СПбГЭТУ «ЛЭТИ»
ПЕРВЫЙ ЭЛЕКТРОТЕХНИЧЕСКИЙ

Татарникова Татьяна Михайловна

Управление большими данными

Конспект лекций

СПбГЭТУ «ЛЭТИ», 2022 г.

ТЕМА 2. УПРАВЛЕНИЕ ДОСТУПОМ К БАЗАМ ДАННЫХ

База данных – это возможность решения многих задач несколькими пользователями.

Поэтому основная характеристика современных СУБД – наличие многопользовательской технологии работы.

2.1. Архитектурные решения

Рассмотрим основные архитектурные решения, которые реализуют многопользовательский доступ. Известны три архитектуры СУБД, которые приходили на смену друг другу с развитием технологий баз данных. Это централизованная, файл-серверная и клиент-серверная архитектура.

Исторически первой архитектурой считается **централизованная, в которой** база данных, СУБД и прикладная программа (приложение) располагаются на одном компьютере. Сегодня такая архитектура изжила себя, но с точки зрения того, что все на одном компьютере можно сказать, что на новом витке развития ИТ эта архитектура переродилась во встраиваемую: СУБД тесно связана с приложением и работает на том же компьютере, но не требуя профессионального администрирования и не требуется доступ с многих компьютеров.

На «рабочем столе» практически каждого персонального компьютера или смартфона есть программы, в которых может быть встраиваемая СУБД: почтовые клиенты и мессенджеры, телефонные справочники, заметки и т.п., которые хранят данные владельца гаджета. Централизованная или встраиваемая архитектура приведена на рисунке 2.1.

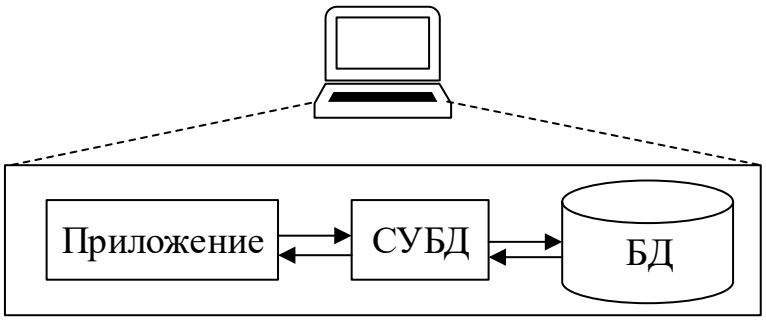


Рисунок 2.1 – Централизованная (встраиваемая) архитектура

Работа централизованной (встраиваемой) архитектуры построена следующим образом:

– БД в виде набора файлов находится во внешней памяти компьютера. На том же компьютере установлены СУБД и приложение для работы с БД.

– Пользователь запускает приложение, инициируя обращение к БД на выборку/обновление информации.

– Все обращения к БД идут через СУБД, которая инкапсулирует внутри себя все сведения о физической структуре БД.

– СУБД инициирует обращения к данным, обеспечивая выполнение запросов пользователя.

– Результат СУБД возвращает в приложение.

– Приложение, используя пользовательский интерфейс, отображает результат выполнения запросов.

Как становится очевидно из принципа работы централизованной архитектуры, многопользовательская технология работы обеспечивается:

а) режимом мультипрограммирования, то есть одновременно могут работать процессор и внешние устройства,

б) режимом разделения времени, то есть пользователям по очереди выделялись кванты времени на выполнении их программ.

Таким образом, основной недостаток этой модели – резкое снижение производительности при увеличении числа пользователей.

«Файл-сервер» – это архитектура при которой БД находится на сервере, СУБД на клиентском компьютере, доступ СУБД к данным осуществляется через локальную сеть, то есть это архитектура с сетевым доступом.

Файл-сервер – это машина, которая отвечает на запрос. Как правило это один из компьютеров сети – выделенный сервер (файлы базы данных)

Рабочая станция – это запрашивающая машина (обычно персональный компьютер).

Синхронизация чтений и обновлений осуществляется посредством файловых блокировок.

При запросе файлы с сервера передаются на рабочие станции пользователей, где осуществляется основная часть обработки данных. Сервер выполняет роль хранилища файлов, который в обработке данных не участвует.

Архитектура «файл-сервер» приведена на рисунке 2.2.

