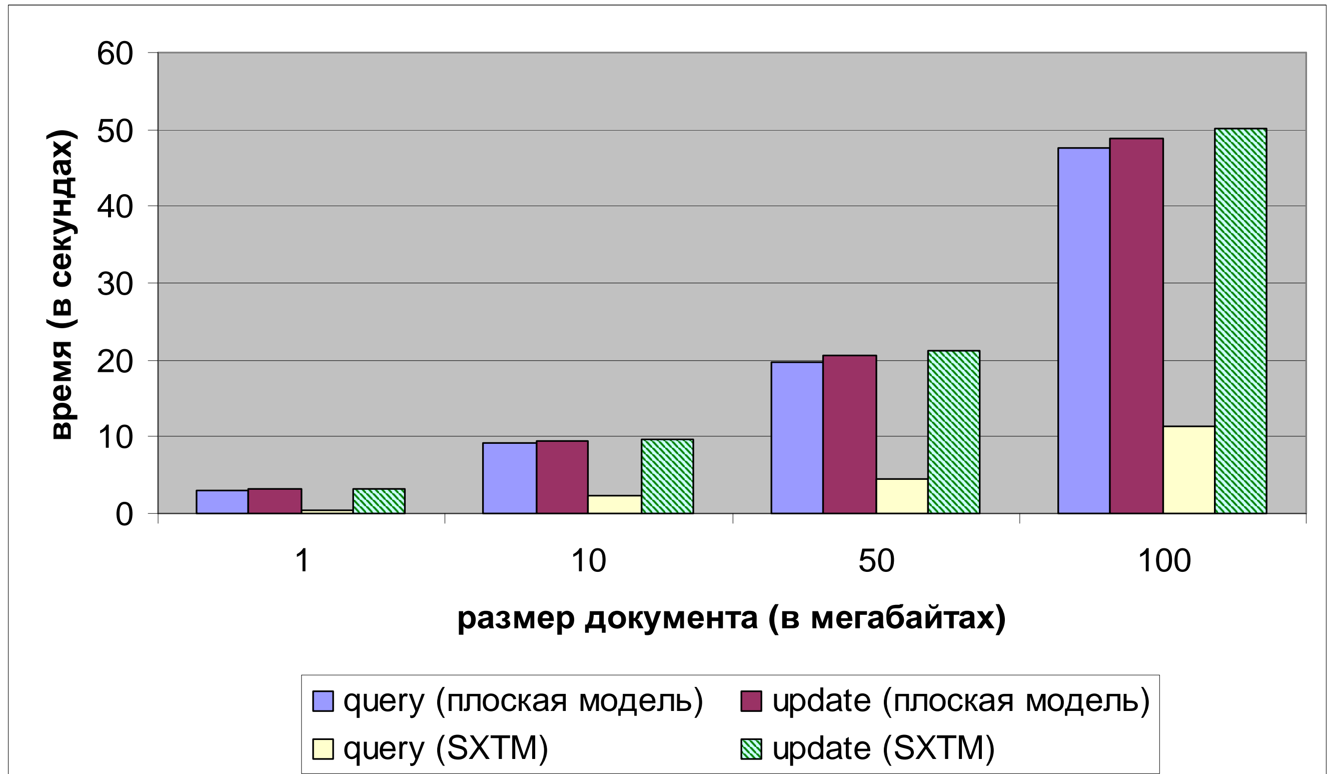


Рис. 4.6: Измерение времени отклика РСУБД

индексы и индексы по значениям, применение которых позволяет выполнять запросы эффективно. В результате, если выполнять только читающие транзакции, то они выполнятся очень быстро. С другой стороны, при использовании плоской модели транзакций читающие транзакции большую часть времени ждут освобождения монопольной блокировки всего XML-документа, которую устанавливает изменяющая транзакция. Фактически, в плоской модели читающие и изменяющие транзакции выполняются последовательно через одну: после выполнения пишущей транзакции выполняется читающая транзакция, затем снова выполняется пишущая транзакция и т.д. В SXTM же за счет применения семантических блокировок XDGL читающие транзакции конфликтуют с пишущими транзакциями гораздо реже, что и приводит к существенному увеличению срднего количества выполненных транзакций на чтение в единицу времени.

4.6.4 Эксперимент 3: время отклика

Третий эксперимент направлен на измерение среднего время отклика для транзакций при наличии конкурентных транзакций на чтение и изменение. Используемые потоки транзакций мы описывали в ходе обсуждения первого эксперимента. Однако отличие заключается в том, что потоки запускались параллельно.

На рисунке 4.6 изображено среднее время отклика для транзакций из вышеописанного эксперимента. Обозначения шкал аналогичны предыдущему эксперименту. На диаграмме видно, что время отклика значительно уменьшается для запросов на чтение. Это происходит, прежде всего, за счет использования протокола XDGL. Время отклика запросов на изменение практически не меняется с использованием SXTM. Это связано не только с тем, что SXTM накладывает дополнительные издержки для таких запросов, но и с тем, что в MS SQL Server 2005 запросы на чтение выполняются гораздо быстрее, чем запросы на изменение. И ожидания изменяющих транзакций даже при использовании плоской модели увеличиваются не очень значительно. Однако автор предполагает, что для будущих версий РСУБД, в которых будет реализована более эффективная поддержка запросов на изменение, использование SXTM будет уменьшать время отклика и для изменяющих транзакций. Однако дополнительная оптимизация по распараллеливанию запросов внутри транзакций для коллекций XML-документов позволяют уменьшить время отклика для изменяющих транзакций. Это демонстрирует следующий эксперимент.

4.6.5 Эксперимент 4: время отклика транзакций при использовании параллелизма внутри транзакций

Наш четвертый эксперимент проводился для коллекции XML-документов, над которыми снова параллельно запускалось два потока транзакций на чтение и изменение XML-документов в коллекции. Коллекцию XML-документов мы получили путем деления исходного XML-документа XMark на части. Причем мы проводили эксперименты для коллекций с размером документов 1Мб (100 документов), 100Кб (1000 документов) и 10Кб (10000 документов).

На рисунке 4.7 изображено среднее время отклика транзакций для описанного эксперимента. Основной вывод, который мы сделали, заключается в том, что чем меньше гранулярность XML-документов в коллекции, тем меньше время отклика транзакций. Действительно, параллелизм увеличивается как за счет того, что исходный запрос разбивается на много “маленьких” DB-транзакций, которые затем запускаются параллельно, так и за счет того, что “грубые” блокировки снимаются с документов в коллекции очень быстро. Мы видим, что время отклика для изменяющих транзакций также уменьшилось. Это происходит, в частности, и из-за того, что в MS SQL Server 2005 проведение операций изменения в нескольких маленьких документах-кандидатах происходит заметно быстрее, чем в одном большом XML-документе. В плоской модели транзакций время отклика также уменьшается. Это происходит из-за того, что документы-кандидаты имеют маленький размер, и операции изменения над ними происходят достаточно эффективно.

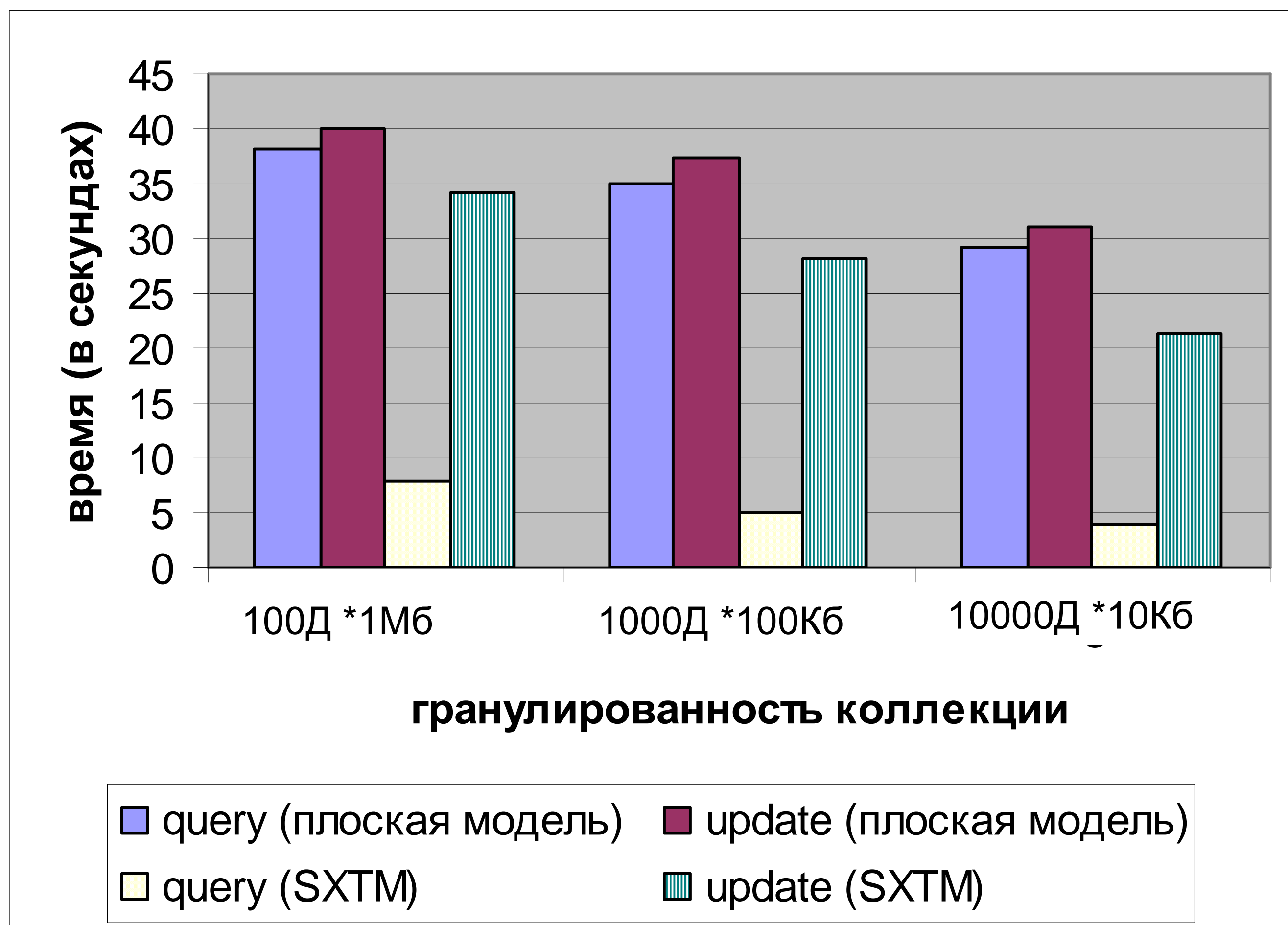


Рис. 4.7: Измерение времени отклика РСУБД для коллекции XML-документов

4.7 Выводы

В настоящей главе описан разработанный автором метод управления XML-транзакциями в РСУБД, который основан на применении двухуровневой модели транзакций. Основным преимуществом этого метода является то, что семантический менеджер управления XML-транзакциями значительно сокращает количество искусственных конфликтов между транзакциями за счет использования специального протокола XDGL. Метод может использоваться для любых РСУБД с поддержкой XML, поскольку он не зависит от физических особенностей поддержки XML в РСУБД. Метод позволяет существенно повысить производительность РСУБД при наличии конкурентных транзакций к XML-документам, что подтверждается проведенными экспериментами.

Глава 5

Управление транзакциями в прирожденных XML-СУБД

В этой главе описывается предложенный автором метод управления транзакциями в полнофункциональных прирожденных XML-СУБД. Метод базируется на понятии снимка базы данных. На основе снимков базы данных реализуется как изоляция параллельных транзакций, так и восстановление базы данных после сбоев в системе.

5.1 Требования к управлению транзакциями в прирожденных XML-СУБД

В главе 2 мы обсуждали неразвитость методов управления параллельными транзакциями в прирожденных XML-СУБД. Мы также отмечали, что в XML-СУБД подсистема управления транзакциями должны разрабатываться с нуля, в отличии от РСУБД, для которых, как было продемонстрировано в главе 4, эффективное управление XML-транзакциями может существенным образом основываться на существующих в РСУБД базовых механизмах управления транзакциями. Поэтому прежде чем переходить к детальному описанию методов управления транзакциями в прирожденных XML-СУБД, мы определим специфичные требования к этим методам, накладываемые приложениями XML-СУБД, а также методами поддержки хранимых XML-данных. В этом разделе автор также предлагает общие подходы и методы для выполнения этих требований.

1. *Читающие транзакции не должны конфликтовать с изменяющими транзакциями.*

   Как уже обсуждалось в главе 2, для многих приложений XML-СУБД характерна ситуация, когда необходимо выполнять в параллельном режиме достаточно “длинные”

транзакции на чтение и "короткие" транзакции на изменение данных. Необходимо, чтобы эти транзакции могли выполняться без конфликтов друг с другом. Иными словами, присутствие читающих транзакций не должно приводить к приостановке или даже откату изменяющих транзакций.

Очевидно, что применение стандартного двухфазного протокола сериализации транзакций, реализованного во многих коммерческих СУБД, приводит к длительным ожиданиям изменяющих транзакций, что во многих ситуациях недопустимо. Это связано с тем, что читающие транзакции блокируют на длительное время (до конца транзакции) большое количество элементов базы данных, и в результате попытка изменить эти данные со стороны другой транзакции приводит к ее блокировке. Поэтому двухфазный протокол не может удовлетворить указанное требование.

Для решения этой проблемы существует расширение двухфазного протокола механизмом версионности данных. Первой системой, в которой был реализован версионный двухфазный протокол, является СУБД компании Prime Computer Corporation [63]. Также версионный двухфазный протокол был реализован в реляционной системе DEC Rdb [65]. Наконец, версионные протоколы сериализации транзакций применяются в Oracle [30].

Широко используемой разновидностью многоверсионного протокола является протокол ROMV, в котором читающие транзакции не устанавливают блокировки на прочитываемые данные. При этом протокол гарантирует, что читающие транзакции всегда получают консистентную версию данных. Это достигается за счет чтения не последней версии данных, а некоторой старой зафиксированной версии, временная метка которой меньше, чем временная метка читающей транзакции. Изменяющие транзакции для синхронизации друг с другом используют двухфазный протокол. Также поддерживается процедура удаления старых версий данных, которые уже не нужны читающим транзакциям. Такая форма версионности данных, когда старые версии элементов данных хранятся сравнительно небольшое время с целью разрешения конфликтов между читающими и изменяющими транзакциями называется *кратковременной* версионностью (transient versioning) [70].

2. *Метод изоляции изменяющих транзакций не должен приводить к большому количеству псевдоконфликтов между этими транзакциями. Кроме того, накладные расходы, связанные с изоляцией изменяющих транзакций, должны быть невелики.*

   В предыдущем пункте мы обозначили общий подход к решению проблемы безконфликтного выполнения читающих и изменяющих транзакций - многоверсионные

протоколы (в частности ROMV). Протокол ROMV позволяет выполнять читающие транзакции без конфликтов с изменяющими транзакциями, но синхронизация между изменяющими транзакциями по-прежнему осуществляется на основе двухфазного протокола. Следовательно между изменяющими транзакциями могут возникать конфликты (по блокировкам). Поэтому блокировочный механизм для изменяющих транзакций должен быть таким, чтобы его применение не приводило к большому количеству псевдоконфликтов между транзакциями. Для достижения этой цели блокировочный механизм должен учитывать семантические и структурные особенности XML модели данных.

В главе 3 автором был предложен такой блокировочный механизм, и поэтому, фактически, для обеспечения указаного требования необходимо совместить версионный механизм с протоколом XDGL.

3. *Версионный механизм не должен существенно влиять на эффективность операции переходов по указателям.*

Мы уже отмечали в разделе 1.3.2, что в XML-СУБД очень важно эффективно поддерживать операцию перехода по указателю от одного узла к другому. В [61] был предложен метод управления памятью для хранимых XML-данных, в котором переход по указателю реализован очень эффективно. Основная идея метода заключается в создании слоистого адресного пространства (САП) базы данных, которое позволяет использовать одно и то же представление данных как во внешней, так и в оперативной памяти. Это и позволяет реализовать операцию перехода по указателю очень эффективно, поскольку нет накладных расходов, связанных с трансляцией адресов. Фактически, САП реализует виртуальную память СУБД на основе виртуальной памяти операционной системы. Однако применение версионного механизма совместно с САП может приводить к потере эффективности операции перехода по указателям. Рассмотрим подробнее эту проблему.

Существуют две различные стратегии поддержки версионности данных, предложенные в работах Чана [63] и Кэри [66]. Сразу отметим, что обе стратегии были предложены в контексте реляционных баз данных.

В [63] используется версионность на уровне *блоков* (см. рисунок 5.1). Это предполагает, что перед проведением операции изменения последняя версия блока копируется в другой блок (все старые версии блоков хранятся в отдельном сегменте), затем происходит операция изменения в блоке, и, наконец, из измененной версии блока устанавливается указатель на старую версию блока. Таким образом, все версии блока

Рис. 5.1: Физическая организация БД с версионностью на уровне страниц

связаны в однонаправленный обратный список. Если читающей транзакции необходима некоторая старая версия блока, то она должна последовательно прочитать все более новые версии этого блока, переходя к более старой версии по указателю.

В [66] используется версионность на уровне *записей* (см. рисунок 5.2). Это подразумевает, что при изменении одной записи в блоке создается копия только этой записи. Причем, по возможности, старые версии записей хранятся в том же блоке, что и последняя версия. Для этого в каждом блоке резервируется некоторое место для хранения старых версий записей. Однако при заполнении этого пространства старые версии записи все равно копируются в отдельный блок.