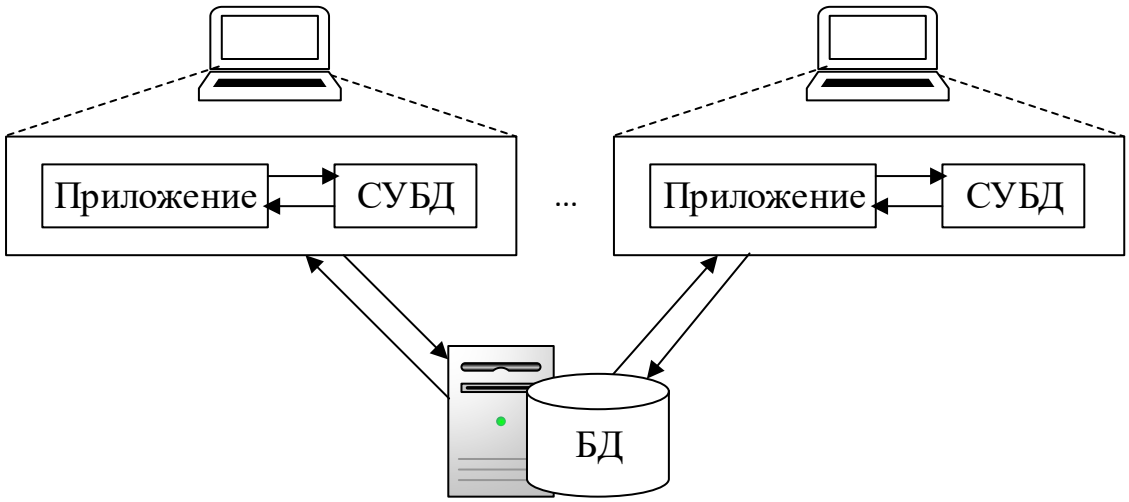


Рисунок 2.2 – Архитектура «файл-сервер»

Работа архитектуры «файл-сервер» построена следующим образом:

– БД в виде набора файлов находится на файловом сервере.
– Локальная сеть состоит из рабочих станций, с установленными СУБД и приложением для работы с БД.
– На каждой из рабочих станций инициируются обращения к БД.
– Все обращения к БД идут через СУБД.
– Часть файлов БД копируется на рабочую станцию и обрабатывается.
– Результат обработки СУБД возвращает в приложение.
– Приложение, используя пользовательский интерфейс, отображает результат выполнения запросов.

Преимущество архитектуры «Файл-сервер»:

– низкая нагрузка на процессор файлового сервера.

Недостатки архитектуры «Файл-сервер»:

– потенциально высокая загрузка локальной сети;
– невозможность централизованного управления;
– обеспечения высокой надежности, доступности и безопасности.

СУБД с архитектурой «Файл-сервер» применялись в локальных приложениях с функциями управления БД.

Технология считается устаревшей, а ее использование в крупных информационных системах недостатком.

Клиент-сервер – это сетевая архитектура, в которой узлы являются либо клиентами, либо серверами.

Клиент – это запрашивающая машина (обычно персональный компьютер).

Сервер – это машина, которая отвечает на запрос.

Клиент и сервер – это физические устройства/программное обеспечение (аппаратный или программный компонент вычислительной системы).

Взаимодействие клиента и сервера осуществляется по определенному протоколу.

Программа-клиент может запрашивать с сервера какие-либо данные, манипулировать данными непосредственно на сервере, запускать на сервере новые процессы и т. п.

Архитектура «клиент-сервер» разделяет функции приложения пользователя, называемого клиентом и сервера. Схема архитектуры «клиент-сервер» приведена на рисунке 2.3.

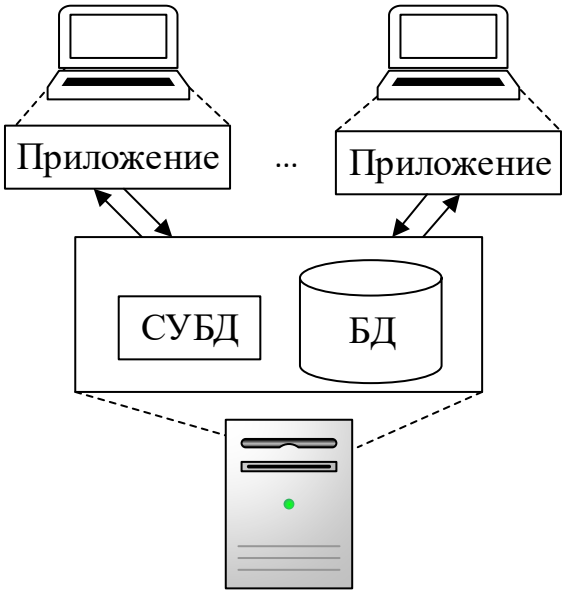


Рисунок 2.3 – Архитектура «клиент-сервер»

Работа построена следующим образом:

– База данных в виде набора файлов находится на жестком диске специально выделенного компьютера (сервера сети).

– СУБД располагается также на сервере сети.