Рассмотрим реализацию версионности для XML-данных на уровне записей. При изменении дескриптора некоторого узла n_1 транзакцией T_1 его старая версия должна быть помещена на новое место (возможно даже в другом блоке). Предположим, что на этот дескриптор ссылается некоторый другой узел n_2, расположенный на один уровень выше по иерархии XML-документа. В этом случае, если параллельно запущена читающая транзакция T_2, и она переходит от дескриптора узла n_2 к узлу n_1, то ей нужно прочитать старую версию этого дескриптора. Таким образом, ей необходимо сделать еще один переход по указателю на старую версию записи. В результате теряется возможность *прямых* переходов от одного узла к другому, а вместе с этим теряется и эффективность операции перехода по указателю. Поскольку при выполнении типичных запросов на языках XPath/XQuery требуется, как правило, огромное число переходов

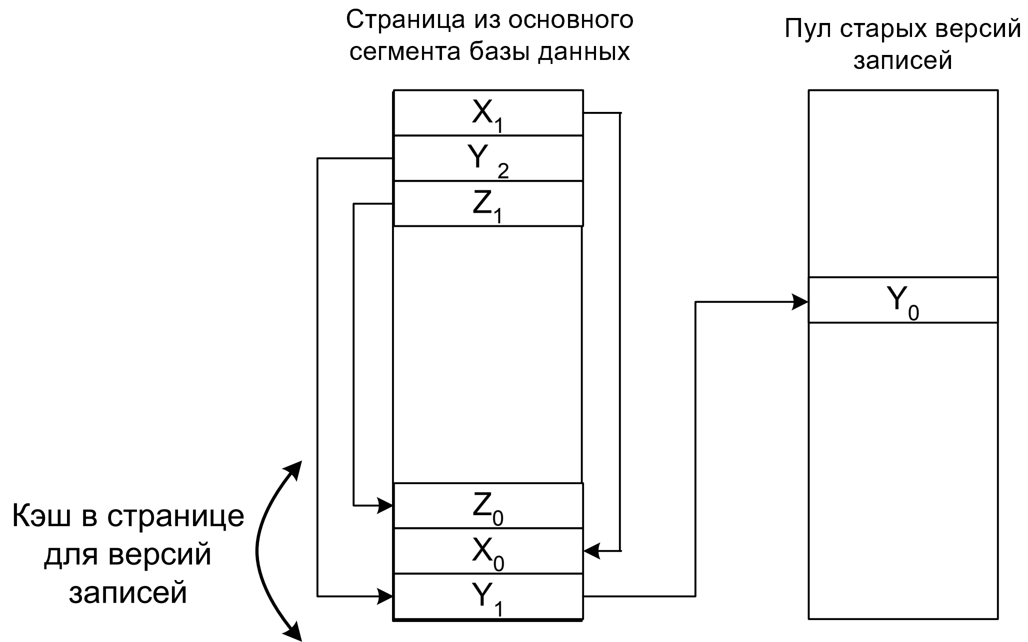


Рис. 5.2: Физическая организация БД с версионностью на уровне записей

от одного узла к другому по указателям, то и общее время выполнения запросов также может существенно увеличиться. Кроме того, при переходе необходимо постоянно проверять, подходит ли данная версия узла для транзакции или нет, что также может существенно увеличить время переходов. Наконец, версионный механизм не является прозрачным для подсистемы выполнения запросов, что существенно усложняет его реализацию.

Теперь рассмотрим реализацию версионности на уровне блоков. В этом случае при изменении узла n_1 будет создана копия блока. Тогда переход от узла n_2 к узлу n_1, выполненный транзакцией T_2, *всегда* требует обращение к другому (копии) блоку, что также снижает эффективность операции перехода по указателю. Проблема, связанная с усложнением реализации подсистемы выполнения запросов, также остается.

Основная идея решения вышеописанных проблем заключается в том, что необходимо поддерживать версионность на уровне блоков, а также необходимо ввести дополнительный уровень адресации блоков. Физические адреса блоков идентифицируют физическое расположение блока (например на диске), а логические адреса - это адреса в слоистом адресном пространстве, то есть адреса, с которыми работает подсистема выполнения запросов. Все версии одного блока обладают разными физическими адресами, но одним и тем же логическим адресом. Все связи между узлами XML-документа реализуются при помощи логических адресов. Кроме того, физический адрес последней версии блока совпадает с его логическим адресом.

Фактически в одноверсионной базе данных физический и логический адрес блока совпадают, а в многоверсионной базе данных логический адрес блока совпадает с физическим адресом последней версии блока, что позволяет эффективно определять (выходить на) последнюю версию блока. Подсистема выполнения запросов работает с логическими адресами, и при попытке отобразить очередной блок в памяти ей "подсовывается" необходимая версия блока, причем эта версия отображается на то же место, на какое был бы отображен блок с физическим адресом, равным логическому адресу рассматриваемого блока. В случае, если в памяти есть блок с требуемым логическим адресом, то гарантируется, что это – требуемая версия блока, и никаких накладных расходов, связанных с поиском требуемой версией блока, нет. А поскольку при большинстве переходов по указателю блок уже отображен в память, то и накладных расходов на косвенные переходы нет. Кроме того, такой подход позволяет полностью избежать какого-либо управления версиями в подсистеме выполнения. Фактически, весь версионный механизм можно "спрятать" в менеджер буферной памяти.

4. *Версионный механизм не должен приводить к неограниченному росту размера базы данных.*

   Одним из главных недостатков версионных протоколов является большой рост размера базы данных. Действительно, например, в протоколе ROMV может быть создано неограниченное количество версий одного элемента базы данных, что на практике неприемлимо. Необходимо ограничить максимально возможное количество версий.

   В данной работе эта проблема решается путем введения консистентных снимков базы данных, на основе которых можно выполнять читающие транзакции. Достаточно двух консистентных снимков базы данных (это снимки базы данных, в которых представлены изменения, произведенные уже зафиксированными транзакциями), чтобы можно было выполнять читающие и изменяющие транзакции без конфликтов, и при этом время от времени проводить операции обновления снимков базы данных. Версии элементов, не входящие в потенциальные или существующие снимки базы данных можно удалять. Мы показываем, что используя два консистентных снимка базы данных, можно ограничить до четырех максимальное количество версий элементов базы данных.

5. *Методы управления транзакциями не должны накладывать какие-либо ограничения на политику вытеснения блоков из буферной памяти.*

   По мнению автора, это требование очень важно, поскольку какие-то дополнительные правила вытеснения блоков (например, для обеспечения атомарности транзакций)

могут существенно повлиять на эффективность операций переходов по указателям, поскольку при вытеснении на диск блок выгружается из буферной памяти, и "разрывается" отображение из процесса, выполняющего запросы, на буферную память. Если это отображение будет часто "разрываться", то операция перехода по указателю будет менее эффективной.

5.2 Снимки базы данных и их применение для изоляции читающих и изменяющих транзакций

Далее в этой главе мы подразумеваем, что при первом изменении элемента базы данных транзакцией T создается новая копия (или *версия*) этого элемента, и изменения производятся в новой версии элемента. Кроме того, мы будем считать, что с каждой версией элемента базы данных ассоциируется временная метка[1] ее создания, а также идентификатор транзакции, создавший эту версию.

Перед тем, как привести определение консистентного снимка базы данных, введем несколько обозначений. Транзакции, осуществляющие только чтение элементов базы данных, мы будем обозначать символом T^r. Транзакции, осуществляющие как чтение элементов базы данных, так и их изменение, будем обозначать символом T^w. Элемент базы данных будем обозначать символом x, а символом x_i - i-ю физическую версию элемента x. Наконец, временную метку версии x_i мы будем обозначать символом $ts(x_i)$, а идентификатор транзакции, создавший эту версию, - символом $tid(x_i)$.

Определение 12. *Консистентным снимком базы данных, сделанным в момент времени t, мы будем называть такое состояние базы данных, в котором каждый элемент x базы данных представлен версией x_i, удовлетворяющей двум условиям: (1) x_i создана транзакцией, зафиксированной в момент времени, меньший чем t, и (2) не существует другой версии x_j, удовлетворяющей условиям (1) и $ts(x_i) < ts(x_j) < t$.*

Консистентный снимок базы данных, сделанный в момент времени t, мы будем обозначать символом S_t.

Далее мы будем предполагать, что версионность базы данных поддерживается на уровне блоков. Таким образом, при изменении какого-либо узла некоторой транзакцией T создается копия всего блока, в котором хранится узел. После этого все изменения блока транзакцией T будут производиться в этой копии.

[1] Мы считаем, что в системе есть некоторый механизм поддержки времени, на основе которого генерируются временные метки версий.

Замечание 4. *В разделе 5.5 мы обосновываем выбор версионности на уровне блоков, а не на уровне отдельных узлов XML-документа.*

Как мы уже упомянули выше, с каждой версией блока x_i ассоциируются временная метка $ts(x_i)$ и идентификатор $tid(x_i)$ транзакции. При этом они присваиваются версии блока при ее создании и не изменяются за все время жизни этой версии. В дополнение к этому, в системе поддерживается список активных T^w транзакций, который мы будем обозначать символом $ActiveT^w$. В этот список динамически добавляются идентификаторы новых T^w транзакции, и из него удаляются идентификаторы завершившихся T^w транзакции[2]. Удаление идентификатора транзакции из $ActiveT^w$ неявно подразумевает, что версия x_i, созданная этой транзакцией, становится последней зафиксированной версией x. Список активных T^w транзакций в момент времени t мы будем обозначать символом $ActiveT^w_t$

Предположим, что в момент времени $t^{snapshot}$ в системе необходимо сделать консистентный снимок базы данных (обозначим его символом $S_{t^{snapshot}}$). Для этого необходимо запомнить текушее время $t^{snapshot}$ и текущий список активных транзакций $ActiveT^w_{t^{snapshot}}$. Выбор версии элемента x, соответствующей снимку $S_{t^{snapshot}}$, делается следующим образом: находится версия x_i, удовлетворяющая условиям: $ts(x_i) < t^{snapshot}$ (и при этом $\nexists x_j : ts(x_i) < ts(x_j) < t^{snapshot}$) и $tid(x_i) \notin ActiveT^w_{t^{snapshot}}$.

Определение 13. *Мы будем говорить, что транзакция T^r выполняется на основе консистентного снимка S_t, если она “видит” состояние базы данных, определяемое этим снимком.*

Для полной изоляции T^r и T^w транзакций достаточно поддерживать один снимок базы данных, на основе которого можно выполнять все T^r транзакции, а T^w транзакции выполнять на основе двухфазного протокола. Однако в этом случае операция продвижения (обновления) снимка должна запретить запуск новых T^r транзакций, дождаться завершения выполнения всех текущих T^r транзакций (или выполнить их откат), сделать продвижение снимка и только после этого запустить новые T^r транзакции. Таким образом, в случае использования одного снимка базы данных операция продвижения снимка неявно блокирует T^r транзакции. Поэтому мы используем *два* консистентных снимка базы данных: S_{t_0} и S_{t_1} ($t_0 < t_1$), на основе которых можно проводить операцию продвижения снимков без влияния на T^r транзакции. Снимок S_{t_0} мы будем называть старым, а снимок S_{t_1} – новым.

Все активные T^r транзакции делятся на два списка. В первый список $Lst^{t_0}_{T^r}$ входят T^r транзакции, выполняемые на основе снимка S_{t_0}, а во второй список $Lst^{t_1}_{T^r}$ входят T^r

[2]По умолчанию мы подразумеваем, что при завершении T^w транзакции ее идентификатор сразу удаляется из списка $ActiveT^w$. Случай, при котором идентификатор T^w транзакции удаляется из списка $ActiveT^w$ через некоторое время после завершения T^w транзакции, обсуждается в разделе 5.6.2.

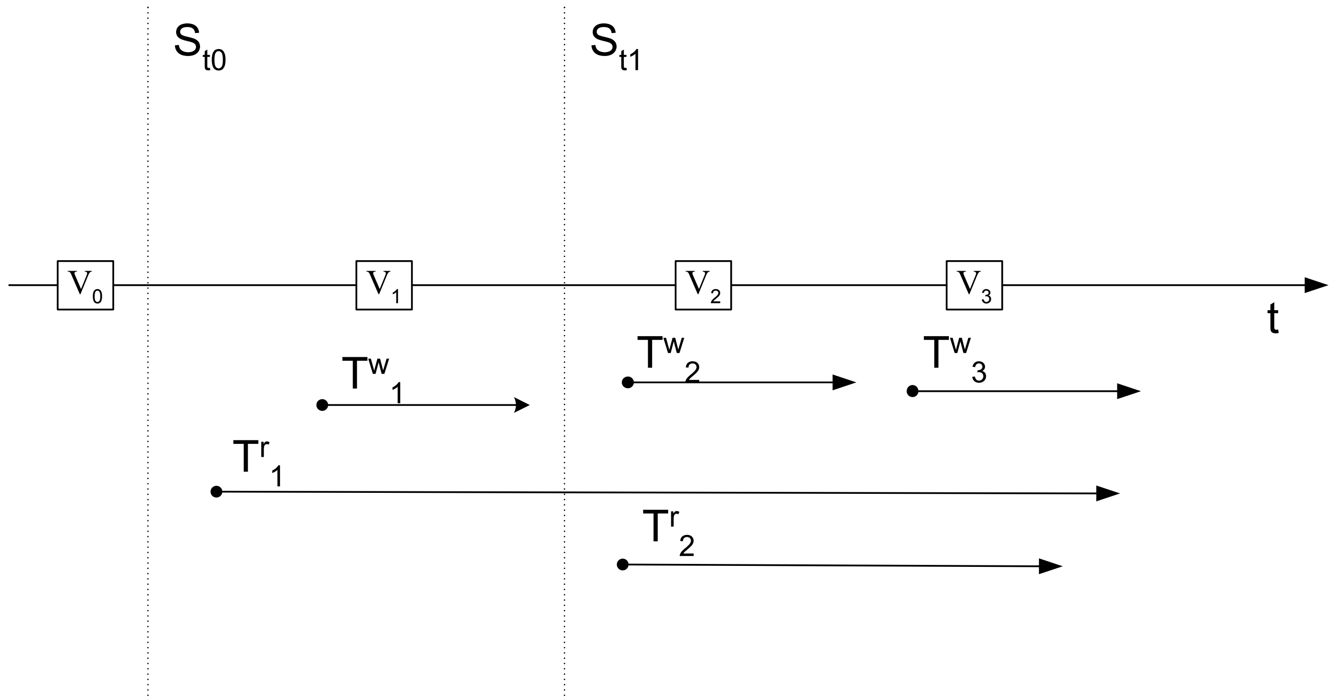


Рис. 5.3: Пример ситуации с 4-мя версиями одного блока

транзакции, выполняемые на основе снимка S_{t_1}. При этом вновь начавшиеся T^r транзакции выполняются на основе снимка S_{t_1}, а старые T^r транзакции, которые были запущены до образования снимка S_{t_1} выполняются на основе снимка S_{t_0}. В момент времени (обозначим его символом t^{adv}), когда список $Lst^{t_0}_{T^r}$ становится пустым, происходит операция *продвижения* снимка, при которой новый снимок становится старым, а консистентный снимок базы данных, сделанный в момент времени t^{adv}, становится новым. Таким образом, T^r транзакции, которые выполнялись до t^{adv} на новом снимке, после продвижения снимка будут выполняться на старом снимке (поскольку это тот же самый снимок), а T^r транзакции, начавшиеся после t^{adv}, будут выполняться на новом снимке. Детальное обсуждение продвижения снимков приводится в разделе 5.3.

Поскольку новые T^r транзакции выполняются всегда на основе нового снимка, то список $Lst^{t_0}_{tr}$ транзакций не растет и через некоторое время становится пустым, что позволяет выполнить очередную операцию продвижения снимка без приостановки новых T^r транзакций. Кроме того, за счет выполнения T^r транзакций на основе только двух снимков мы ограничиваем максимальное количество версий одного блока числом 4. Действительно, в крайнем случае необходимы две версии, образующие старый и новый снимок, одна версия, представляющая собой последнее зафиксированное состояние блока, и, наконец, еще одна версия необходима транзакции для изменения блока.

На Рисунке 5.3 изображена ситуация, при которой возникает четыре версии одного

блока. Мы предполагаем, что транзакции T_1^r и T_2^r являются читающими, а транзакции T_1^w, T_2^w и T_3^w – изменяющими. Кроме того, мы предполагаем, что вся база данных состоит из одного блока.

Транзакция T_1^r выполняется на основе снимка S_{t_0}, и поэтому версия V_0 не может удаляться до конца T_1^r. Транзакция T_1^w создала новую версию V_1 блока, образовавшую новый снимок S_{t_1}, на основе которого выполняется транзакция T_2^r. Аналогично, версия V_1 должна существовать до завершения T_2^r. Изменения транзакции T_2^w отражаются в версии V_2, которая после фиксации T_2^w становится последней зафиксированной версией блока. Наконец, при работе T_3^w создает четвертую версию V_3, в которой производятся изменения. Таким образом, в момент модификации блока транзакцией T_3^w в базе данных должны поддерживаться четыре версии блока: V_0 для T_1^r, V_1 для T_2^r, V_2 как последняя зафиксированная версия и, наконец, V_3 как "рабочая" версия транзакции T_3^w. Заметим, что после фиксации T_3^w версия V_3 становится последней зафиксированной версией блока, и поэтому V_2 может удаляться.

5.3 Продвижение снимков

Как мы обсуждали в предыдущем разделе, все T^r транзакции делятся на две группы: в первую группу входят транзакции, выполняемые на основе старого снимка S_{t_0}, а во вторую группу входят транзакции, выполняемые на основе нового снимка S_{t_1}. При такой организации T^r транзакции могут читать не самое последнее состояние базы данных. В связи с этим необходимо периодически проводить операцию продвижения (обновления) снимков базы данных.

В нашем методе продвижение снимков осуществляется при условии, что один из списков T^r транзакций ($Lst_{T^r}^{t_0}$ или $Lst_{T^r}^{t_1}$) становится пустым. Возможны два случая.

В первом случае становится пустым список $Lst_{T^r}^{t_0}$. Тогда новый снимок становится старым, а снимок базы данных, сделанный в текущий момент времени, становится новым. Таким образом, выполняются следующие действия:

$Lst_{T^r}^{t_0} := Lst_{T^r}^{t_1}$; $ActiveT_{t_0}^w := ActiveT_{t_1}^w$; $t_0 := t_1$;

$Lst_{T^r}^{t_1} := \emptyset$; $ActiveT_{t_1}^w := ActiveT_{t_{current}}^w$; $t_1 := t_{current}$;

Напомним, что $ActiveT_t^w$ определяет список активных транзакций на момент времени t, а $t_{current}$ определяет текущее время.

Во втором случае становится пустым список $Lst_{T^r}^{t_1}$. Тогда старый снимок так и остается старым, а новый снимок обновляется снимком на текущий момент. При этом выполняются следующие действия:

$Lst_{T^r}^{t_1} := \emptyset$; $ActiveT_{t_1}^w := ActiveT_{t_{current}}^w$; $t_1 := t_{current}$;

Вышеуказанные операции позволяют реализовать продвижение снимков. Причем эти

операции могут выполняться в оперативной памяти и не требуют доступа к диску. Поэтому они выполняются эффективно.