– Существует локальная сеть, состоящая из клиентских компьютеров, на каждом из которых установлено клиентское приложение для работы с БД.

– На каждом из клиентских компьютеров пользователи имеют возможность запустить приложение (инициирует запрос к БД). По сети от клиента к серверу передается лишь текст запроса SQL.

– СУБД инкапсулирует внутри себя все сведения о физической структуре БД, расположенной на сервере.

– СУБД инициирует обращения к данным, находящимся на сервере, в результате которых на сервере осуществляется вся обработка данных и лишь результат выполнения запроса копируется на клиентский компьютер. Таким образом СУБД возвращает результат в приложение.

– Приложение, используя пользовательский интерфейс, отображает результат выполнения запросов.

В функции приложения-клиента входит:

– Посылка запросов серверу.

– Интерпретация результатов запросов, полученных от сервера.

– Представление результатов пользователю в некоторой форме (интерфейс пользователя).

В функции серверной части входит:

– Прием запросов от приложений-клиентов.

– Интерпретация запросов.

– Оптимизация и выполнение запросов к БД.

– Отправка результатов приложению-клиенту.

– Обеспечение системы безопасности и разграничение доступа.

– Управление целостностью БД.

– Реализация стабильности многопользовательского режима работы.

Архитектура «клиент-сервер» обладает следующими достоинствами:

– Существенно уменьшается сетевой трафик.

– Уменьшается сложность клиентских приложений (большая часть нагрузки ложится на серверную часть), а, следовательно, снижаются требования к аппаратным мощностям клиентских компьютеров.

– Наличие специального программного средства – SQL-сервера – приводит к тому, что существенная часть проектных и программистских задач становится уже решенной.

– Существенно повышается целостность и безопасность БД.

Архитектура «клиент-сервер» обладает следующими недостатками:

– более высокие финансовые затраты на аппаратное и программное обеспечение,

– трудности со своевременным обновлением клиентских приложений на всех компьютерах-клиентах.

Схема многозвенной (трехзвенной) архитектуры «Клиент-сервер» приведена на рисунке 2.4.

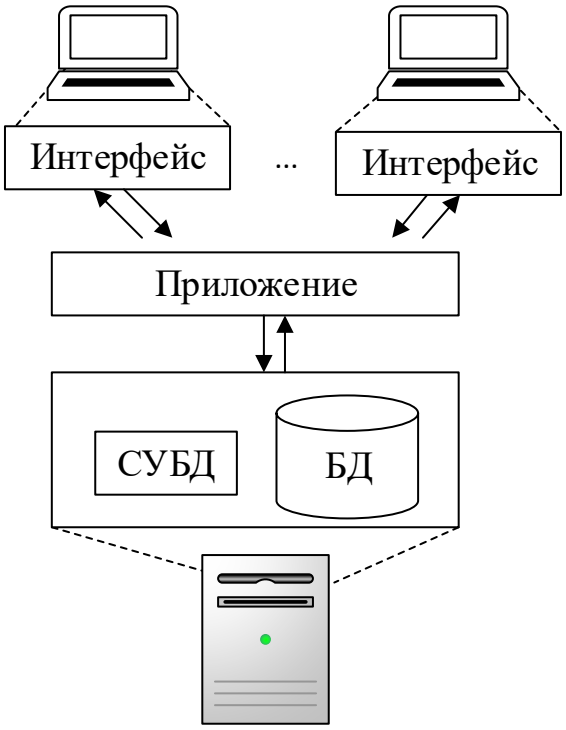


Рисунок 2.4 – Трехзвенная архитектура

Работа трехзвенной архитектуры построена следующим образом:

– База данных в виде набора файлов находится на сервере.

– СУБД располагается также на сервере сети.

– Существует специально выделенный сервер приложений, на котором располагается программное обеспечение (ПО) делового анализа (бизнес-логика).

– Существует множество клиентских компьютеров, на каждом из которых установлен так называемый «тонкий клиент» – клиентское приложение, реализующее интерфейс пользователя.

– На каждом из клиентских компьютеров может быть запущено приложение – тонкий клиент. Оно инициирует обращение к ПО делового анализа, расположенному на сервере приложений.

– Сервер приложений анализирует требования пользователя и формирует запросы к БД. По сети от сервера приложений к серверу БД передается лишь текст запроса на языке SQL.

– СУБД инкапсулирует внутри себя все сведения о физической структуре БД, расположенной на сервере.

– СУБД инициирует обращения к данным, находящимся на сервере, в результате которых результат выполнения запроса копируется на сервер приложений.

– Сервер приложений возвращает результат в клиентское приложение (пользователю).

– Приложение, используя пользовательский интерфейс, отображает результат выполнения запросов.

Достоинства трехзвенной архитектуры заключаются в следующем:

– Уменьшается объем передаваемого сетевого трафика между «Клиентом» и «Сервером».

– Уменьшается сложность клиентских приложений (большая часть нагрузки ложится на серверную часть).

– Наличие SQL-сервера приводит к тому, что существенная часть задач уже решена.

– Существенно повышается целостность и безопасность БД.

– При изменении бизнес-логики более нет необходимости изменять клиентские приложения и обновлять их у всех пользователей.

2.2. Технология «клиент-сервер»

Основным достоинством архитектуры «клиент-сервер» является возможность распределения БД по нескольким серверам и параллельной обработки запросов благодаря возможности обращения к разным серверам.

БД физически распределяется по узлам данных на основе фрагментации и репликации данных.

При наличии схемы реляционной базы данных каждое отношение может фрагментироваться на горизонтальные или вертикальные разделы. **Горизонтальная фрагментация** реализуется при помощи операции селекции, которая направляет каждый кортеж отношения в один из разделов, руководствуясь предикатом фрагментации. При **вертикальной фрагментации** отношение делится на разделы при помощи операции проекции. За счет фрагментации данные приближаются к месту их наиболее интенсивного использования, что потенциально снижает затраты на пересылки; уменьшаются также размеры отношений, участвующих в пользовательских запросах.

При отсутствии схемы реляционной базы данных, то есть БД NoSQL – фрагментация выполняется по агрегатам – горизонтальная фрагментация и по принципу «ведущий-ведомый» для вертикальной фрагментации.

Высокая производительность – одна из важнейших целей, на достижение которой направлены технологии параллельных СУБД

Возможность параллельной обработки запросов реализуется СУБД.

Известны три вида параллелизма, присущие обработке данных.

Межзапросный параллелизм предполагает одновременное выполнение множества запросов, относящихся к разным транзакциям.

Под **внутризапросным параллелизмом** понимается одновременное выполнение сразу нескольких операций, относящихся к одному и тому же запросу.

И внутризапросный, и межзапросный параллелизм реализуется на основе разделения данных, аналогичного горизонтальному фрагментированию.

Наконец, понятие **внутриоперационного параллелизма** означает параллельное выполнение одной операции в виде набора субопераций с применением, в дополнение к фрагментации данных, также и фрагментации функций.

На рисунке 2.5 приведены все перечисленные виды параллелизма.

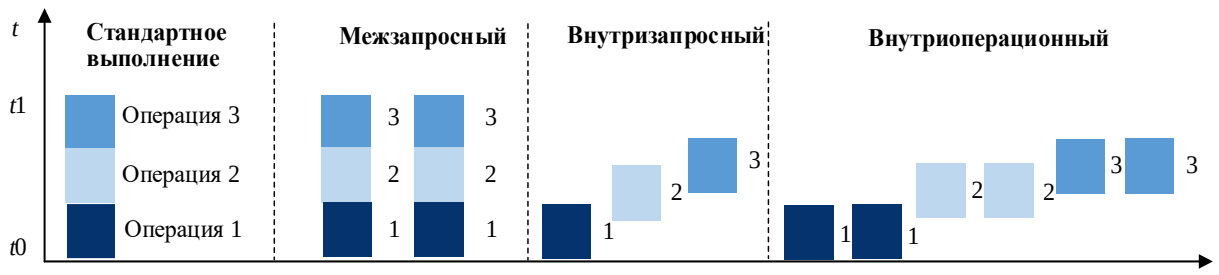
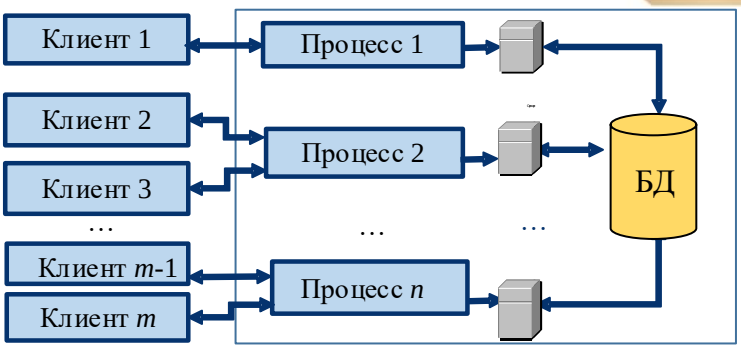