Продвижение снимка происходит только в случае, если один из списков $Lst^{t_0}_{T^r}$ или $Lst^{t_1}_{T^r}$ становится пустым. Очевидный вопрос, который при этом возникает: возможна ли ситуация, при которой ни один из списков никогда не станет пустым ? Ответ на этот вопрос отрицательный, поскольку в список $Lst^{t_0}_{T^r}$ новые T^r транзакции *никогда* не добавляются, и поэтому при завершении всех транзакций из этого списка он станет пустым.

5.4 Отображение логических версий на физические версии

При применении версионного метода на основе двух снимков существует 4 логические версии одной страницы: (1) версия страницы, необходимая для старого снимка S_{t_0}, которую мы будем обозначать $x_{S_{t_0}}$, (2) версия страницы, необходимая для нового снимка S_{t_1}, которую мы будем обозначать $x_{S_{t_1}}$, (3) последняя зафиксированная версия страницы (эта версия будет использоваться в очередном снимке), которую мы обозначим x_{LC} и (4) версия страницы, в которой производятся изменения текущей транзакцией, которую мы будем обозначать x_W (эта версия отражает незафиксированное состояние страницы).

Однако физически не всегда должны существовать 4 версии страницы, поскольку, например, логические версии $x_{S_{t_0}}$, $x_{S_{t_1}}$ и x_{LC} могут представляться одной и той же физической версией страницы в случае, если некоторый промежуток времени страницу не изменяла ни одна транзакция. Кроме того, логическая версия x_W вообще не соответствует ни одной физической странице, если нет активных транзакций, работающих со страницей x. Поэтому мы будем говорить, что одной физической версии x_i страницы соответствует множество логических версий $LVSet(x_i)$.

При наступлении определенных событий множество $LVSet(x_i)$ для физической версии x_i может измениться. Такую ситуацию мы будем называть переходом физических версий от одного множества логических версий к другому. В случае, если физическая версия x_i перестает соответствовать хотя бы одной логической версии (т.е $LVSet(x_i) = \emptyset$), эта физическая версия может удаляться.

Ниже мы описываем четыре правила, в соответствии с которыми изменяется множество $LVSet(x_i)$.

- *Правило 1.* При первой модификации страницы x транзакцией T^w создается новая физическая версия $x_i : LVSet(x_i) = \{x_W\}$.

- *Правило 2.* При фиксации транзакции T^w физической версии x_i, созданной этой транзакцией, ставится в соответствие множество $LVSet(x_i) = \{x_{LC}\}$, а для физической версии $x_j : x_{LC} \in LVSet(x_j)$ ставится в соответствие множество $LVSet(x_j) \setminus \{x_{LC}\}$.

- *Правило 3.* При продвижении снимка S_{t_0} ($Lst^{t_0}_{T^r}$ становится пустым) происходят следующие изменения. Для версии $x_i : x_{S_{t_0}} \in LVSet(x_i)$ ставится в соответствие множество $LVSet(x_i) = LVSet(x_i) \setminus x_{S_{t_0}}$. Для версии $x_i : x_{S_{t_1}} \in LVSet(x_i)$ ставится в соответствие множество $LVSet(x_i) = \{LVSet(x_i) \setminus x_{S_{t_1}}\} \cup \{x_{S_{t_0}}\}$. Наконец, для версии $x_i : x_{LC} \in LVSet(x_i)$ ставится в соответствие множество $LVSet(x_i) = LVSet(x_i) \cup x_{S_{t_1}}$.

- *Правило 4.* При продвижении снимка S_{t_1} ($Lst^{t_1}_{T^r}$ становится пустым) происходят следующие изменения. Для версии $x_i : x_{S_{t_1}} \in LVSet(x_i)$ ставится в соответствие множество $LVSet(x_i) = LVSet(x_i) \setminus x_{S_{t_1}}$. Для версий $x_i : x_{LC} \in LVSet(x_i)$ ставится в соответствие множество $LVSet(x_i) = LVSet(x_i) \cup x_{S_{t_1}}$.

Замечание 5. *Важно заметить, что переход физической версии от одного множества логических версий к другому множеству логических версий происходит неявно. В базе данных ни с одной физической версией не связывается множество соответствующих ему логических версий. Вместо этого соответствие физической версии множеству логических версий определяется динамически при обращении к странице транзакцией.*

На рисунке 5.4 изображен пример переходов для версий одного блока. Мы предполагаем, что в момент времени t_1 существует только одна версия блока x - версия x_0. Вертикальные линии обозначают момент времени, в который отображение физических версий на логические версии изменяется. В момент времени t_2 транзакция T^w_1 создает новую версию x_1 блока x, и согласно правилу 1 изменяется отображение версий. Далее мы предполагаем, что между моментами времени t_2 и t_3 произошло продвижение снимка (например, по причине фиксации некоторых читающих транзакций, которые не изображены на рисунке), что, однако, не приводит к изменению отображения физических версий на логические для блока x. Отметим, что транзакции T^r_1 и T^r_2 будут выполняться на разных снимках. В момент времени t_3 транзакция T^w_1 фиксируется, и согласно правилу 2 версия x_1 отображается на логическую x_{LC} версию. В момент времени t_4 новая транзакция T^w_2 создает новую физическую версию x_2 блока x, и согласно правилу 1 эта версия отображается на логическую версию x_w. В момент времени t_5 изменяются отображения аналогично ситуации в момент t_3. В момент времени t_6 фиксируется транзакция T^r_1, что приводит к продвижению снимков, поскольку транзакция, связанная со старым снимком, завершается. Поэтому изменение отображений осуществляется в соответствии с Правилом 3. Правило 3 также

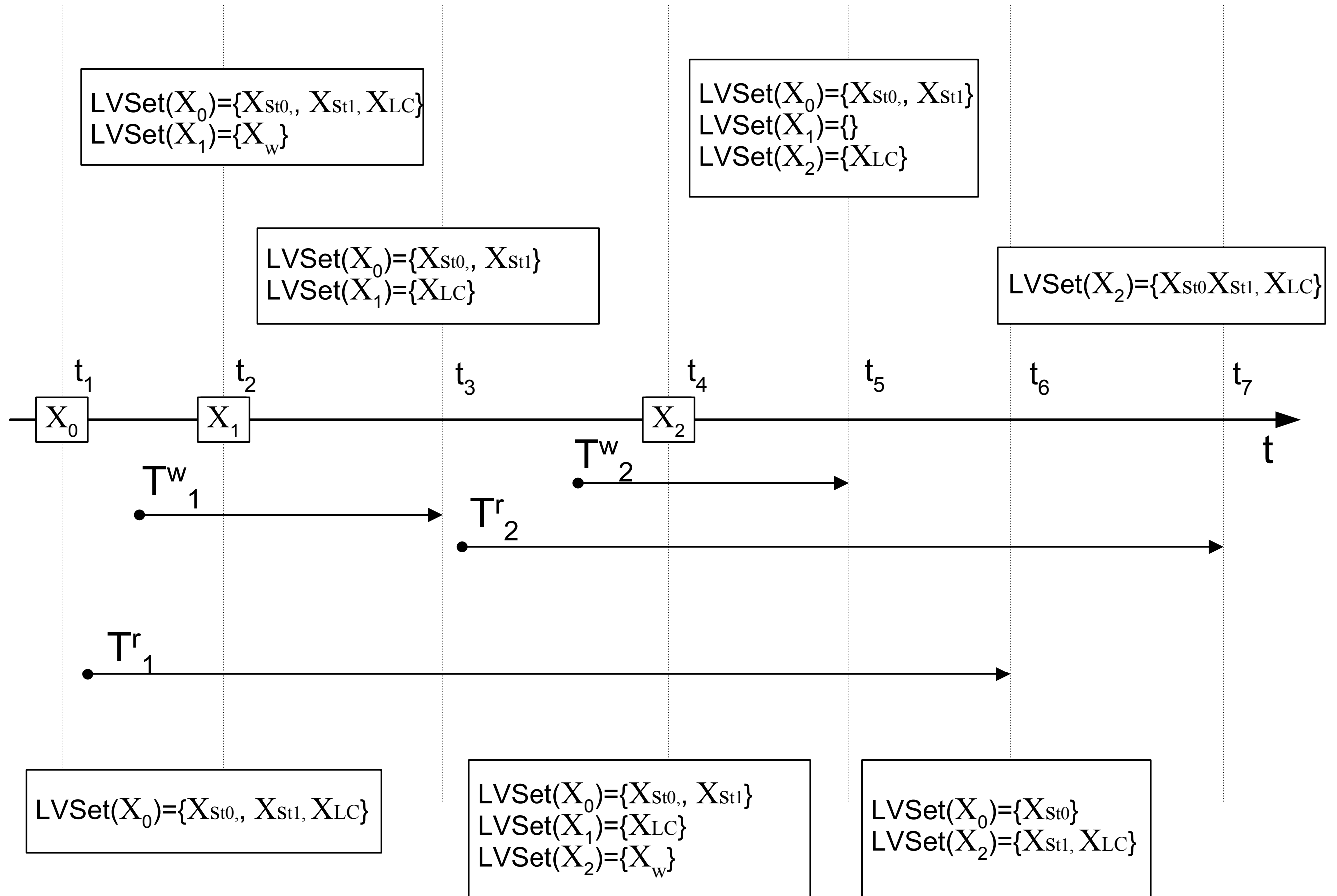


Рис. 5.4: Пример применения правил переходов

определяет новое отображение версий в момент времени t_7, когда фиксируется транзакция T_2^r.

5.5 Адресация версий и идентификация страниц из снимков базы данных

Механизм адресации версий страниц базы данных играют важнейшую роль, поскольку с ним связана эффективность операции перехода по указателю. В описываемом механизме адресации предлагается выделить два уровня адресации: логический и физический. Все связи между узлами XML-документа мы описываем при помощи адресов логического уровня, а поиск необходимой версии осуществляется по физическим адресам. При этом поиск версии по физическим адресам делается не при каждом обращении к блоку (по логическому адресу), а только в случае, если *xptr* этого блока еще не отображен в адресное пространство подсистемы выполнения запросов. Если же блок с требуемым логическим адресом отображен

в адресное пространство транзакции, то гарантируется, что это требуемая версия для транзакции, и никаких дополнительных действий по поиску необходимой версии данного блока выполнять не нужно. В результате, за счет применения логической и физической адресации, дорогостоящая операция поиска необходимой версии, которая потенциально может требовать обращение к диску, выполняется сравнительно редко. За счет этого обеспечивается эффективность переходов по указателю в БД с несколькими версиями.

Замечание 6. *Выбор версионности на уровне блоков обусловлен необходимостью обеспечить эффективные переходы по указателям.*

Итак, каждая версия блока обладает двумя адресами: логическим и физическим, которые мы будем обозначать ptr_{log} и ptr_{phys} соответственно. Естественный вопрос, который при этом возникает: каким образом по логическому адресу искать необходимую версию блока?

В предлагаемом методе логический адрес блока *всегда* совпадает с физическим адресом последней (возможно, еще не зафиксированной) версии блока. Физические адреса остальных версий блока с данным логическим адресом хранятся в заголовке последней версии блока. В дополнение к этому в заголовке последней версии блока хранятся временные метки и идентификаторы всех более старых версий. Поскольку гарантируется, что версий может быть максимум 4, то для хранения этой информации можно заранее выделить в блоке необходимое место, которое затем не нужно будет расширять.

Таким образом, в случае, если блок с заданным логическим адресом ptr_{log} не отображен в адресное пространство транзакции, необходимо обратиться к менеджеру буферов, чтобы в буфере оказалась требуемая версия, которая затем будет отображена в адресное пространство транзакции. При этом менеджеру буферов передается идентификатор транзакции, которая требует блок с адресом ptr_{log}. Менеджер буферов ищет в начале блок с физическим адресом равным ptr_{log}, а затем по информации в заголовке этого блока и информации о самой транзакции выбирает необходимую версию и помещает ее в буферную память (при этом, возможно, делается еще *один* переход к другому блоку). Затем этот блок отображается в адресное пространство транзакции.

Адрес блока в виртуальном адресном пространстве процесса транзакции определяется функцией F трансляции адресов САП в адреса в процессе транзакция. В случае 32-х адресной ОС результатом функции F является адрес равный младшими32-м битам адреса в САП, а в случае 64-х адресной ОС функция F осуществляет тождественное преобразование [61]. На Рисунке 5.5 изображено четыре активных транзакции. Транзакции T_1, T_2 и T_3 являются читающими, но первые две из них читают старый снимок, а последняя – новый снимок. Транзакция T_4 является изменяющей. Все эти транзакции работают с

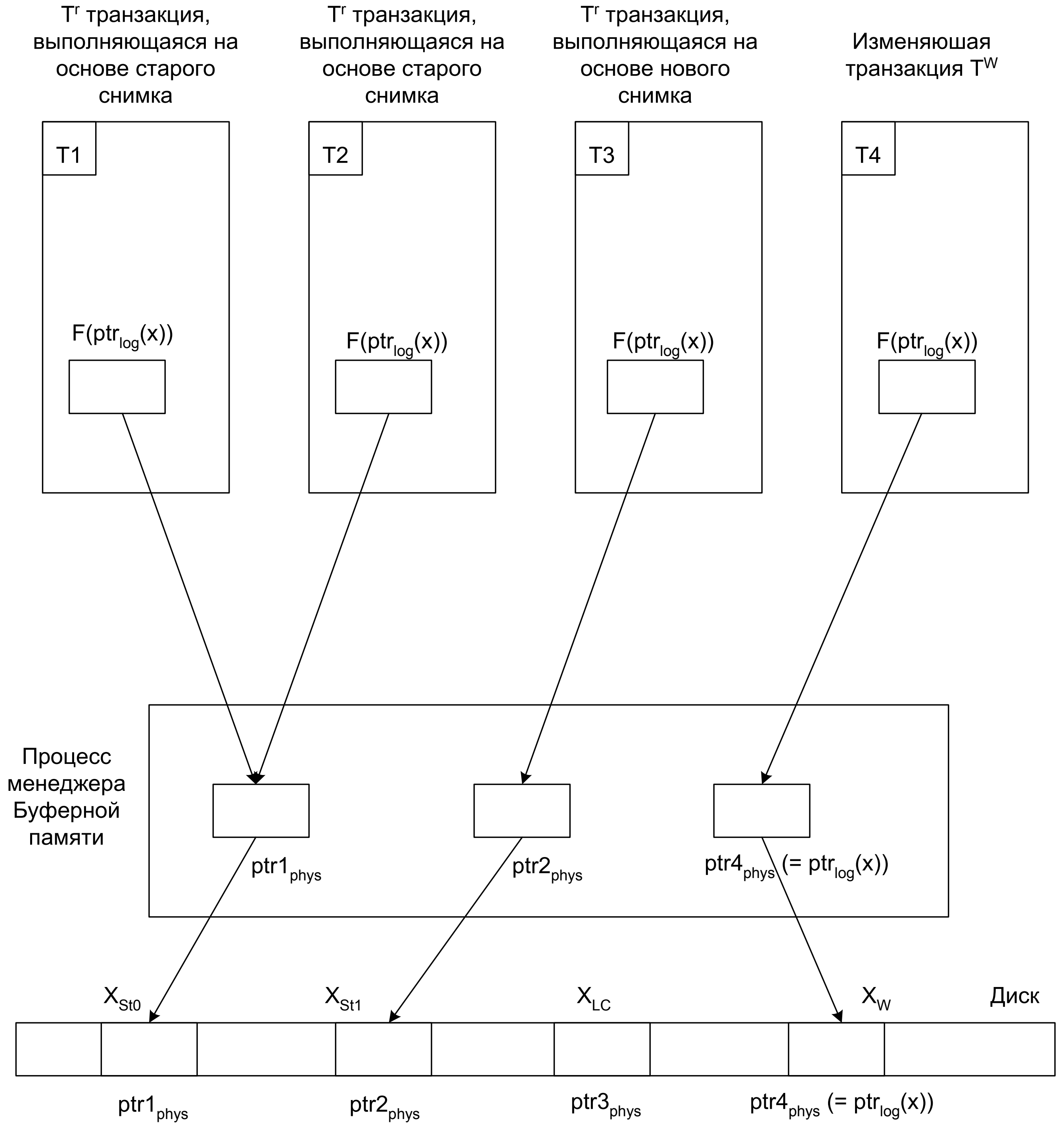


Рис. 5.5: Отображение версий в адресное пространство транзакций

одним блоком x. На рисунке 5.5 показывается принцип отображения версий блока x в адресные пространства процессов транзакций. Различные версии блока x, необходимые для транзакций T_1, T_2, T_3 и T_4, отображаются на одно и то же место памяти $F(ptr_{log}(x))$ в адресном пространстве процессов транзакций. При этом для различных версий существуют различные блоки в буферной памяти.

```
struct
{
  xptr phys_ptr; //физический адрес версии
  transaction_id tid; //идентификатор транзакции, создавший версию
  timestamp ts;//временная метка версии
} version_metadata;

typedef version_metadata[4] block_versions_info;
```

Рис. 5.6: Структура заголовка блока

На рисунке 5.6 изображены структура данных `block_versions_info`, которая хранится в заголовке последней версии блока и позволяет найти необходимую физическую страницу. Мы предполагаем, что элементы `version_metadata` в массиве `block_versions_info` *упорядочены* по временной метке `ts` по убыванию.

Далее мы описываем алгоритмы идентификации блоков, принадлежащих старому S_{t_0} и новому S_{t_1} снимкам базы данных.

Алгоритм 1 Поиск версии страницы x с логическим адресом ptr_x для снимка S_{t_0}

1: $memoffs_x \Leftarrow \text{put_in_buffer}(ptr_x)$
2: $vinfo \Leftarrow block_versions_info(memoffs_x)$
3: **for** $i = 0$ to 4 **do**
4:   **if** $vinfo[i].ts < t_0 \wedge vinfo[i].tid \notin ActiveT^w_{t_0}$ **then**
5:     $memoffs \Leftarrow put_in_buffer(vinfo[i].phys_ptr)$
6:     **return** $\{ memoffs, vinfo[i].phys_ptr \}$
7:   **end if**
8: **end for**

В первой строке алгоритма 1 последняя версия блока x помещается в буферную память, и возвращается адрес в буферной памяти (БП) $memoffs_x$ на начало этого блока. Затем в переменную $vinfo$ помещается заголовок блока. В строках 3-7 осуществляется выбор версии блока, которая была создана до образования снимка S_{t_0} (первое условие в **if** выражении) и при этои является зафиксированной (второе условие в **if** выражении). Наконец, найденный блок помещается в БП. Результатом алгоритма является физический адрес требуемого блока, а также его адрес в БП.