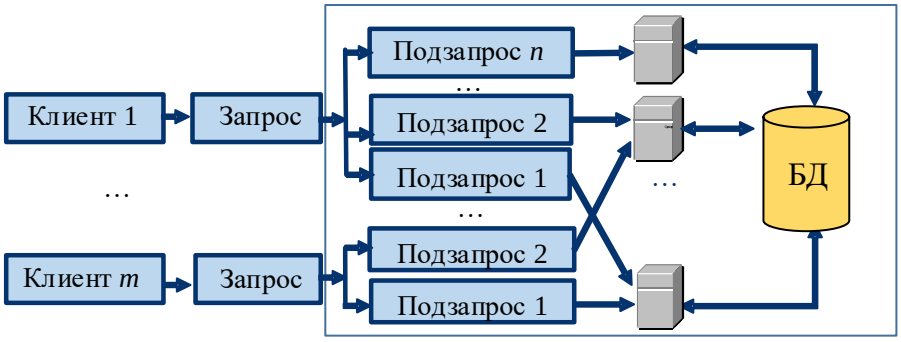
Рисунок 2.5 – Виды параллелизма обработки данных

Распределенность и параллельная работа на серверах повышает сложность базы данных, поэтому при выборе модели параллелизма взвешиваются все аргументы.

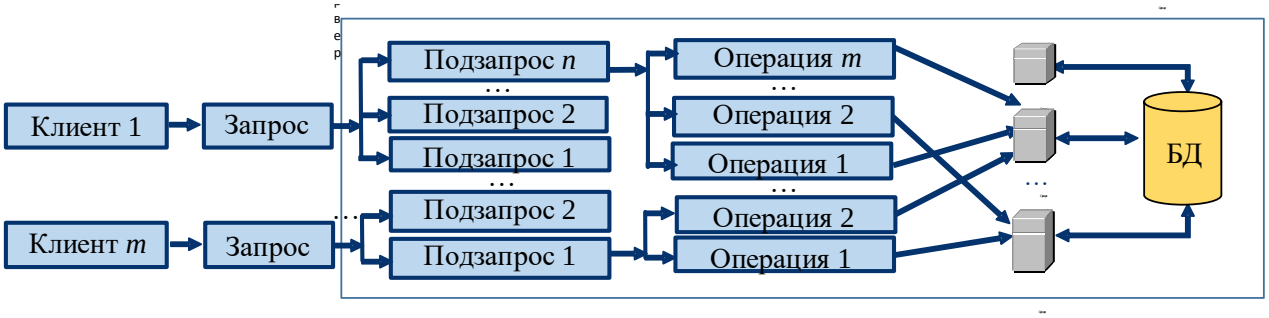
На рисунке 2.6 приведены структурные схемы, которые объясняют разницу между видами организации параллельной работы СУБД.



а – межзапросный параллелизм



б – внутризапросный параллелизм



в – внутриоперационный параллелизм

Рисунок 2.6 – Структурные схемы параллельной работы СУБД

В распределенных базах данных с поддержкой репликации каждому логическому элементу данных соответствует несколько физических копий. В такого рода системах возникает проблема поддержкой согласованности копий. Наиболее известным критерием согласованности является критерий **полной эквивалентности копий**, который требует, чтобы по завершении окна несогласованности все копии логического элемента данных были идентичны.

Протокол управления репликами отвечает за отображение операций над x в операции над физическими копиями x (x_1, x_2 ,..., x_n).

Известный (типичный) протокол управления репликами, следующий критерию полной эквивалентности копий, известен под названием ROWA (**Read-Once/Write-All** – чтение из одной копии, запись во все копии).

Каждая операция записи в логический элемент данных *x* отображается на множество операций записи во все физические копии *x*.

Протокол ROWA прост и прямолинеен, но он требует доступности всех копий элемента данных, чтобы завершить транзакцию. Сбой на одном из узлов приведет к блокированию транзакции, что снижает доступность базы данных.

Поэтому распространение получили протоколы, основанные на механизме голосования на основе кворума, которые рассматривались в БД NoSQL.

Распределенная обработка – это архитектура клиент-сервер, где БД, СУБД, приложения размещены на нескольких серверах. Под технологией «клиент-сервер» также подразумевают облачные вычисления (cloud computing), когда хотят подчеркнуть прозрачность технологии.

Облачные вычисления – это модель обеспечения удобного сетевого доступа по требованию к некоторому общему фонду конфигурируемых вычислительных ресурсов, например к сетям передачи данных, серверам, системам хранения данных, приложениям и сервисам – как вместе, так и по отдельности.

«Облако» строится на основе центров обработки данных (ЦОД). Структура ЦОД состоит из серверов и систем хранения данных, объединенных локальной сетью (см. рис. 2.7).

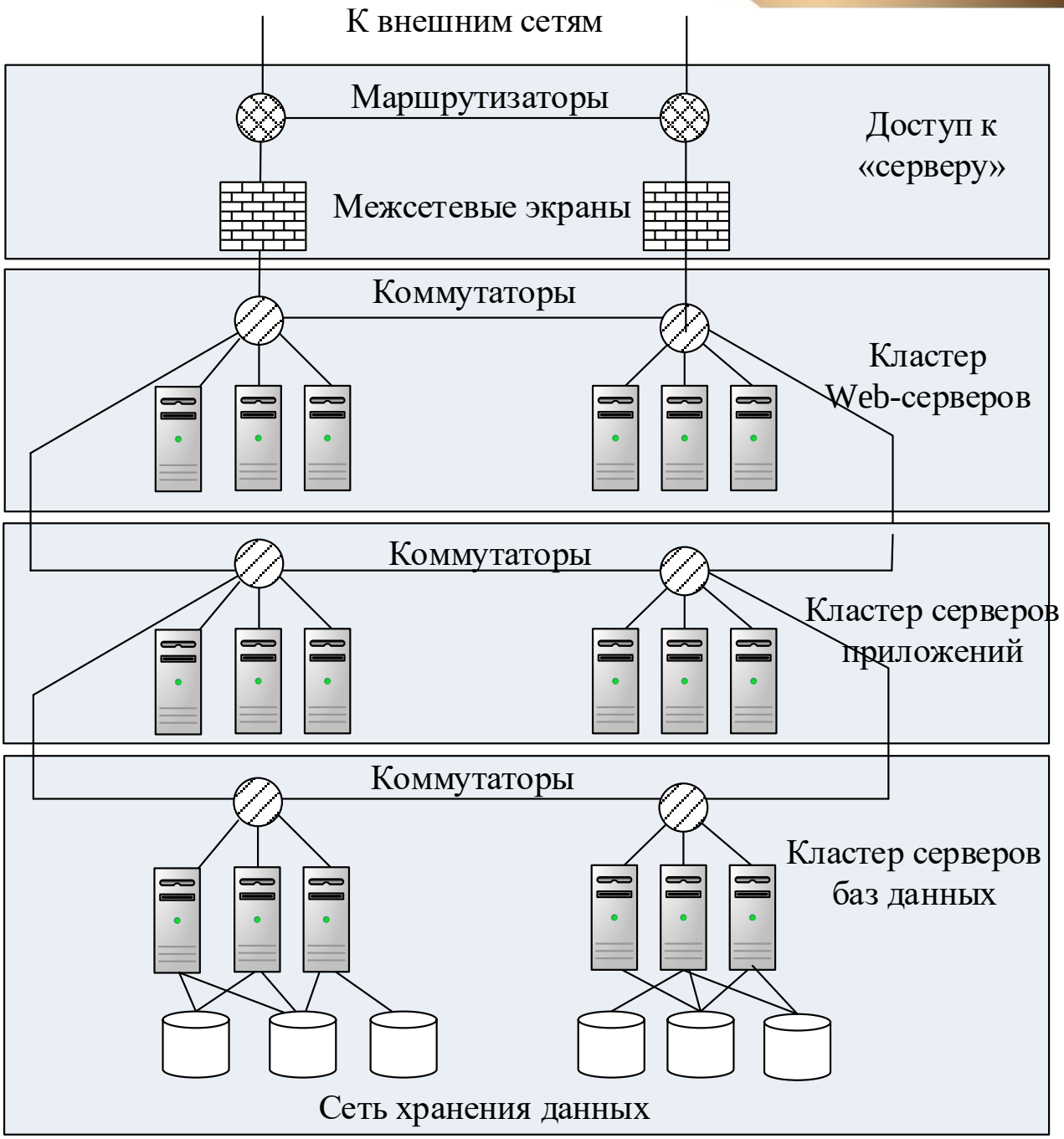


Рисунок 2.7 – Структура ЦОД

В БД NoSQL распределенность и параллельность решается другими механизмами.

БД NoSQL – это молодое направление, которое возникло в ответ на Большие данные, с которыми традиционные реляционные СУБД плохо справляются.

Эффективная обработка огромных объемов является нетривиальной задачей, для решения которой в 2004 году компания Google разработала модель

распределенных вычислений под названием MapReduce (дословно – отображение-свертка).

MapReduce – это модель (шаблон) параллельной обработки данных на кластерах: включает две параллельные операции Map и Reduce (см. рис. 2.8).