Алгоритм 2 описывает поиск версии страницы x с логическим адресом ptr_x для снимка S_{t_1}. Он отличается от алгоритма 1 только в строчке 4 , где время t_0 заменяется на t_1.

Алгоритм 2 Поиск версии страницы x с логическим адресом ptr_x для снимка S_{t_1}

1: $memoffs_x \Leftarrow \mathrm{put_in_buffer}(ptr_x)$
2: $vinfo \Leftarrow block_versions_info(memoffs_x)$
3: **for** $i = 0$ to 4 **do**
4: **if** $vinfo[i].ts < t_1 \wedge vinfo[i].tid \notin ActiveT^w_{t_1}$ **then**
5: $memoffs \Leftarrow put_in_buffer(vinfo[i].phys_ptr)$
6: **return** $\{ memoffs, vinfo[i].phys_ptr\}$
7: **end if**
8: **end for**

5.6 Изоляция T^w транзакций

Применение снимков базы данных позволяет полностью исключить конфликты между читающими и изменяющими транзакциям за счет применения механизма версионности данных. Однако изоляция изменяющих транзакций друг от друга требует специального рассмотрения.

Для этого мы используем строгий двухфазный протокол (так же, как и в ROMV), в соответствии с которым перед обращением к данным транзакция должна установить долговременную блокировку на эти данные, и снять ее только в конце транзакции. Естественные вопрос, который возникает при использовании блокировочного метода: какую гранулированность захватов использовать для долговременных блокировок? Существует несколько вариантов гранулированности захватов: (1) на уровне XML-документов, (2) на уровне блоков данных, (3) на уровне отдельных записей[3]. Наиболее простым является первый метод, однако он не позволяет обеспечить высокий параллелизм транзакций, работающих с одним документом. В современных СУБД, как правило, используется блокировочный механизм с гранулированностью на уровне блоков или на уровне отдельных записей.

В главе 3 мы разработали протокол XDGL сериализации XML-транзакций на основе описывающей схемы XML-документа. Гранулированность захватов в этом протоколе может доходить до отдельного узла в XML-документе. Кроме того, что XDGL специально разработан для изоляции XML-транзакций и учитывает семантические особенности операций над XML-данными, этот протокол привлекателен для использования в прирожденных XML-СУБД, поскольку он основан на описывающей схеме XML-документа, которая во многих системах используется в качестве индекса [76], основы для организации хранения XML-данных [76, 83] или вспомогательного средства для оптимизации запросов [84].

[3]В XML-документе отдельной записи соответствует узел XML-документа.

Поэтому дополнительные издержки на поддержание описывающей схемы XML-документа отсутствуют. Мы используем протокол XDGL в качестве основы для изоляции T^w транзакций. Блокировки XDGL являются долговременными и снимаются только при завершении транзакции в соответствии с правилами двухфазного протокола.

Гранулированность блокировок тесно связана с гранулированностью версий. В разделе 5.5 мы обосновали выбор гранулированности версий на уровне блоков для XML-СУБД. Во всех известных автору работах [63, 64, 66, 67, 69] гранулированность версий совпадает с гранулированностью блокировок (или гранулированность блокировок больше, чем гранулированность версий), поэтому применение методов, описанных в этих работах, приводило бы к гранулированности блокировок не меньше, чем на уровне блоков, и, как следствие, невозможности применения XDGL протокола.

Дело в том, что при комбинировании версионного механизма на уровне блоков с блокировками на уровне узлов возникают дополнительные сложности с поддержкой консистентных снимков базы данных. Как мы обсуждали выше, в блоке хранится идентификатор той транзакции, которая *последней* выполнила операцию изменения блока. Однако, даже если эта транзакция зафиксирована, то нельзя гарантировать, что не было других изменений в этом блоке со стороны других транзакций в более ранний момент времени, которые до сих пор не зафиксированы. Поэтому проверка условия, что идентификатор транзакции в блоке не принадлежит списку активных транзакций, не гарантирует, что в блоке нет незафиксированных изменений.

Кроме того, применение блокировочного механизма с гранулированностью захватов меньшей, чем блок, может приводить к ситуации, когда в блоке *всегда* могут присутствовать незафиксированные изменения какой-нибудь транзакции. При этом консистентные снимки базы данных всегда состоят только из зафиксированных блоков. Поэтому при продвижении снимка новые изменения не будут входить в новый снимок, если даже большинство изменений в блоке было произведено транзакциями, которые успешно завершились к моменту продвижения снимка. Таким образом, возникает проблема с обновляемостью данных в снимках базы данных.

Далее в данной работе предлагается набор техник, которые позволят совместить гранулированность версий на уровне блоков с гранулированностью блокировок меньшей, чем весь блок (например, на уровне узлов), и избежать вышеуказанных проблем.

Однако сначала, в разделе 5.6.1 мы опишем более простой вариант протокола изоляции T^w транзакций с гранулированностью захватов на уровне блоков. Затем в разделе 5.6.2 мы ослабим это условие и рассмотрим изоляцию T^w транзакций на основе протокола XDGL.

5.6.1 Изоляция T^w транзакций на уровне блоков

Итак, в этом разделе мы подразумеваем, что T^w транзакции устанавливают долговременные блокировки на блоки. Мы выделяем два типа блокировок:

- S_L^P - разделяемая долговременная (Long) блокировка на уровне блоков (Page).
- X_L^P - эксклюзивная долговременная (Long) блокировка на уровне блоков (Page).

Матрица совместимости этих блокировок очевидна: совместимы только разделяемые S_L^P блокировки.

Далее рассмотрим, в какой момент и как устанавливаются эти блокировки.

При переходе от одного блока к другому в транзакции могут возникнуть два случая.

В первом случае требуемый блок не отображен в процесс транзакции, и поэтому необходимо обратиться к менеджеру буферов, чтобы он поместил требуемый блок в буферную память (если, конечно, его там еще нет), и затем отобразить его на необходимое место в адресном пространстве процесса транзакции. В этом случае при обращении к менеджеру буферов необходимо проверить, не установлена ли блокировка S_L^P на этот блок[4], и, если она еще не установлена, то необходимо ее установить (возможно ожидание транзакции).

Во втором случае требуемый блок отображен в процесс транзакции. В этом случае на блок уже *точно* установлена S_L^P блокировка (это было сделано раньше в момент установления отображения), и к менеджеру буферов обращаться не нужно.

Когда блок отображен в процессе транзакции он, возможно, будет модифицироваться этой транзакцией. Поэтому необходимо гарантировать, что перед тем, как блок будет изменяться, на него будет установлена X_L^P блокировка. Для этих целей используется механизм операционной системы.

В случае, если на блок еще не установлена X_L^P блокировка, при отображении блока в процесс транзакции необходимо установить права только на чтение для этого блока. В ОС Windows это делается при помощи системного вызова `VirtualProtectEx`, а в ОС Linux при помощи системного вызова `mmap`. При попытке транзакции изменять блок возникает исключительная ситуация (в ОС Windows генерируется структурное исключение `EXCEPTION_ACCESS_VIOLATION`, а в ОС Linux сигнал `SIGSEGV`), которая обрабатывается специальным обработчиком, внутри которого устанавливается X_L^P блокировка на блок, а также устанавливаются права на этот блок как на чтение, так и на запись. Таким образом, после завершения работы обработчика на блок установлена эксклюзивная блокировка и его

[4]Для управления блокировками мы используем менеджер блокировок, построенный по принципам, описанным в [10].

можно изменять. В результате установка X_L^P блокировок прозрачна для микроопераций изменения узлов XML-документов, что существенно облегчает программирование, а также позволяет избежать дополнительных ошибок, связанных с тем, что программист забывает явно установить эксклюзивную блокировку на блок.

Кроме того, описанные механизмы установки S_L^P и X_L^P блокировок не накладывают существенные издержки, поскольку не при каждом обращении к блоку необходимо проверять (и устанавливать) требуемые блокировки.

Алгоритм 3 описывает последовательность шагов, которую необходимо выполнить менеджеру буферов для получения необходимой версии блока x (с логическим адресом ptr_x), в случае, если транзакция T^w намерена модифицировать этот блок. Мы подразумеваем, что каждый запрос к менеджеру буферов выполняется в отдельной потоке, и поэтому вызов $wait$ не блокирует остальные запросы к менеджеру буферов[5].

В первых двух строках алгоритма запрашивается эксклюзивная блокировка на блок x, и в случае, если функция `SET_LOCK` возвращает значение `waiting`, то необходимо приостановить выполнение алгоритма (и тем самым приостановить работу транзакции), пока несовместимая блокировка на блок x не будет освобождена. Начиная со строки 4, на блок x установлена блокировка X_L^P. В строках 4-5 в переменной $memoffs_x$ сохраняется адрес в БП блока с физическим адресом равным логическому адресу блока x (т.е. последняя версия блока x), а в переменной $vinfo$ сохраняется заголовок этого блока. Далее, условие в выражении if (строка 6) определяет, первая ли это первая модификация блока x транзакцией T^w. Если это так, то возвращается последняя версия блока (логическая x_W версия), в которой необходимо производить изменения. Иначе необходимо создать новую версию блока, в которой будут производиться изменения. В цикле for определяются индексы в массиве $vinfo$ логических версий $x_{S_{t_0}}$, $x_{S_{t_1}}$. Затем в переменной new_phys_ptr сохраняется физический адрес нового блока, в переменной new_buff_offs сохраняется адрес в БП слота памяти, в котором будет храниться новый блок. Копирование на новое место в БП последней версии блока (логическая x_{LC} версия) осуществляется в строке 19. После этого в строках 20-21 устанавливается новая связь между блоками в буферной памяти и их физическими адресами. Так, блок в новом слоте памяти получает физический адрес, равный логическому адресу блока x, а блок в памяти $memoffs_x$ получает физический адрес, равный new_phys_ptr. Функция $compact_and_shift$ убирает возможные пропуски в массиве $new_buff_offs.vinfo$ между логическими версиями x_{LC}, $x_{S_{t_0}}$, $x_{S_{t_1}}$ и смещает их так, чтобы первая ячейка массива $new_buff_offs.vinfo$ была свободна. Наконец, в строках 23-24 в ячейку $new_buff_offs.vinfo[0]$ заносятся требуемые значения для новой

[5]При этом мы предполагаем, что существует механизм синхронизации этих потоков. Для простоты изложения описание этого механизма мы опускаем.

Алгоритм 3 Получение версии блока x с логическим адресом ptr_x для транзакции T^w для модификации

1: **if** (SET_LOCK(X_L^P, ptr_x) == waiting) **then**
2: wait //ожидание пока блокировка не освободится
3: **end if**
4: $memoffs_x \Leftarrow$ put_in_buffer(ptr_x)
5: $vinfo \Leftarrow block_versions_info(memoffs_x)$
6: **if** ($vinfo[0].tid \in ActiveT^w_{t_{current}}$) **then**
7: **return** $\{memoffs_x, vinfo[0].phys_ptr\}$
8: **else**
9: **for** $i = 0$ to 4 **do**
10: **if** $vinfo[i].ts < t_0 \wedge vinfo[i].tid \notin ActiveT^w_{t_0}$ **then**
11: $index_{S_{t_0}} \Leftarrow i$
12: **else if** $vinfo[i].ts < t_1 \wedge vinfo[i].tid \notin ActiveT^w_{t_1}$ **then**
13: $index_{S_{t_1}} \Leftarrow i$
14: **end if**
15: **end for**
16: $index_{LC} \Leftarrow 0$
17: new_phys_ptr $\Leftarrow$ allocate_new_block()
18: $new_buff_offs \Leftarrow get_new_buffer()$
19: copy_block_in_memory(new_buff_offs, $memoffs_x$)
20: associate_buf_mem_ptr(new_buff_offs, ptr_x)
21: associate_buf_mem_ptr($memoffs_x$, new_phys_ptr)
22: compact_and_shift(new_buff_offs.vinfo, $index_{LC}$, $index_{S_{t_1}}$, $index_{S_{t_0}}$)
23: new_buff_offs.vinfo[0] $\Leftarrow \{ptr_x, current_time, tid(T^w)\}$
24: **return** {new_buff_offs, new_buff_offs.vinfo[0].phys_ptr}
25: **end if**

версии блока, и возвращается адрес требуемого блока в БП, а также его физический адрес.

Алгоритм 4 описывает последовательность шагов, которую необходимо выполнить менеджеру буферов для получения необходимой версии блока x (с логическим адресом ptr_x), в случае, если транзакция T^w намерена читать этот блок.

Все шаги алгоритма 4 очевидны. В начале устанавливается разделяемая блокировка на блок, а затем последняя версия блока помещается в БП и возвращается адрес блока в БП, а также физический адрес последней версии (совпадает с логическим адресом).

Алгоритм 4 Получение версии блока x с логическим адресом ptr_x для транзакции T^w для чтения

1: **if** (SET_LOCK(S_L^P, ptr_x) == waiting) **then**
2: wait //ожидание пока блокировка не освободится
3: **end if**
4: $memoffs_x \Leftarrow$ put_in_buffer(ptr_x)
5: **return** $\{memoffs_x, ptr_x\}$

Сборка устаревших версий блоков

Все версии блоков можно разделить на две группы. В первую группу входят версии блоков, которые составляют снимки базы данных S_{t_0} или S_{t_1}. Эти версии блоков могут удаляться только после устаревания этих снимков и при условии, что новые снимки будут составлять новые версии этих блоков. Во вторую группу попадают версии блоков, которые не входят в какой-либо снимок базы данных, а представляют последнюю зафиксированную версию блока (логическая x_{LC} версия). В случае, если была зафиксирована новая версия блока, а предыдущая x_{LC} версия по-прежнему не использовалась ни одним снимком базы данных, она может удаляться.

Ниже мы описываем алгоритм сборки устаревших страниц:

- При создании новой версии блока транзакцией T^w (см. Алгоритм 3) менеджер буферов идентифицирует физические адреса логических версий x_{LC}, $x_{S_{t_1}}$ и $x_{S_{t_0}}$. Обозначим эти физические адреса символами ptr_{LC}, $ptr_{S_{t_0}}$ и $ptr_{S_{t_1}}$ соответственно. Возможны четыре случая:
  - $ptr_{LC} = ptr_{S_{t_0}} = ptr_{S_{t_1}}$. В этом случае необходимо добавить адрес ptr_{LC} в список $GBL^1_{tid(T^w)}$.
  - $ptr_{LC} = ptr_{S_{t_1}} \wedge ptr_{S_{t_0}} \neq ptr_{S_{t_1}}$. В этом случае необходимо добавить пару $(ptr_{LC}, t_{current})$ в список $GBL^2_{tid(T^w)}$.
  - $ptr_{LC} \neq ptr_{S_{t_1}} \wedge ptr_{S_{t_0}} = ptr_{S_{t_1}}$. В этом случае необходимо добавить пару $(ptr_{LC}, t_{current})$ в список $GBL^3_{tid(T^w)}$.
  - $ptr_{LC} \neq ptr_{S_{t_1}} \wedge ptr_{S_{t_0}} \neq ptr_{S_{t_1}}$. В этом случае необходимо добавить пару $(ptr_{LC}, t_{current})$ в список $GBL^4_{tid(T^w)}$.
- При фиксации транзакции T^w для списков $GBL^1_{tid(T^w)}$, $GBL^2_{tid(T^w)}$, $GBL^3_{tid(T^w)}$ и $GBL^4_{tid(T^w)}$ выполнить следующие действия:

– Все физические адреса блоков $ptr \in GBL^1_{tid(T^w)}$ поместить в список $2AdvSecondOld$.

– Для каждой пары $(ptr, t) \in GBL^2_{tid(T^w)}$ рассмотреть следующие случаи: (1) если не было ни одного продвижения снимков с момента t[6] или были продвижения только снимка S_{t_1}, то добавить ptr в список $OneNewOrTwoOld$, (2) если с момента t было хоть одно продвижение снимка S_{t_0}, то добавить ptr в список $2AdvSecondOld$.

– Для каждой пары $(ptr, t) \in GBL^3_{tid(T^w)} \cup GBL^4_{tid(T^w)}$ рассмотреть следующие случаи: (1) если не было ни одного продвижения снимков с момента t, то удалить блок с адресом ptr, (2) если были продвижения только снимка S_{t_1}, то добавить ptr в список $OneNewOrTwoOld$, (3) если первое продвижение было для снимка S_{t_0}, а затем все продвижения были только для снимков S_{t_1} (или продвижений большее вообще не было), то добавить ptr в список $OneNewOrTwoOld$, (4) если было минимум два продвижения снимков и при этом второе (или большее) продвижение было для снимка S_{t_0}, то добавить ptr в список $2AdvSecondOld$.

- При очередном продвижении снимка S_{t_1} все блоки из списка $OneNewOrTwoOld$ удаляются, а все блоки из списка $2AdvSecondOld$ добавляются в список $1AdvOld$.

- При очередном продвижении снимка S_{t_0} все блоки из списков $OneNewOrTwoOld$ и $2AdvSecondOld$ добавляются в список $1AdvOld$, а все блоки из списка $1AdvOld$ удаляются.

Протокол изоляции транзакций на уровне блоков

При использовании изоляции транзакций на уровне блоков вся логика изоляции транзакций в XML-СУБД может быть сосредоточена в менеджере буферов (БМ). Фактически, выдавая транзакции ту или иную версию блока, БМ управляет изоляцией транзакций. Поэтому и протокол изоляции сводится к правилам выдачи необходимой версии блока. Далее мы предполагаем, что менеджер блокировок является частью менеджера буферов.

От транзакций к менеджеру буферов поступают запросы следующего вида: $q(tid, op, ptr_x)$, где tid - идентификатор транзакции, op – тип операция и ptr_x – логический адрес запрашиваемого блока. Возможные варианты для значения операции op: R – чтение блока x, W – изменение блока x, C – фиксация транзакции tid, A – откат транзакции tid[7]. Ниже описывается протокол изоляции транзакций на уровне блоков:

[6] Мы предполагаем, что в системе ведется информация о всех продвижениях снимков (время продвижения и тип продвижения) во время работы T^w транзакций.

[7] В случае, если $op = C$ или $op = A$, значение ptr_x неопределено.

- Если выполнено условие: $tid \in Lst^{t_0}_{T^r} \wedge op = R$ (запрос на чтение блока от T^r транзакции, выполняющейся на основе старого снимка), то требуемый адрес блока в БП получить при помощи Алгоритма 1. В случае, если $op = W$, генерировать динамическую ошибку и выполнить откат транзакции.