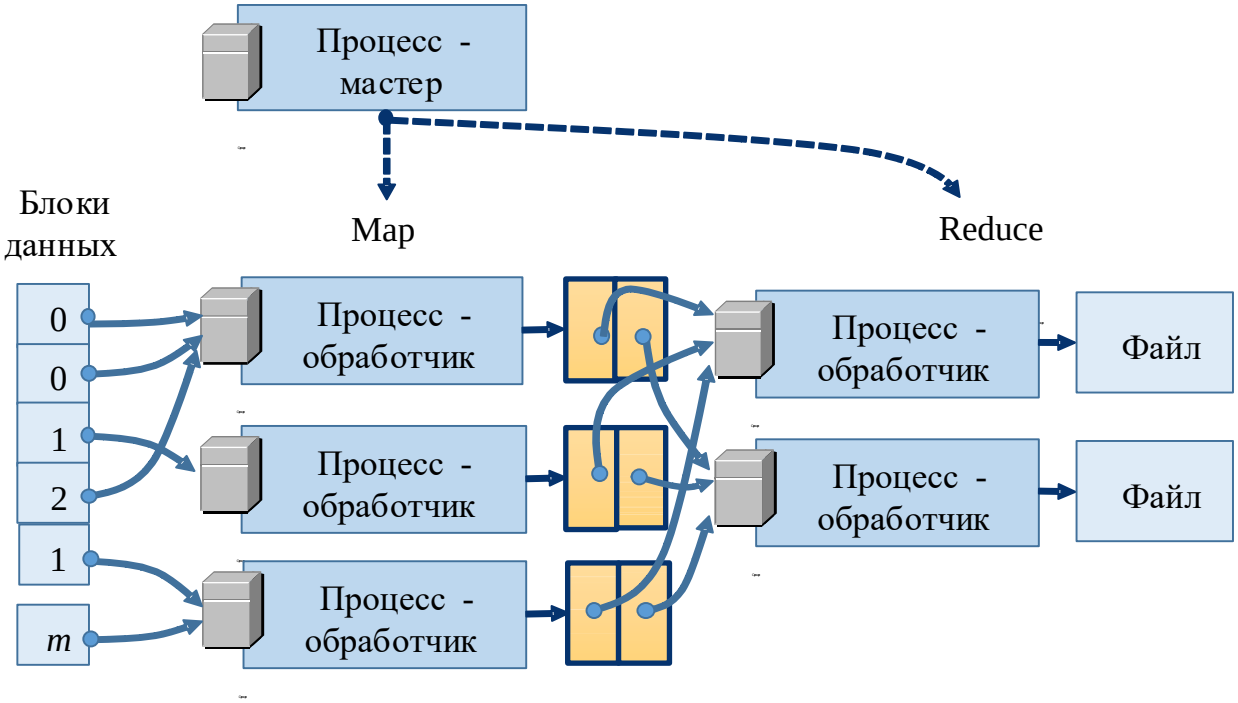


Рисунок 2.8 – Модель MapReduce

Процедура Map: Главный узел (master) делит задачу на части, распределяя их по остальным машинам.

Reduce: Узел Master получает предварительные результаты, и формирует из них конечный результат.

1) Входные файлы разбиваются на m блоков. Данные хранятся в распределенной файловой системе. Типичный размер блока составляет 16-64МБ.

2) На всех узлах кластера системы запускается n экземпляров программы. Один процесс назначается координатором - мастером, а остальные обработчиками.

3) Мастер-процесс распределяет между процессами m задач типа map и r задач типа reduce: $m+ r = n –1$.

4) Процесс типа map выполняет чтение его блока данных и обрабатывает информацию пользовательской функцией map, которая формирует в памяти узла множество пар (ключ-значение)

5) Периодически данные из памяти узла записываются на локальный диск, который разбит на *r* областей. Записанные на локальный диск данные передаются мастер-процессу, который передает эту информацию процессам типа reduce. Номер области $R = 0, 1, \ldots, r-1$ выбирается по формуле: R= hash(key) mod r

6) Получив информацию от мастер-процесса, reduce-процесс выполняет удаленный вызов процедур для чтения данных с локального диска. Прочитав все данные reduce-процесс выполняет их сортировку по ключу, чтобы пары с одинаковыми ключами располагались последовательно.

7) Reduce-процесс передает каждый уникальный ключ и список соответствующих ему значений в пользовательскую функцию reduce, которая порождает на основе этих данных новые данные (ключ-значение). Результаты функции reduce дописывается в выходной файл его reduce-области.

Таким образом, технологии распределенных и параллельных БД направлены на:

– Повышение производительности обработки данных (параллельная работа).
– Повышение надежности (благодаря репликации данных, исключаются одиночные точки отказа).
– Решение вопросов, связанных с возрастанием объема баз данных (масштабирование).

2.3. Транзакции

Данные в БД являются разделяемым ресурсом. Многопользовательский доступ к данным подразумевает одновременное выполнение двух и более запросов к одним и тем же объектам данных (таблицам, блокам и т.п.). Для организации одновременного доступа не обязательно наличие многопроцессорной системы. На однопроцессорной ЭВМ запросы выполняются не одновременно, а параллельно. Для каждого запроса выделяется некоторое количество процессорного времени (квант времени), по истечении которого выполнение запроса приостанавливается, он ставится в очередь запросов, а на выполнение запускается следующий по очереди запрос. Таким образом, процессорное время делится между запросами, и создаётся иллюзия, что запросы выполняются одновременно.