- Если выполнено условие: $tid \in Lst^{t_1}_{T^r} \wedge op = R$ (запрос на чтение блока от T^r транзакции, выполняющейся на основе нового снимка), то требуемый адрес блока в БП получить при помощи Алгоритма 2. В случае, если $op = W$, генерировать динамическую ошибку и выполнить откат транзакции.

- Если выполнено условие: $tid \in ActiveT^w_{t_{current}}$ (запрос от T^w транзакции), то необходимо рассмотреть два случая:

  – Если $op = R$, то требуемый адрес блока в БП получить при помощи Алгоритма 4. При выполнении этого алгоритма возможно ожидание S^P_L блокировки.

  – Если $op = W$, то требуемый адрес блока в БП получить при помощи Алгоритма 3. При выполнении этого алгоритма возможно ожидание X^P_L блокировки.

- Если $op = C$ (фиксация транзакции), то освободить все блокировки, установленные транзакцией tid. Далее рассмотреть три случая:

  – Если $tid \in Lst^{t_0}_{T^r}$, то выполнить $Lst^{t_0}_{T^r} \Leftarrow Lst^{t_0}_{T^r} \setminus tid$.

  – Если $tid \in Lst^{t_1}_{T^r}$, то выполнить $Lst^{t_1}_{T^r} \Leftarrow Lst^{t_1}_{T^r} \setminus tid$.

  – Если $tid \in ActiveT^w_{t_{current}}$, то выполнить $ActiveT^w_{t_{current}} \Leftarrow ActiveT^w_{t_{current}} \setminus tid$.

- Если $op = A$ (откат транзакции), то рассмотреть три случая:

  – Если $tid \in Lst^{t_0}_{T^r}$, то выполнить $Lst^{t_0}_{T^r} \Leftarrow Lst^{t_0}_{T^r} \setminus tid$ и освободить все блокировки, установленные транзакцией tid.

  – Если $tid \in Lst^{t_1}_{T^r}$, то выполнить $Lst^{t_1}_{T^r} \Leftarrow Lst^{t_1}_{T^r} \setminus tid$ и освободить все блокировки, установленные транзакцией tid.

  – Если $tid \in ActiveT^w_{t_{current}}$, то выполнить откат транзакции, а затем освободить все блокировки, установленные транзакцией tid. Процедура отката T^w транзакции детально рассматривается в разделе 5.7.4.

5.6.2 Изоляция T^w транзакций на основе протокола XDGL

Кратковременные блокировки на блоках

Протокол XDGL обеспечивает логическую целостность базы данных, однако он не гарантирует физическую целостность БД. Для обеспечения физической целостности в СУБД, как правило, используются кратковременные блокировки на уровне блоков [45], которые устанавливаются при обращении к блоку из микрооперации изменения узлов и снимаются после того, как работа с блоком в микрооперации завершается. Автор также следует этому подходу и сочетает кратковременые блокировки на уровне блоков с долговременными XDGL-блокировками.

Установка кратковременных блокировок на уровне блоков делается по аналогии с методом, описанным в разделе 5.6.1. Однако, поскольку блокировки на блоках освобождаются раньше, чем заканчивается транзакция, то необходимо гарантировать, что транзакция будет работать с блоком только после того, как необходимая блокировка будет вновь установлена.

Таким образом, при освобождении блокировки с блока необходимо гарантировать, что при последующей попытке работать с этим блоком транзакция обязательно обратится к менеджеру буферов за этим блоком (и установит на блок соответствующую блокировку). Для этого достаточно "разорвать" отображение соответствующего блока из процесса транзакции на буферную память перед тем, как освобождать блокировку с блока. Отметим, что операция "разрыва" отображения не требует откачку блока на диск. Поэтому, если транзакция через некоторое время вновь будет обращаться к этому блоку, то вполне вероятно, что он будет находиться в буферной памяти, и операция установления отображения не будет очень дорогой.

Момент освобождения кратковременных блокировок с блока может варьироваться. Например, это можно делать после завершения некоторого набора микроопераций. Однако, это не следует делать слишком часто, поскольку операция повторного установления отображения для блока ухудшает временные характеристики операции перехода по указателю. По мнению автора, разумным решением является освобождение кратковременных блокировок с блоков после выполнения в транзакции каждого выражения (statement) на XQuery/XUpdate[8].

[8] Напомним, что вся транзакция состоит из последовательности выражений, записанных на языках XQuery/XUpdate.

Блокировки XDGL

В разделе 3.5 мы определили алгоритм работы XDGL-планировщика. Этот алгоритм является общим и не зависит от СУБД, в которой применяется XDGL. Однако момент установки XDGL-блокировок может зависеть от типа СУБД. Например, в главе 4 автор предложил архитектуру с дополнительным менеджером XML-транзакций, который построен над РСУБД. В этом случае все XDGL блокировки устанавливаются до выполнения самого запроса. Фактически можно говорить, что XDGL-блокировки устанавливаются на фазе статического анализа. Ограничением этого подхода является то, что при выполнении XQuery функций, определяемых пользователем, не всегда можно определить на какие узлы схемы необходимо устанавливать блокировки, что, в свою очередь, ведет к огрублению блокировок.

При реализации XDGL в прирожденной XML-СУБД этой проблему можно избежать, запрашивая XDGL блокировки во время выполнения запроса, когда можно точно определить, на какие узлы схемы требуется установить XDGL-блокировки. Например, динамическое установление XDGL-блокировок не приводит к огрублению блокировок в случае, если в XQuery/XUpdate запросе присутствуют обращения к функциям, которые определены пользователем.

C_L^P блокировки

Каждый раз при необходимости изменять блок x T^w транзакция должна проверить, является ли зафиксированной последняя версия блока x. В случае, если версия является зафиксированной, необходимо создавать новую версию блока и изменения производить в ней, иначе изменения можно проводить в последней версии блока. Но каким образом можно убедиться, что транзакция содержит только зафиксированные данные ?

Для этих целей мы используем информацию от менеджера блокировок. При освобождении кратковременных блокировок X_L^P они не полностью удаляются из менеджера блокировок, а конвертируются в блокировки C_L^P, которые совместимы как с S_L^P, так и с X_L^P блокировками. C_L^P блокировка освобождается при завершении транзакции. Единственное предназначение C_L^P блокировки – определить, является ли последняя версия блока зафиксированной. Проверка осуществляется следующим образом: если на блок установлена хотя бы одна C_L^P блокировка, то в блоке есть изменения, которые еще не зафиксированы, иначе все изменения в блоке зафиксированы.

Замечание 7. *Блокировка X_L^P устанавливается, только если сработал механизм защиты памяти. Поэтому X_L^P блокировка устанавливается на блок только в случае, если блок действительно будет модифицироваться. Поэтому невозможна ситуация, при которой*

X_L^P *блокировка на блок установлена, а блок тем не менее изменяться транзакцией не будет.*

Граф зависимостей между T^w транзакциями

Для решения проблем, связанных с поддержкой консистентных снимков базы данных, а также обновляемостью данных в снимках при условии, что один блок одновременно могут изменять несколько транзакций, предлагается использовать *граф зависимостей между T^w транзакциями (DTG)*.

Граф DTG является неориентированным. Вершинами этого графа являются транзакции. Граф строится по следующему правилу. Если транзакция с идентификатором tid_i намерена модифицировать блок x (последнюю версию этого блока обозначим x^k), и при этом на блок x другой транзакцией установлена блокировка C_L^P, то создать две новые вершины в графе (если они до этого не были созданы) с именами tid_i и $tid(x^k)$, пометить их меткой R *(run)* и соединить их дугой. По построению DTG может быть несвязанным графом.

Перед фиксацией T^w транзакции необходимо выполнить следующие действия:

- Проверить условие $tid(T^w) \in DTG$.
  - Если условие выполнено, то пометить узел $tid(T^w)$ меткой P *(prepared)*.
  - Если условие не выполнено, то удалить $tid(T^w)$ из списка $ActiveT^w_{t_{current}}$.
- Проверить, нет ли в графе DTG максимально связанного подграфа, все узлы которого помечены меткой P. Если такой подграф найден, то из списка $ActiveT^w_{t_{current}}$ можно удалить все идентификаторы транзакций, для которых существуют вершины в найденном подграфе. Затем можно удалить найденный подграф.

В результате список $ActiveT^w_{t_{current}}$ можно разделить на два списка. В первый список, который мы будем обозначать $RunT^w_{t_{current}}$, входят T^w транзакции, которые запущены в момент времени $t_{current}$. Во второй список, который мы будем обозначать $PreparedT^w_{t_{current}}$, входят T^w транзакции, которые завершились к моменту времени $t_{current}$, но при этом входят в какой-то из подрафов в графе DTG.

Фактически, транзакции из списка $PreparedT^w_{t_{current}}$ являются зафиксированными, но изменения этих транзакций не будут видны для T^r транзакций. Это связано с тем, что в блоках, которые изменяли T^w транзакции из списка $PreparedT^w_{t_{current}}$ могут находиться незафиксированные изменения транзакций из списка $RunT^w_{t_{current}}$. В случае, если найден максимальный связанный подграф, у которого все узлы представляют зафиксированный

транзакции (но помеченные меткой P), все блоки, которые модифицировали эти транзакции, являются зафиксированными и могут использоваться в снимках.

Таким образом, за счет изменения правил удаления идентификаторов транзакций из списка $ActiveT^w_{t_{current}}$ мы достигаем результата, когда алгоритмы идентификации блоков для старого и нового снимков базы данных остаются такими же, как и алгоритмы, описанные в разделе 5.5.

Возможна ситуация, при которой граф DTG будет постоянно увеличиваться, но при этом нельзя найти максимальный связанный подграф, у которого все узлы помечены меткой P. Это может приводить к тому, что список $PreparedT^w_{t_{current}}$ будет расти, но изменения, произведенные транзакциями из этого списка, не будут видны для T^r транзакций, если даже регулярно происходят продвижения снимков.

Для предотвращения такой ситуации мы вводим параметр MAX_DTG_NODES - максимальное число узлов, которое может быть у максимально связанного подграфа графа DTG, в котором хотя бы один узел помечен меткой R. При превышении значения MAX_DTG_NODES менеджер блокировок переводится в специальный режим (propogation mode), при котором все последующие блокировки S^P_L и X^P_L на блоках удерживаются до конца транзакции, и при этом X^P_L блокировка становится несовместимой с C^P_L блокировкой. Иными словами, новые X^P_L блокировки, которые запрашиваются у менеджера блокировок в специальном режиме не совместимы с уже установленными C^P_L блокировкой. В результате в специальном режиме блок, который содержит незафиксированные данные, могут модифицировать только транзакции, которые уже модифицировали его до установки специального режима. Остальные транзакции будут ждать завершения всех этих транзакций. Специальный режим используется до тех пор, пока подграф в графе DTG, у которого максимальное число вершин превысило значение MAX_DTG_NODES, не будет удален. После этого менеджер блокировок вновь переводится в обычный режим (normal mode), при котором используются кратковременные блокировки на блоках.

Идентификация версий блоков для T^r и T^w транзакций

Идентификация версий блоков для T^r транзакций может осуществляться при помощи алгоритмов 1, 2 (для снимков S_{t_0} и S_{t_1} соответственно), описанных в разделе 5.5.

Идентификация версии блока для T^w транзакции в случае чтения блока может производиться в соответствии с алгоритмом 4, описанным в разделе 5.6.1. Но алгоритм идентификации блоков для модификации в T^w транзакциях должен быть немного изменен. Это связано с тем, что не всегда при первой модификации блока должна создаваться новая версия блока. Новая версия блока должна создаваться только в случае, если

последняя версия блока, которую транзакция намерена модифицировать, содержит только зафиксированные данные. Таким образом, условие в строке 6 алгоритма 3 необходимо заменить на условие: $tid(T^w) \in ActiveT^w_{t_{current}} \vee exists(C^P_L, ptr_x)$. Кроме того, если это условие выполнено, то в строке 7 алгоритма 3 необходимо добавить действия: $memoffs_x.vinfo[0].ts \Leftarrow current_time$ и $memoffs_x.vinfo[0].tid \Leftarrow tid(T^w)$. Такой модифицированный алгоритм мы назовем алгоритмом 3'.

4VXDGL: версионный протокол сериализации транзакций на основе XDGL

Мы предполагаем, что любая транзакция T_i состоит из набора выражений $e_j(T_i)$, написанных на языке XQuery/XUpdate. Последним выражением транзакции T_i являются операция C (фиксация транзакции) или A (откат транзакции). В свою очередь, для выполнения каждого выражения $e_j(T_i)$ транзакция T_i обращается с запросами $q(tid(T_i), op, ptr_x)$ к менеджеру буферов. Возможные варианты для значения операции op: R – чтение блока x, W – изменение блока x. Ниже описываются 4VCDGL протокол изоляции транзакций.

Замечание 8. *Для простоты изложения мы предполагаем, что XDGL блокировки устанавливаются до выполнения выражения $e_j(T_i)$, но, как мы обсуждали выше в разделе 5.6.2, XDGL-блокировки могут устанавливаться динамически в ходе выполнения выражения $e_j(T_i)$.*

- Перед выполнением очередного выражения $e_j(T_i)$ на языке запросов XQuery/XUpdate установить XDGL блокировки на соответствующей схеме (или даже нескольких схемах) XML-документа. XDGL-планировщик, описанный в разделе 3.5, определяет, какие блокировки необходимо установить для выражения $e_j(T_i)$, и на какие узлы схемы. При установке XDGL-блокировок возможно ожидание.

  В ходе выполнения выражения $e_j(T_i)$ к менеджеру буферов приходит последовательность низкоуровневых запросов $q^k_j(tid(T_i), op, ptr_x)$, которые должны быть обработаны по следующим правилам:

  – Если выполнено условие: $tid(T_i) \in Lst^{t_0}_{T^r} \wedge op = R$ (запрос на чтение блока от T^r транзакции, выполняющейся на основе старого снимка), то требуемый адрес блока в БП получить при помощи Алгоритма 1. В случае, если $op = W$, генерировать динамическую ошибку и выполнить откат транзакции.

  – Если выполнено условие: $tid(T_i) \in Lst^{t_1}_{T^r} \wedge op = R$ (запрос на чтение блока от T^r транзакции, выполняющейся на основе нового снимка), то требуемый адрес блока в БП получить при помощи Алгоритма 2. В случае, если $op = W$, генерировать динамическую ошибку и выполнить откат транзакции.

  - Если выполнено условие: $tid(T_i) \in ActiveT^w_{t_{current}}$ (запрос от T^w транзакции), то необходимо рассмотреть два случая: (1) если $op = R$, то требуемый адрес блока в БП получить при помощи Алгоритма 4 (возможно ожидание S^P_L блокировки), (2) если $op = W$, то требуемый адрес блока в БП получить при помощи Алгоритма 3' (возможно ожидание X^P_L блокировки).

- При завершении выполнения выражения $e_j(T_i)$ необходимо освободить все S^P_L блокировки, конвертировать все X^P_L блокировки в блокировки C^P_L и "разорвать" отображение соответствующих блоков из процесса, в котором выполняется выражение $e_j(T_i)$, на буферную память.

- Если $e_j(T_i) = C$ (фиксация транзакции), то освободить все блокировки, установленные транзакцией T_i (включая XDGL блокировки и кратковременные блокировки на блоках). Далее рассмотреть три случая:

  - Если $tid \in Lst^{t_0}_{T^r}$, то выполнить $Lst^{t_0}_{T^r} \Leftarrow Lst^{t_0}_{T^r} \setminus tid$.
  - Если $tid \in Lst^{t_1}_{T^r}$, то выполнить $Lst^{t_1}_{T^r} \Leftarrow Lst^{t_1}_{T^r} \setminus tid$.
  - Если $tid \in ActiveT^w_{t_{current}}$, то проверить условие $tid(T_i) \in DTG$ (см. подробности в разделе 5.6.2). Если условие выполнено, то пометить узел $tid(T_i)$ в графе DTG меткой P (prepared), иначе выполнить $ActiveT^w_{t_{current}} \Leftarrow ActiveT^w_{t_{current}} \setminus tid(T_i)$. Кроме того, необходимо проверить, нет ли в графе DTG максимального связанного подграфа, все узлы которого помечены меткой P. Если такой подграф найден, то из списка $ActiveT^w_{t_{current}}$ можно удалить все идентификаторы транзакций, для которых существуют вершины в найденном подграфе. Затем можно удалить найденный подграф.

- Если $e_j(T_i) = A$, то действия аналогичны тем, что изложены в предыдущем пункте, но в начале необходимо выполнить откат транзакции. Детальное обсуждение индивидуального отката транзакций см. в разделе 5.7.4.

Замечание 9. *Процедура сборки устаревших версий блоков в случае использования протокола 4VXDGL аналогична процедуре сборки устаревших версий, рассмотренной в разделе 5.6.1.*

Обоснование корректности протокола 4VXDGL

Теорема 2. *Протокол 4VXDGL генерирует сериализуемые планы выполнения XML-транзакций*

Доказательство. *Изоляция изменяющих T^w транзакций осуществляется на основе протокола XDGL, для которого доказано (теорема 1 в главе 3), что он генерирует сериализуемые планы выполнения транзакций. Поэтому для всех зафиксированных T^w транзакций всегда можно построить сериальный план, который будет эквивалентен плану выполнения T^w транзакций на основе протокола XDGL.*

Поскольку читающие T^r транзакции по определению не конфликтуют друг с другом, для них всегда можно построить сериальный план выполнения. Например, T^r транзакции можно сериализовать по времени старта этих транзакций.

Осталось доказать, что для смеси T^r и T^w транзакций тоже всегда можно построить эквивалентный сериальный план выполнения. Для этого достаточно показать, что снимки базы данных, на основе которых выполняются T^r транзакции, не содержат незафиксированных изменений. Докажем этот факт для нового снимка, а для старого снимка доказательство аналогично.

Предположим, что новый снимок содержит изменения, произведенные транзакциями, не зафиксированными на момент создания снимка t_1. Пусть эти изменения присутствуют в версии блока x, которую мы обозначим x^{nc}. Причем x^{nc} входит в новый снимок. Обозначим идентификатор транзакции, которая произвела изменения в x^{nc}, но при этом не зафиксирована к моменту времени t_1, символом tid_i. Покажем, что $tid(x^{nc}) \in ActiveT^{w}_{t_1}$[9]*. Возможны два случая. В первом случае $tid_i = tid(x^{nc})$. Тогда $tid(x^{nc}) \in ActiveT^w_{t_1}$ по исходному предположению, что в новом снимке присутствуют незафиксированные изменения. Во втором случае $tid_i \neq tid(x^{nc})$. Тогда существует последовательность транзакций $tid_i, tid_{i+1}, ..., tid_{i+l}, tid(x^{nc})$, которые последовательно изменяли версию x^{nc}. Поскольку транзакция tid_i не зафиксирована в момент времени t_1, по определению построения графа зависимостей между транзакциями все транзакции из последовательности $tid_i, tid_{i+1}, ..., tid_{i+l}, tid(x^{nc})$ будут присутствовать в DTG_{t_1} и между ними будут дуги (поскольку транзакция tid_i не зафиксирована, то на блок x установлена блокировка C^P_L, и последующие транзакции, изменяющие этот блок, будут заноситься в граф зависимостей). Таким образом, вершины $tid_i, tid_{i+1}, ..., tid_{i+l}, tid(x^{nc})$ образуют связанный подграф в DTG_{t_1}. Причем как минимум одна вершина (а именно tid_i) этого подграфа помечена меткой R. Из того факта, что все идентификаторы транзакций, входящие в DTG_{t_1} входят в список $ActiveT^w_{t_1}$, следует, что $tid(x^{nc}) \in ActiveT^w_{t_1}$.*

Напомним, что по определению все версии x^i блоков, относящиеся к новому снимку удовлетворяют условию $tid(x^i) \notin ActiveT^w_{t_1}$ (см алгоритм 2). Таким образом, мы получили противоречие, что x^{nc} относится к новому снимку и при этом содержит данные, проведенные транзакцией, незафиксированной к моменту времени t_1. Теорема доказана. □

[9] Напомним, что $tid(x^{nc})$ обозначает идентификатор транзакции, последней изменявшей версию x^{nc}.

Свойства протокола 4VXDGL

Теорема 3. *Максимальное количество версий одного блока, используемых в протоколе 4VXDGL, ограничено числом 4.*

Доказательство. *Это свойство протокола 4VXDGL следует из того факта, что изоляция* T^r *и* T^w *осуществляется на основе только двух снимков, а все версии, созданные между этими снимками не используются. Действительно, в крайнем случае, одна версия блока необходима для старого снимка, еще одна версия блока для нового снимка, для последней зафиксированной версии блока необходима еще одна версия, и, наконец, четвертая версия необходима для транзакции* T*, которая модифицирует последнюю версию блока ("рабочая" версия блока). При фиксации транзакции* T *"рабочая" версия блока транзакции* T *становится последней зафиксированной версией блока, и поэтому после этого предыдущая последняя версия блока больше не нужна (только если она теперь не используется каким-либо снимком) и может удаляться. Таким образом, после фиксации транзакции* T *количество версий уменьшается на одну, и поэтому при старте следующей транзакции* T' *их снова может стать не более, чем четыре. Кроме того, при устаревании снимка все версии блоков, которые использовались только этим снимком, также больше не используются протоколом 4VXDGL и могут удаляться. Теорема доказана.*

□

Определение 14. *Мы будем говорить, что две транзакции конфликтуют, если их параллельное выполнение может привести к приостановке выполнения (или даже откату) другой транзакции.*

Теорема 4. *Процедура продвижения снимков не приводит к приостановке выполнения* T^r *или* T^w *транзакций*

Доказательство. *Это свойство протокола 4VXDGL следует из алгоритма продвижения снимков (см. раздел 5.3). Действительно, процедура продвижения снимков не требует установки каких-либо блокировок, что может приводить к конфликтам с* T^r *или* T^w *транзакциями. Теорема доказана.*

□

Теорема 5. *Параллельное выполнение* T^r *и* T^w *транзакций на основе протокола 4VXDGL никогда не приводит к конфликту между этими транзакциями.*

Доказательство. *Единственная возможность конфликтов между транзакциями - это конфликты по блокировкам. Несовместимость запрашиваемой блокировки может*

привести к приостановке выполнения транзакции или даже ее откату, если транзакция попала в тупик и была выбрана "жертвой" процедурой разрешения тупиков.

T^r транзакции не устанавливают блокировки в ходе своего выполнения, поэтому они не могут быть приостановлены (или откатаны) из-за появления параллельных T^r или T^w транзакций. T^w транзакции устанавливают блокировки, и поэтому это может приводить к приостановке выполнения (или даже откату) других T^w транзакций, но никак не может привести приостановке (или откату) T^r транзакций, поскольку последние не устанавливают блокировки. Процедура продвижения снимков также не может приводить к приостановке (или откату) T^r или T^w транзакций, поскольку никакие блокировки при выполнении этой процедуры не устанавливаются. Теорема доказана.

□

5.7 Метод восстановления транзакций после мягких сбоев в системе

Основная идея метода заключается в том, что один из снимков базы данных можно использовать в качестве основы для физического восстановления базы данных в случае сбоя. Для реализации этой идеи необходимо обеспечивать для некоторого снимка базы данных свойство постоянности, т.е блоки данных, принадлежащие этому снимку, не должны удаляться при устаревании снимка. Такой снимок мы будем называть *контрольным* снимком. В результате при восстановлении после сбоя состояние базы данных будет возвращено к контрольному снимку, и затем по журналу все транзакции, которые были зафиксированы после создания контрольного снимка, но до сбоя должны быть восстановлены. Дополнительное обстоятельство, что контрольный снимок содержит только зафиксированные данные (transaction-consistent data), позволяет упростить восстановление. Но прежде, чем переходить к описанию метода, основанного на контрольных снимках, опишем физический и логический журналы.

5.7.1 Физический журнал

Вообще говоря, физический журнал представляет собой некоторую область пространства на жестком диске (как правило это отдельный файл) в который можно заносить записи переменной длины. Запись в физический журнал осуществляется не сквозным образом сразу в файл, а через некоторый буфер фиксированного размера. Если очередная запись не помещается в буфер журнала, то срабатывает процедура вытеснения буфера в файл с последующей очисткой буфера.

Каждая запись в журнале обладает своим идентификатором, который мы будем обозначать аббревиатурой LSN (Log Sequence Number). Интуитивно LSN записи в журнале можно представлять как смещение записи в файле относительно его начала[10]. В предлагаемом методе все записи физического журнала ассоциируются с каким-то блоком базы данных. При этом в заголовке каждого блока мы храним LSN последней записи в журнале, ассоциированной с этим блоком.

Для записей физического журнала применяется протокол WAL. Иными словами, перед тем как сбросить блок на диск необходимо сбросить на диск все записи физического журнала, ассоциированные с этим блоком. Для этого достаточно сбросить все записи журнала, LSN которых меньше или равен, чем LSN в блоке.

Заметим, что в нашем методе в физический журнал не будут заноситься записи, описывающее физической состояние базы блоков данных до и после проведения микрооперации (before-image и after-image). Вместо этого в физический журнал будут попадать записи, содержащие некоторую метаинформацию о блоках контрольного снимка базы данных. Поэтому размер физического журнала растет медленно. Кроме того, физический журнал зацикливается (новые записи начинают записываться поверх старых) при создании очередной контрольной точки в базе данных (см. раздел 5.7.3).

Направление перемещения по физическому журналу может осуществляться по двум направлением: вперед и назад. Перемещение вперед осуществляется при анализе физического журнала после сбоя. Перемещение назад осуществляется при восстановлении. Для поддержания этих операций в заголовке каждой записи хранится указатель на предыдущую запись (перемещение назад), и длина самой записи (перемещение вперед).

5.7.2 Логический журнал

Логический журнал также представляет собой некоторый физический файл на жестком диске. Причем запись в логический журнал осуществляется также через специальный буфер постоянного размера. Кроме того, все записи в логическом журнале могут иметь различную длину, и идентифицируются также при помощи LSN.

Подавляющее большинство записей логического журнала описывают изменения, произведенные микрооперацией. Причем эти изменения описываются *логически*. Это означает, что в журнал не попадают отдельные фрагменты состояния базы данных до и после проведения операции изменения. Вместо этого записывается информация о самой операции. Например, при вставке одного узла в XML-документ в логический журнал

[10] Обычно LSN записи журнала вычисляется как сумма смещения записи в файле и некоторого базового смещения.

достаточно занести информацию о самом узле (тип, имя), а также адрес места в базе данных (ptr_{log}), куда вставляется узел, а также, возможно, адреса левого, правого и родительского узлов.

Существенная разница между логическим и физическим журналами заключается в том, что в отличии от записей физического журнала записи логического журнала не должны следовать протоколу WAL. Вместо этого все записи логического журнала, принадлежащие одной транзакции, должны быть сброшены на диск перед фиксацией этой транзакции. Для этого достаточно при фиксации транзакции сбросить все записи логического журнала, LSN которых меньше или равен, чем LSN последней записи этой транзакции.

Все записи логического журнала, записанные одной T^w транзакцией, связываются в обратный список. В логическом журнале поддерживается операция продвижения в прямом направлении по журналу, а также операция перемещения в обратном направлении по записям одной транзакции.

5.7.3 Контрольные точки базы данных

Понятие контрольной точки

Мы предполагаем, что при наступлении определенных событий (о которых мы поговорим ниже) в системе необходимо установить очередную контрольную точку (checkpoint) [89]. Основная идея создания контрольных точек заключается в том, что после сбоя необходимо восстановить только те транзакции, которые были зафиксированы после создания контрольной точки. Далее мы будем понимать под восстановлением транзакции процедуру, которая для зафиксированных транзакций обеспечивает присутствие в базе всех изменений произведенных транзакцией (и только их), а для незафиксированных транзакций обеспечивает полный откат всех изменений произведенных транзакцией в базе данных.

В предлагаемом автором методе в процессе создания контрольной точки создается контрольный снимок базы данных. Контрольным снимком базы данных становится снимок базы данных текущий в момент создания контрольной точки. Поскольку при изоляции транзакций мы использовали временные (transient) снимки базы данных, то их прямое использование для контрольного снимка не подходит. Требуется создавать *постоянный* (persistent) снимок базы данных, никакие блоки которого не будут удаляться при устаревании снимка. Фактически, блоки постоянного снимка базы данных могут удаляться только тогда, когда этот снимок больше не нужен. В нашем случае контрольный снимок базы данных должен сохраняться до создания следующего контрольного снимка базы данных.

Создание контрольной точки

Итак, предположим, что в некоторый момент времени в системе необходимо создать очередную контрольную точку. Рассмотрим, какие для этого необходимо выполнить действия.

1. На момент работы процедуры по созданию контрольной точки запретить продвижение снимков.

2. В переменных t_{cp} и $ActiveT^{w}_{t_{cp}}$ необходимо сохранить значения переменных t_1 и $ActiveT^{w}_{t_1}$ соответственно.

3. Все блоки, составляющие контрольный снимок, нужно сделать постоянными. Эти блоки контрольного снимка делятся на две группы. Для каждой группы блоков ниже мы описываем действия, которые необходимом сделать, чтобы сделать их постоянными.

   - В первую группу входят блоки, для которых существуют более новые версии. Эти блоки входят в списки $OneNewOrTwoOld$ или $2AdvSecondOld$, поскольку эти два списка содержит все блоки нового снимка базы данных, для которых существуют более новые версии. Таким образом, необходимо сохранить содержимое списков $OneNewOrTwoOld$ и $2AdvSecondOld$ в контрольной записи, а затем обнулить содержимое этих списков. Кроме того, необходимо, чтобы в контрольной записи был сохранен набор пар вида (ptr_{log}, ptr_{phys}) (т.е логический и физический адрес каждого блока). Все пары вида (ptr_{log}, ptr_{phys}) из списка $1AdvOld$ также должны сохраниться в контрольной записи, чтобы после восстановлении эти блоки не были потеряны. Заметим, что в процедуре сборки устаревших блоков мы не сохраняли логические адреса блоков. Поэтому в случае необходимости восстанавливать СУБД после мягких сбоев в процедуре сборки устаревших страниц в списки $OneNewOrTwoOld$, $2AdvSecondOld$ и $1AdvOld$ необходимо также заносить логические адреса блоков.

   - Во вторую группу входят блоки контрольного снимка, которые на момент создания контрольной точки представляли последнюю версию блока. В этом случае при последующих созданиях новых версий этих блоков необходимо определять, что эти блоки составляют контрольный снимок и не должны удаляться. Для этого при создании новой версии блока транзакцией T^w необходимо проверить, что последняя версия блока принадлежит контрольному снимку. Условие, которое гарантирует это? следующее: $tt(x_{LC}) < t_{cp} \wedge tid(T^w) \notin ActiveT^{w}_{t_{cp}}$. Кроме того, при создании новой версии, если это условие выполнено, в физический журнал

необходимо занести запись, описывающую логический и физический адреса версии блока, составляющей контрольный снимок.

4. Сбросить на диск все версии блоков, расположенные в буферной памяти, которые соответствуют контрольному снимку базы данных. Поскольку контрольный снимок соответствует текущему снимку, необходимо сбросить на диск все физические версии блоков, соответствующие логическим $x_{S_{t_1}}$ версиям блока.

5. В контрольную запись журнала занести идентификаторы всех транзакций из списка $ActiveT^w_{t_{cp}}$. Для каждой транзакции tid_i из списка $ActiveT^w_{t_{cp}}$ занести адрес первой записи в логическом журнале этой транзакции. Обозначим его $StartLSN(tid_i)$ [11].

6. Занести в логический журнал контрольную запись и полностью сбросить на жесткий диск логический и физический журналы.

7. По мастер-записи логического журнала (в нашем методе она хранится в заголовке физического журнала) найти адрес контрольной записи предыдущего контрольного снимка. Считать из нее адреса блоков контрольного снимка (сохраненные списки $OneNewOrTwoOld$ и $2AdvSecondOld$). Поместить эти адреса в список $PrevCheckpointBlocksList$. Считать все записи физического журнала и все адреса ptr_{phys}, содержащиеся в них. Добавить их также в список $PrevCheckpointBlocksList$.

8. В мастер-запись логического журнала поместить LSN контрольной записи в журнале.

9. Запретить операцию выделения новых блоков на время выполнения последних двух шагов алгоритма.

10. Поместить блоки $PrevCheckpointBlocksList$ в список свободных блоков базы данных. Далее эти блоки будут считаться свободными.

11. Атомарным действием обнулить физический журнал и сбросить на диск мастер-запись. Это можно сделать, например, следующим образом. Необходимо хранить мастер запись в заголовке физического журнала. Тогда нужно обновить в оперативной памяти заголовок журнала так, что последующие записи будут попадать в начало журнала. И наконец сбросить сразу весь заголовок журнала на диск путем одной операции записи[12].

---

[11] Мы предполагаем, что в системе для каждой транзакции в оперативной памяти хранится LSN первой записи транзакции в логическом журнале.

[12] Обычно аппаратура жестких дисков позволяет сбрасывать один кластер диска равный 512 или более байтам атомарно. Очевидно, одного кластера диска достаточно для хранения заголовка журнала, и поэтому запись на диск заголовка будет происходить атомарно.

Замечание 10. *Процедура создания контрольной точки считается успешно завершившейся, если был сброшен на диск заголовок физического журнала (включая мастер-запись). Восстановление базы данных начинается с определения последней успешно завершившейся контрольной точки.*

Замечание 11. *Важной особенностью вышеописанного алгоритма является то, что во время его работы микрооперации изменения данных также могут работать(в отличие от алгоритма создания контрольной точки в System R [89], где на время создания контрольной точки все операции изменения приостанавливаются). Единственная синхронизация с микрооперациями выполняется в пункте 9, в котором запрещается выделять новые блоки для микроопераций до конца работы алгоритма. Это делается по той причине, что в случае, если транзакции будет выделен свободный блок, который входил в список* $PrevCheckpointBlocksList$*, и она его изменит, но при этом до конца работы алгоритма произойдет сбой, то последний успешный контрольный снимок будет поврежден. В результате восстановление базы данных будет невозможно. Условие 9 можно ослабить: можно выделять микрооперациям новые блоки, но необходимо следить, что они не входят в список* $PrevCheckpointBlocksList$*.*

Критерии создания контрольной точки могут быть различными. С одной стороны, процедура создания контрольной точки требует обращение к диску, и поэтому она может занимать достаточно длительное время, но, с другой стороны, чем чаще в системе создается контрольная точка, тем быстрее времени будет выполняться восстановление базы данных после сбоев. Автор выбрал стратегию, зависящую от двух параметров.

Первый параметр - размер физического журнала. Обозначим ее $MaxPhysLogSize$. При достижении граничного значения размера журнала $MaxPhysLogSize$ необходимо выполнить контрольную точку. Заметим, что в физический журнал попадают записи о создании новой версии блока, которая принадлежит контрольному снимку. Поэтому, ограничивая сверху физический журнал, мы тем самым ограничиваем и количество постоянных версий блоков, не представляющих x_{LC} версию.

Второй параметр - размер логического журнала от последней контрольной записи до последней записи в журнале. Обозначим этот размер $MaxLogicalLogSize$. При достижении этого параметра также необходимо создать контрольную точку.

Поясним, почему недостаточно только первого параметра. Дело в том, что может случиться ситуация, при которой все транзакции работают с одним и тем же достаточно небольшим множеством блоков. В этом случае размер физического журнала в определенный момент совершенно перестанет расти. Но размер логического журнала (от последней контрольной записи) будет постоянно увеличиваться. В

результате в случае сбоя восстановление может занять очень большое время. Именно по этой причине мы ввели два параметра. Фактически, параметр $MaxPhysLogSize$ ограничивает максимальное количество постоянных версий, не являющихся последними зафиксированными версиями (x_{LC} версии), а параметр $MaxLogicalLogSize$ ограничивает время выполнения восстановления после сбоя.

5.7.4 Индивидуальный откат транзакции

Индивидуальный откат транзакции зависит от гранулированности изоляции T^w транзакции. Выше в разделах 5.6.1 и 5.6.2 мы описали два метода изоляции T^w транзакций: (1) с уровнем гранулированности на уровне блоков, и (2) с уровнем гранулированности меньшей, чем блок (на основе протокола XDGL). Далее для каждого из этих методов мы рассмотрим процедуру отката.

В случае использования первого метода с уровнем изоляции на уровне блоков откат T^w транзакции должен проводиться по следующим правилам:

- Получить список адресов $ptr_{log}(x_i)$ всех версии блоков x_i, созданных T^w транзакцией (т.е необходимо получить все x_i^w версии блоков, созданные этой транзакцией). Например, эту информацию можно получить от менеджера блокировок, поскольку на каждый изменяемый блок должна быть установлена X_L^P блокировка.
- Для каждой версии x_i^w с адресом $ptr_{log}(x_i)$ выполнить следующие действия:
  - Если блок x_i^w отсутствует в БП, то поместить его в БП. Из заголовка этого блока определить физический адрес x_{LC} версии этого блока и рассмотреть два случая: (1) если в БП нет версии x_{LC}, то поместить ее в свободный слот БП, ассоциировать этот слот с адресом $ptr_{log}(x_i)$, поместить физический адрес x_{LC} версии в список свободных блоков, и освободить слот памяти, занимаемый x_i^w версией, (2) если в БП есть x_{LC} версия, то ассоциировать ее с адресом $ptr_{log}(x_i)$, поместить физический адрес x_{LC} версии в список свободных блоков, и освободить слот памяти, занимаемый x_i^w версией.

В случае использования метода изоляции транзакций на основе протокола XDGL процедура отката T^w транзакции выглядит по другому, поскольку удаление всех x^w версий может привести к потере изменений, проведенных другими T^w транзакциями. Для отката T^w транзакций мы используем записи из логического журнала, по которым в обратном порядке производится выполнение обратных операций. Но по логическому журналу можно производить откат только законченных микроопераций, и нельзя производить откат

незавершившихся микроопераций. Очевидно, что в T^w транзакции может быть только одна незавершившаяся операция - последняя. При этом ситуация, когда незавершившаяся микрооперация будет присутствовать при откате, может произойти в следующем случае: T^w транзакция попала в тупик в середине выполнения микрооперации, и была выбрана жертвой процедуры разрешения тупиков. Поэтому для отката неполных микроопераций мы поддерживаем в оперативной памяти журнал всех *физических* изменений, произведенных этой микрооперацией. Этот журнал обнуляется при завершении микрооперации и заполняется вновь при выполнении новой микрооперации. Поскольку этот журнал используется только для индивидуальных откатов T^w транзакций, то его не нужно сбрасывать на диск. Кроме того, из свойств локальности микроопераций (см. подробности в работе [61]) следует, что размер этого журнала не будет большим, и он может уместиться в оперативной памяти. Мы будем называть его журналом микрооперации. Таким образом, для отката T^w транзакции необходимо выполнить следующие действия:

- По журналу микроопреации откатить последнюю неполную микрооперацию (если такая имеется).
- По логическому журналу откатить все действия, произведенные транзакцией. Мы предполагаем, что запись в логический журнал об очередной микрооперации производится после обнуления журнала микрооперации, и поэтому невозможна ситуация, когда откат последней микрооперации будет производится дважды - один раз по журналу микроопераций, а затем по логическому журналу.
- Определить список физических адресов блоков x^w, для которых откатываемая транзакция T^w создала новую версию (для этого необходимо динамически в транзакции поддерживать список блоков для которых были созданы новые версии). И для каждого из этих блоков выполнить действия аналогичные действиям, описанным в процедуре отката транзакции с изоляцией T^w транзакций на уровне блоков.

5.7.5 Восстановление базы данных после сбоя

Общая идея восстановления базы данных после мягкого сбоя заключается в том, что сначала необходимо восстановить базу данных к состоянию контрольного снимка, а затем путем анализа логического журнала необходимо выполнить все изменения зафиксированных транзакций в контрольном снимке. Ниже мы детально описываем процедуру восстановления.

1. Выполнить восстановление по физическому журналу. Для этого выполнить следующие действия:

- Извлечь из мастер-записи логического журнала адрес последней контрольной записи.
- Извлечь из контрольной записи все пары (ptr_{log}, ptr_{phys}) из списков $OneNewOrTwoOld$ и $2AdvSecondOld$. Для каждой пары произвести копирование на диске блоков с адресом ptr_{phys} на место ptr_{log}.
- Для каждой записи физического журнала, начиная с последней, в обратном порядке выделить пару (ptr_{log}, ptr_{phys}). Для каждой пары произвести копирование на диске блоков с адресом ptr_{phys} на место ptr_{log}.
- Извлечь из контрольной записи все пары (ptr_{log}, ptr_{phys}) из списка $1AdvOld$. Все блоки с адресами ptr_{phys} поместить на место свободных блоков.

2. Выполнить восстановление по логическому журналу. Для этого выполнить следующие действия:

   - Из контрольной записи извлечь пары $(tid_i, StartLSN(tid_i))$.
   - Определить адрес логического журнала, начиная с которого необходимо произвести анализ на предмет зафиксированных транзакций. Обозначим этот адрес символом $RcvLSN$. Этот адрес равен $min(StartLSN(tid_i))$.
   - Начиная с адреса $RcvLSN$ пройти до конца логического журнала и определить список зафиксированных транзакций $RedoTransList$. Поскольку контрольный снимок является консистентным (transaction-consistent), то после физического отката в базе данных нет изменений, выполненных неполными (незавершившимися) транзакциями. Поэтому необходимо определить только список транзакций, для которых необходимо выполнить операцию REDO.
   - Для каждой транзакции из списка $RedoTransList$ выполнить операцию REDO по логическому журналу. Для этого достаточно начиная с адреса $RcvLSN$ пройти в прямом направлении по всем записям логического журнала и если очередная запись сделана одной из транзакций из списка $RedoTransList$, то извлечь параметры соответствующей операции и выполнить операцию REDO для этой записи.

3. Создать контрольную точку базы данных.

Замечание 12. *При восстановлении базы данных микрооперации изменения данных необходимо перевести в специальный режим в котором они (1) не пишут записи в логический журнал, (2) отключается механизм версионности данных, и все изменения производятся в одной версии блока, (3) в физический журнал записи также не пишутся.*

Замечание 13. *В случае если, сбой происходит во время восстановления, новое восстановление начинается с прежней контрольной точки аналогичным образом. Отметим также, что при восстановлении блоки, составляющие контрольный снимок, не изменяются.*

5.8 Экспериментальная оценка методов управления XML-транзакциями

Для экспериментальной оценки предложенных методов управления XML-транзакциями мы их реализовали на языке C++ в прирожденной XML-СУБД Седна. При помощи экспериментов была измерена производительность Седны с использованием протокола 4VXDGL и стандартного двухфазного протокола синхронизации с гранулированностью захватов на уровне блоков.

Ниже мы будем рассматривать результаты экспериментов, направленные на измерение следующих характеристик прирожденной XML-СУБД:

- Пропускная способность XML-СУБД для читающих и изменяющих транзакций при наличии параллельных потоков транзакций на чтение и изменение XML-документов. Параметром экспериментов является селективность читающих транзакций.

- Время отклика для читающих и изменяющих транзакций при наличии параллельных потоков на чтение и изменение XML-документов. Параметром экспериментов является селективность читающих транзакций.

В качестве экспериментальной установки мы используем систему аукционов, рассмотренную в разделе 4.6.1. Эксперименты проводились над XML-СУБД Седна, установленной на компьютере Intel Pentium(R) 4 CPU 2.80ГГц, 512Мб ОЗУ с ОС MS Windows XP.

5.8.1 Эксперимент 1: пропускная способность

Первая группа экспериментов направлена на измерение пропускной способности XML-СУБД при наличии конкурентных потоков транзакций на чтение и изменение XML-документов. При этом мы используем 8 потоков транзакций. Поток транзакций состоит из читающих или изменяющих транзакций. Общее количество изменяющих транзакций составляет примерно 75%. Причем изменяющие транзакции являются небольшими и производят изменения от 5 до 10 элементов. Размер читающих транзакций является параметром и изменяется в сторону

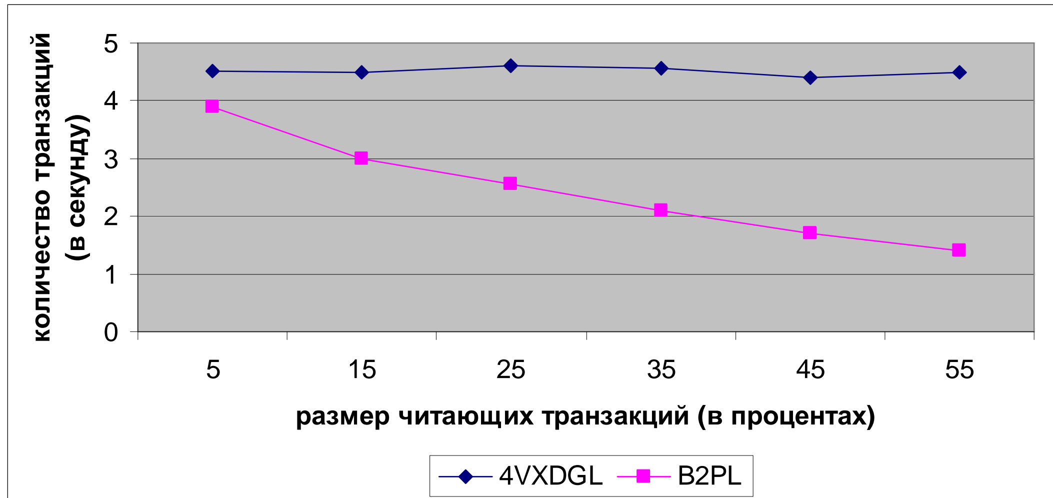


Рис. 5.7: Пропускная способность XML-СУБД для изменяющих транзакций

увеличения в процентах от общего размера документа. Размер XML-документа составляет примерно 10 Мб.

На рисунке 5.7 изображены результаты замеров для изменяющих транзакций. Кривые, помеченные ромбом и квадратом, изображают среднее количество выполненных изменяющих транзакций в секунду при использовании протоколов 4VXDGL и B2PL (двухфазный протокол с уровнем гранулированности захватов на уровне блоков) соответственно.

На графике видно, что с ростом размера читающих транзакций среднее количество выполненных изменяющих транзакций сильно уменьшается для протокола B2PL и остается примерно постоянным для 4VXDGL. Это объясняется двумя фактами. Во-первых, с ростом размера читающих транзакций для протокола B2PL резко возрастает количество конфликтов между короткими изменяющими транзакциями с читающими транзакциями. Фактически, "большие" читающие транзакции на длительное время блокируют выполнение коротких изменяющих транзакций. Во-вторых, бо̀льшее количество выполненных изменяющих транзакций в единицу времени для 4VXDGL по сравнению с B2PL при наличии читающих транзакций небольшого размера объясняется применением семантических XDGL-блокировок при изоляции изменяющих транзакций.

На рисунке 5.8 изображена пропускная способность XML-СУБД для читающих транзакций. В данном случае мы видим, что с ростом размера читающих транзакций среднее количество выполненных транзакций уменьшается как для протокола 4VXDGL, так и для протокола B2PL. Причем мы видим, что для читающих транзакций "большого" размера протокол B2PL немного превосходит протокол 4VXDGL. Это прежде всего объясняется тем, что в протоколе 4VXDGL появляются дополнительные издержки на один дополнительный

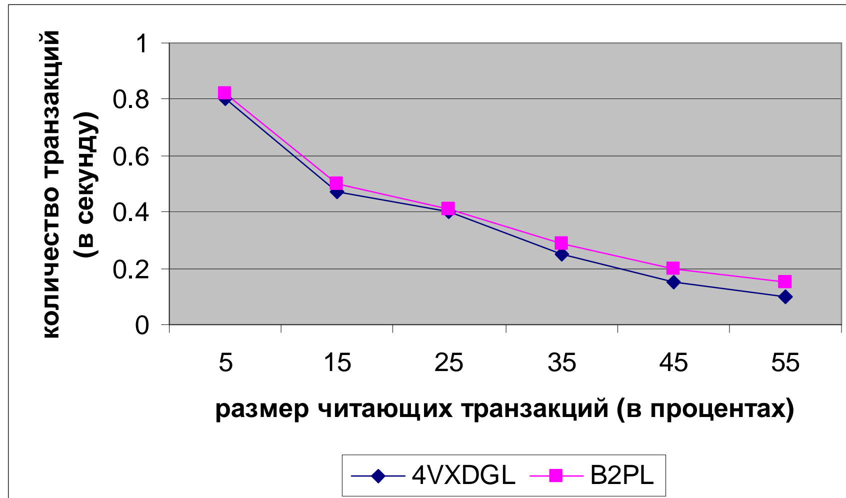


Рис. 5.8: Пропускная способность XML-СУБД для читающих транзакций

переход к старой версии. Причем эти издержки перекрывают издержки, связанные с конфликтами между изменяющими и читающими транзакциями в протоколе B2PL. Это объясняется тем, что если даже читающей транзакции и приходится ожидать конца выполнения изменяющей транзакции, то это занимает небольшой промежуток времени, поскольку изменяющие транзакции выполняют небольшие изменения.

Итак, наибольший выигрыш в пропускной способности (при использовании 4VXDGL) XML-СУБД получает для изменяющих транзакций, а для читающих транзакций демонстрируются примерно такие же временные характеристики, как и для двухфазного протокола. Заметного ухудшения временных характеристик для читающих транзакций не происходит поскольку в 4VXDGL используется максимум один косвенный переход к старой версии.

5.8.2 Эксперимент 2: время отклика

Вторая группа экспериментов направлена на измерение времени отклика транзакций в XML-СУБД при наличии конкурентных потоков транзакций на чтение и изменение XML-документов. При этом мы используем те же потоки транзакций, что и при проведении эксперимента 1.

На рисунке 5.9 изображены результаты замеров для изменяющих транзакций. Кривые, помеченные ромбом и квадратом, изображают среднее время отклика изменяющих транзакций при использовании протоколов 4VXDGL и B2PL соответственно.

На графике видно, что с ростом размера читающих транзакций время отклика изменяющих транзакций сильно увеличивается для протокола B2PL и остается примерно

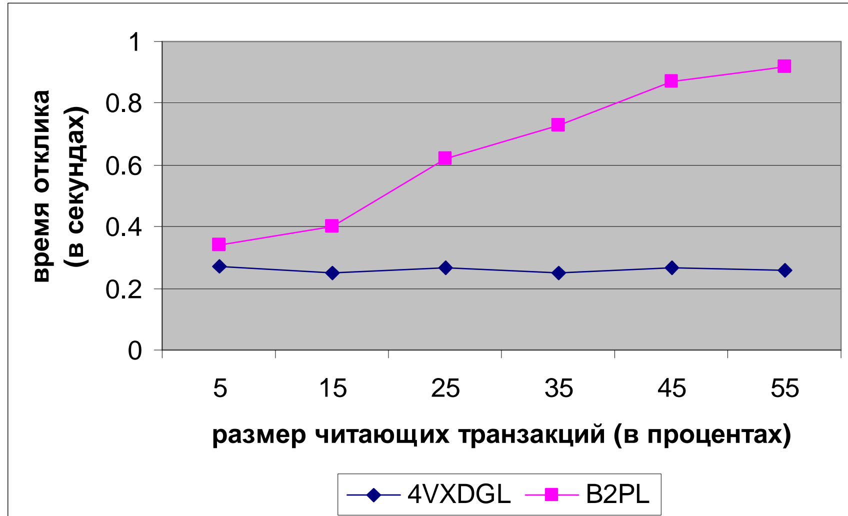


Рис. 5.9: Время отклика для изменяющих транзакций

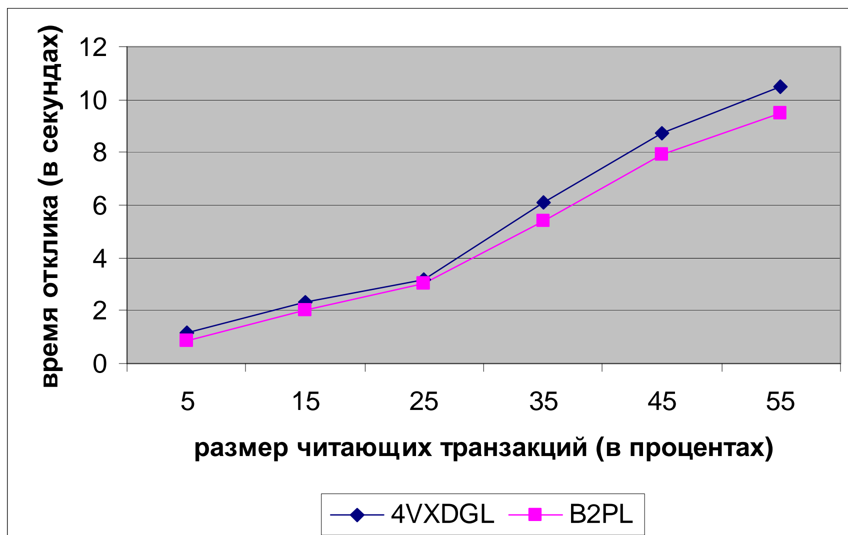


Рис. 5.10: Время отклика для читающих транзакций

постоянным для 4VXDGL. Это также объясняется тем, что читающие транзакции в 4VXDGL не устанавливают блокировки на прочитываемые данные. В протоколе B2PL, в свою очередь, время отклика изменяющих транзакций растет с ростом размера читающих транзакций поскольку они вовлекаются во все большее количество конфликтов и большую часть времени они ожидают блокировки.

На рисунке 5.10 изображены графики, обозначающие время отклика для читающих транзакций. Здесь мы наблюдаем совсем небольшое преимущество двухфазного протокола за счет того, что в нем чтение блока не требует косвенных переходов, в отличии от 4VXDGL протокола.

Основной вывод из экспериментов этой группы заключается в том, что протокол 4VXDGL существенно превосходит стандартный двухфазный протокол для изменяющих транзакций, а для читающих транзакций время отклика транзакций примерно одинаковое для обоих протоколов.

5.9 Выводы

В главе описан эффективный версионный метод управления транзакциями в прирожденных XML-СУБД. Основные свойства предложенного метода, необходимые для удовлетворения требований, изложенных в начале главы, доказываются в теоремах 2, 3, 4, 5. Кроме того, экспериментальная оценка также подтверждает эффективность предложенных методов. Особенно важно то, что версионный механизм не приводит к значительному понижению эффективности операции перехода по указателю, которая является базовой в прирожденных XML-СУБД.

Заключение

В диссертационной работе получены следующие результаты:

1. Разработан универсальный протокол XDGL изоляции XML-транзакций, который обеспечивает полную сериализацию XML-транзакций и учитывает семантику XML-модели данных при определении конфликтов между конкурентными XML-транзакциями.

2. На основе семантического протокола изоляции XML-транзакций разработан механизм управления конкурентными XML-транзакциями для реляционных СУБД с поддержкой XML.

3. Разработан версионный протокол 4VXDGL изоляции XML-транзакций для прирожденных XML-СУБД, в котором учитываются как семантика XML-модели данных, так и физические особенности организации структур внешней и оперативной памяти в прирожденных XML-СУБД. Протокол 4VXDGL также учитывает специфику использования прирожденных XML-СУБД, обеспечивая безконфликтное выполнение читающих и изменяющих транзакций.

4. Произведена экспериментальная оценка предложенных методов изоляции XML-транзакций для реляционной СУБД MS SQL Server 2005 и прирожденной XML-СУБД Седна, которая демонстрирует существенное увеличение производительности системы при наличии параллельных потоков транзакций на чтение и модификацию XML-документов.

5. Разработаны методы обеспечения атомарности и надежности транзакций в прирожденных XML-СУБД.

В дальнейшей работе автор намерен сконцентрироваться на решении следующих задач:

- Адаптация предложенных методов управления транзакциями в XML-СУБД к распределенным архитектурам. Актуальность этой задачи объясняется высокой

привлекательностью XML как средства интеграции разнородных и распределенных источников данных в глобальной сети [97, 98, 99, 100].

- Расширение предложенного версионного протокола 4VXDGL поддержкой W|R транзакций [96], когда на первой W-фазе транзакция производит модификацию БД и устанавливает блокировки на изменяемые данных, а на второй R-фазе производит только чтение данных (возможно старых версий) без установки каких-либо блокировок. В частности, поддержка W|R транзакций позволит выполнять проверки ограничений целостности БД (ограничения по ключам, валидация на основе предписывающей схемы и т.д.) на R-фазе без установки блокировок, что может существенно может повысить параллелизм транзакций в XML-СУБД.

Список литературы

[1] P. Pleshachkov, L. Novak. Transaction Isolation In the Sedna Native XML DBMS. Proc. SYRCoDIS 2004, Saint-Petersburg, Russia.

[2] P. Pleshachkov, P. Chardin, S. Kuznetsov. A DataGuide-Based Concurrency Control Protocol for Cooperation on XML Data. Proc. ADBIS 2005, LNCS 3631 Springer 2005, Tallinn, Estonia.

[3] P. Pleshachkov, P. Chardin, S. Kuznetsov. XDGL: XPath-Based Concurrency Control Protocol for XML Data. Proc. BNCOD 2005, LNCS 3567 Springer 2005, Sunderland, UK.

[4] P. Pleshachkov, P. Chardin, S. Kuznetsov. A Locking Based Scheduler for XML Databases. Proc. SEBD 2005, ISBN 8-548-0122-4, Brixen-Bressanone, Italy.

[5] P. Pleshachkov, P. Chardin. A Locking Protocol for Scheduling Transactions on XML Data. Proc. SYRCoDIS 2005, Saint-Petersburg, Russia.

[6] P. Pleshachkov. Transaction Management for XML Stored in Relational Database Systems. Proc. PhD Forum BNCOD 2006, Belfast, Northern Ireland, UK.

[7] П. О. Плешачков, С. Д. Кузнецов. Управление транзакциями в РСУБД с поддержкой XML. Программирование. – М.: Наука, 2006. – N 5.

[8] Плешачков П. О. SXTM: высокопроизводительный менеджер управления XML-транзакциями. Препринт 11 Института Системного Программирования РАН, 2006.

[9] Weikum G., Vossen G. Transactional Information Systems Morgan Kauffmann, 2002

[10] J. Gray, A. Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufmann, 1993.

[11] P.A. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency Control and Recovery in Database Systems. Addison-Wesley, 1987.

[12] G. Weikum, H. J. Schek. Database Transaction Models for Advanced Applications. Morgan Kaufmann, 1992.

[13] П. Чардин. Исследование и разработка методов управления параллельными транзакциями над XML-данными с использованием синхронизационных блокировок и версий. Дипломная работа, ВМиК МГУ, Москва 2006.

[14] The World Wide Web Consortium (W3C). Extensible Markup Language (XML) 1.0 (Third Edition) W3C Recommendation. Jean Paoli, C. M. Sperberg-McQueen et al. 04 Feb. 2004.

[15] ISO 8879. Information Processing - Text and Office Systems - Standard Generalized Markup Language (SGML). 1986.

[16] The World Wide Web Consortium (W3C). HTML 4.0 Specification. W3C Recommendation. D. Raggett, A. Le Hors, I. Jacobs. revised on 24-Apr-1998. http://www.w3.org/TR/1998/REC-html40-19980424/

[17] The World Wide Web Consortium (W3C). XML Schema Part 1: Structures : W3C recommendation. Biron P.V., Malhotra A. (eds.). – 2nd edition. 2004, 28 Oct. http://www.w3.org/TR/xmlschema-1/

[18] The World Wide Web Consortium (W3C). XML Schema Part 2: Datatypes : W3C recommendation. Biron P.V., Malhotra A. (eds.). – 2nd edition. 2004, 28 Oct. http://www.w3.org/TR/xmlschema-2/

[19] The World Wide Web Consortium (W3C). XQuery 1.0 and XPath 2.0 Data Model: W3C Candidate Recommendation. M. Fernandez et al. (eds.). – 2003, 3 Nov. http://www.w3.org/TR/xpath-datamodel/

[20] The World Wide Web Consortium (W3C). XML Path Language (XPath) 2.0: W3C Candidate Recommendation. D. Chamberlin, M. F. Fernandez, M. Kay et al. (eds.). 2006, 8 June. http://www.w3.org/TR/xpath20/

[21] The World Wide Web Consortium (W3C). XQuery 1.0: An XML Query Language : W3C working draft. Boag S. et al. (eds.). – 2004, 29 Oct. http://www.w3.org/TR/2004/WD-xquery-20041029/

[22] The World Wide Web Consortium (W3C). XQuery 1.0 and XPath 2.0 Functions and Operators: W3C Candidate Recommendation. J. Melton et al. (eds.), 2006, 8 June. http://www.w3.org/TR/xpath-functions/

[23] The World Wide Web Consortium (W3C). XSL Transformations (XSLT) Version 2.0. W3C Candidate Recommendation. M. Kay. (eds.), 2006, 8 June. http://www.w3.org/TR/xslt20/

[24] The World Wide Web Consortium (W3C). Document Object Model (DOM) Level 2 Core Specification: W3C Candidate Recommendation. A. Le Hors et al. (eds.). – 2000, 13 Nov.

[25] Conrad A. MSDN Library. http://msdn.microsoft.com/library/en-us/dnexxml/html/xml07162001.asp. Jul 2001.

[26] Klein S. Interactive Microsoft SQL Server and XML online tutorial. http://www.topxml.com/tutorials/main.asp?id=sqlxml.

[27] Rys M. Bringing the Internet to Your Database: Using SQL Server 2000 and XML to Build Loosely-Coupled Systems. Proc. 17th ICDE Conference, Heidelberg, Germany, 2-6 April., 2001.

[28] S. Pal, M. Fussell, and Irwin Dolobowsky. XML Support in Microsoft SQL Server 2005. Microsoft Corporation, Dec. 2005. http://msdn.microsoft.com/xml/default.aspx?pull=/library/en-us/dnsql90/html/sql2k5xml.asp

[29] Eisenberg A., Melton J. SQL/XML is making good progress. SIGMOD Record Vol. 31, N. 2 (2002).

[30] Oracle Corporation. Concurrency Control, Transaction Isolation, and Serializability in SQL92 and Oracle7. Technical report, Oracle Corporation, 1995. Part No. A33745.

[31] Cheng J., Xu J. XML and DB2. Proc. 16th ICDE Conference, San Diego, USA, 28 Feb. 3 March, 2000.

[32] K. Beyer et al. DB2 goes hybrid: Integrating XML and XQuery with relational data and SQL. IBM Systems Journal, Vol. 45, N. 2, 2006.

[33] SQL Server 7.0, SQL Server 2000, and SQL Server 2005 logging and data storage algorithms extend data reliability Microsoft Corp., Article Num. 230785, Feb. 9, 2006 http://support.microsoft.com/default.aspx?scid=kb;en-us;230785

[34] S. Pal, I. Cseri, G. Schaller, O. Seeliger, L. Giakoumakis, V. Zolotov. Indexing XML Data Stored in a Relational Database. Proc. VLDB Conference 2004, Toronto, Canada.

[35] Z. H. Liu, M. Krishnaprasad, V. Arora. Native Xquery processing in oracle XMLDB. Proc. ACM SIGMOD Conference 2005, Baltimore, USA.

[36] Mastering XML DB Storage in Oracle Database 10g Release 2. http://www.oracle.com/technology/tech/xml/xmldb/index.html

[37] Corp. Extensible Information Server. Technical report, Excelon Corp., 2001. http://www.exceloncorp.com/platform/extinfserver.

[38] Microsoft Developers Network. http://msdn.microsoft.com

[39] Linux Documentation (including Manual Pages). http://www.linux.org/docs/index.html

[40] M. H. Kay. SAXON The XSLT and XQuery Processor. http://saxon.sourceforge.net/

[41] Д. Чемберлин. XQuery: язык запросов XML. Открытые системы, 2003. N. 1.

[42] Буре Р. XML и базы данных. Открытые системы, N. 10, 2000.

[43] С. Кузнецов, П. Чардин. Семейство алгоритмов ARIES. Открытые системы, N. 3, 2004.

[44] A. Silberschatz and Z. Kedem. Consistency in hierarchical database systems. Journal of the ACM, 27(1): 72–80, 1980.

[45] C. Mohan, D. Haderie, B. Lindsay, H. Pirahesh, P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. ACM Transactions on Database Systems, Voi. 17, N. 1, March 1992.

[46] C. Mohan, F. Levine. ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging. Proc. SIGMOD Conference 1992, San Diego, California, USA.

[47] C. Mohan. ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes. Proc. VLDB Conference 1990, Brisbane, Queensland, Australia.

[48] T. Grabs, K. Bohm and H.-J. Schek. XMLTM: efficient transaction management for XML documents. Proc. CIKM Conference 2002, McLean, Virginia, USA.

[49] G. Alonso, S. Blott, A. Fessler, H.J. Schek. Correctness and parallelism in composite systems. Proc. PODS Symposium, 1997, Tucson, Arizona, USA.

[50] D. Florescu, A. Y. Levy, and D. Suciu. Query Containment for Conjunct Queries with Regular Expressions. Proc. PODS Symposium, 1998, Seattle, Washington, USA.

[51] J. Shanmugasundaram, K. Tufte, et al. Relational Databases for Querying XML Documents: Limitations and Opportunities. Proc. VLDB Conference, 1999, Edinburgh, Scotland.

[52] A. Deutsch, M. F. Fernandez, et al. Storing Semi-structured Data with STORED. Proc. SIGMOD Conference, 1999, Philadelphia, Pennsylvania, USA.

[53] F. Tian, D. J. DeWitt, J. Chen, C. Zhang. The Design and Performance Evaluation of Alternative XML Storage Strategies. SIGMOD Record Volume 31 N. 1, 2002.

[54] D. Suciu. Semistructured Data and XML. Kluwer Academic Publishers, 2000.

[55] P. Buneman. Semistructured Data. In Proc. PODS Symposium 1997, Arizona, USA.

[56] I. Tatarinov, Z. Ives, A. Halevy, D. Weld. Updating XML. Proc. SIGMOD Conference 2001, Santa Barbara, California, USA.

[57] I. Tatarinov, S. Viglas, K. S. Beyer, J. Shanmugasundaram, et al. Storing and querying ordered XML using a relational database system. Proc. SIGMOD Conference 2002, Madison, Wisconsin, USA.

[58] Gray J., Lorie R., Putzolu G. Granularity of locks and degrees of consistency in a shared data base. Proc. VLDB Conference, 1975.

[59] M. Nicola, B. V. Linden. Native XML Support in DB2 Universal Database. Proc. VLDB Conference 2005, Trondheim, Norway.

[60] M. Rys, D. Chamberlin, D. Florescu. XML and relational database management systems: the inside story. Proc. SIGMOD Conference 2005, Baltimore, Maryland, USA.

[61] Фомичев А. В. Исследование и разработка методов организации выполнения и физической оптимизации запросов к XML-данным. Диссертация на соискание ученой степени кандидата физико-математических наук, ВМиК МГУ, Москва, 2006.

[62] A. Silberschatz, H. Korth, S. Sudarshan. Database System Concepts. Third Edition, McGraw-Hill, 1997.

[63] Chan A., S. Fox, W. Lin, A. Nori, Ries D. The Implementation of an Integrated Concurrency Control and Recovery Scheme. Proc. SIGMOD Conference, 1982, Orlando, Florida, USA.

[64] Chan, A., and R. Gray. Implementing Distributed Read-Only Transactions. IEEE Trans. on Software Eng., SE-11(2), Feb 1985.

[65] Raghavan, A., and Rengarajan, T.K. Database Availability for Transaction Processing. Digital Technical Journal 3(1), Winter 1991.

[66] Bober, P. and M. Carey. On Mixing Queries and Transactions via Multiversion Locking. Proc. ICDE Conference, 1992, Tempe, Arizona, USA.

[67] C. Mohan, H. Pirahesh, R. A. Lorie. Efficient and Flexible Methods for Transient Versioning of Records to Avoid Locking by Read-Only Transactions. Proc. SIGMOD Conference 1992, San Diego, California, USA.

[68] R. Bayer, H. Heller, and A. Reiser. Parallelism and Recovery in Database Systems. ACM Trans. Database Syst. 5(2): 139-156 (1980).

[69] K.-L. Wu, P. S. Yu, M.-S. Chen. Dynamic Finite Versioning: An Effective Versioning Approach to Concurrent Transaction and Query Processing. Proc. ICDE Conference, 1993, Vienna, Austria.

[70] Pirahesh, H., et al. Parallelism in Relational Database Systems: Architectural Issues and Design Approaches. IEEE 2nd International Symposium on Databases in Parallel and Distributed Systems, Dublin, Ireland, July 1990.

[71] M. P. Haustein, Theo Harder Adjustable Transaction Isolation in XML Database Management Systems. Proc. XSym Workshop 2004, Toronto, Canada.

[72] M. P. Haustein, Theo Harder. taDOM: A Tailored Synchronization Concept with Tunable Lock Granularity for the DOM API. Proc. ADBIS 2003, Dresden, Germany.

[73] S. Helmer, C.C. Kanne, G. Moerkotte. Lock-based protocols for cooperation on XML documents. Proc. DEXA 2003, Prague, 2003.

[74] M. Haustein, T Harder, K. Luttenberger. Contest of XML Lock Protocols. Proc. VLDB Conference 2006, Seoul, Korea.

[75] S. Dekeyser, J. Hidders. Path Locks for XML Document Collaboration. Proc. WISE 2002, Singapore.

[76] A. Fomichev, M. Grinev, S. Kuznetsov. Sedna: A Native XML DBMS. Proc. SOFSEM 2006, LNCS 3831, Merin, Czech Republic.

[77] Benchmark of Sedna XML DBMS. MODIS Group. http://modis.ispras.ru/Development/Benchmarks, ISP RAS, 2006.

[78] T. Fiebig, S. Helmer, C.-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, T. Westmann. Anatomy of a native XML base management system. VLDB Journal Vol. 11, Num. 26 (2002).

[79] Software AG. Tamino - The XML Power Database. Technical report, Software AG, 2001. http://www.softwareag.com/tamino/.

[80] H. Jagadish et al. TIMBER: A native system for quering XML. Proc. SIGMOD Conference, 2003, San Diego, USA.

[81] W. Meier. eXist: An Open Source Native XML Database. Proc. Web, Web-Services, and Database Systems 2002, Erfurt, Germany.

[82] X-Hive Corporation, X-HIVE/DB. http://www.x-hive.com/products/db/index.html

[83] X. Meng, D. Luo, M. Lee, J. OrientStore: A Schema Based Native XML Storage System. Proc. VLDB 2003 Conference, Berlin, Germany.

[84] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, J. Widom. Lore: A Database Management System for Semistructured Data. SIGMOD Record Vol. 26, N. 3 (1997).

[85] R. Goldman, J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. Proc. VLDB Conference 1997, Greece, Athens.

[86] K. P. Eswaran, J. Gray, R. A. Lorie, I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. Commun. ACM Vol. 19, N. 11, 1976.

[87] N. A. Aznauryan, S. D. Kuznetsov, L. G. Novak, and M. N. Grinev. SLS: A Numbering Scheme for Large XML Documents. Programming and Computer Software, N. 1, Vol. 32, 2006.

[88] Гринев М.Н., Кузнецов С.Д., Фомичев А.В. XML-СУБД Sedna: технические особенности и варианты использования. Открытые системы N. 8, 2004. http://www.osp.ru/os/2004/08/036.htm

[89] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, I. Traiger. The Recovery Manager of the System R Database Manager. ACM Computing Surveys, Vol. 13, N. 2, June 1981.

[90] H. Hunt, D. Rosenkrantz. The Complexity of Testing Predicate Locks. Proc. ACM SIGMOD 1979, Boston, Massachusetts, USA.

[91] D. Rosenkrantz, H. Hunt. Processing Conjunctive Predicates and Queries. Proc. VLDB Conference 1980, Montreal, Canada.

[92] G. Weikum. Principles and Realization Strategies of Multilevel Transaction Management. ACM Transactions on Database Systems (TODS), Vol. 16, issue 1, 1991.

[93] Лисовский К.Ю. Разработка XML-приложений на языке Scheme. Программирование. Вып. 28 N.1, 2002.

[94] A. Schmidt, F. Waas, M. Kersten, D. Florescu, I. Manolescu, M. Carey, R. Busse. The XML Benchmark Project. Technical Report INS-R0103, CWI, Amsterdam, The Netherlands, April 2001.

[95] D. Florescu and D. Kossmann. Storing and Querying XML Data using an RDMBS. IEEE Data Engineering Bulletin, Vol. 22 N. 3, 1999.

[96] F. Llirbat, E. Simon, D. Tombroff. Using Versions in Update Transactions: Application to Integrity Checking. Proc. VLDB Conference 1997, Athens, Greece.

[97] I. Manolescu, D. Florescu, D. Kossmann. Answering XML Queries on Heterogeneous Data Sources. Proc. VLDB Conference 2001, Roma, Italy.

[98] S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo, and R. Weber. Active XML: Peer-to-Peer Data and Web Services Integration. Proc. VLDB Conference 2002, Hong Kong, China.

[99] S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, T. Milo. Dynamic XML documents with distribution and replication. Proc. SIGMOD Conference 2003, San Diego, California, USA.

[100] A. Bonifati, E. Q. Chang, T. Ho, L. V. S. Lakshmanan, R. Pottinger. HePToX: Marrying XML and Heterogeneity in Your P2P Databases. Proc. VLDB Conference 2005, Trondheim, Norway.