При параллельном доступе к данным запросы на чтение не мешают друг другу. Наоборот, если один запрос считал данные в оперативную память (в буфер

данных), то другой запрос не будет тратить время на обращение к диску за этими данными, а получит их из буфера данных. Проблемы возникают в том случае, если доступ подразумевает внесение изменений. Для того чтобы исключить нарушения логической целостности данных при многопользовательском доступе, используется механизм транзакций.

Транзакция – это упорядоченная последовательность операторов обработки данных, которая переводит базу данных из одного согласованного состояния в другое.

Все команды работы с данными выполняются в рамках транзакций. Для каждого сеанса связи с БД в каждый момент времени может существовать единственная транзакция или не быть ни одной транзакции.

Транзакция обладает следующими свойствами:

1. Логическая неделимость (**атомарность**, Atomicity) означает, что выполняются либо все операции (команды), входящие в транзакцию, либо ни одной. Система гарантирует невозможность запоминания части изменений, произведённых транзакцией. До тех пор, пока транзакция не завершена, её можно "откатить", т.е. отменить все сделанные командами транзакции изменения. Успешное выполнение транзакции (фиксация) означает, что все команды транзакции проанализированы, интерпретированы как правильные и безошибочно исполнены.
2. **Согласованность** (Consistency): транзакция начинается на согласованном множестве данных и после её завершения множество данных согласовано. Состояние БД является согласованным, если данные удовлетворяют всем установленным ограничениям целостности и относятся к одному моменту в состоянии предметной области.
3. **Изолированность** (Isolation), т.е. отсутствие влияния транзакций друг на друга.
4. **Устойчивость** (Durability): результаты завершённой транзакции не могут быть потеряны. Возврат БД в предыдущее состояние может быть достигнут только путём запуска компенсирующей транзакции.

Транзакции, удовлетворяющие этим свойствам, называют ACID-транзакциями (по первым буквам названий свойств).

Для управления транзакциями в системах, поддерживающих механизм транзакций и язык SQL, используются следующие операторы:

- **фиксация** транзакции (запоминание изменений): COMMIT [WORK];

- **откат** транзакции (отмена изменений): ROLLBACK [WORK];
- создание точки сохранения: SAVEPOINT <имя_точки_сохранения>;

(Ключевое слово WORK необязательно). Для фиксации или отката транзакции система создаёт неявные точки фиксации и отката (см. рис. 2.9).

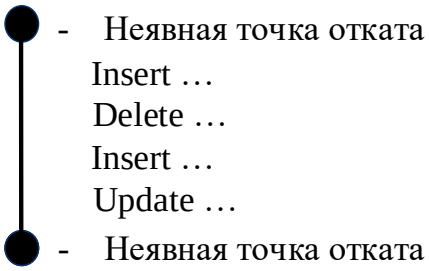


Рисунок 2.9 – Неявные точки фиксации и отката транзакции

По команде rollback система откатит транзакцию на начало (на неявную точку отката), а по команде commit – зафиксирует всё до неявной точки фиксации, которая соответствует последней завершённой команде в транзак-ции. Если в транзакции из нескольких команд во время выполнения очередной команды возникнет ошибка, то система откатит только эту ошибочную команду, т.е. отменит её результаты и сохранит прежнюю неявную точку фиксации.

Для обеспечения целостности транзакции СУБД может откладывать запись изменений в БД до момента успешного выполнения всех операций, входящих в транзакцию, и получения команды подтверждения транзакции (commit). Но чаще используется другой подход: система записывает изменения в БД, не дожидаясь завершения транзакции, а старые значения данных сохраняет на время выполнения транзакции в сегментах отката.

Сегмент отката (rollback segment, RBS) – это специальная область памяти на диске, в которую записывается информация обо всех текущих (незавершённых) изменениях. Обычно записывается "старое" и "новое" содержимое изменённых записей, чтобы можно было восстановить прежнее состояние БД при откате транзакции (по команде rollback) или при откате текущей операции (в случае возникновения ошибки). Данные в RBS хранятся до тех пор, пока транзакция, изменяющая эти данные, не будет завершена. Потом они могут быть перезаписаны данными более поздних транзакций.

Команда savepoint запоминает промежуточную "текущую копию" состояния базы данных для того, чтобы при необходимости можно было вернуться к состоянию БД в точке сохранения: откатить работу от текущего момента до точки сохранения (rollback to <имя_точки>) или зафиксировать работу от начала

транзакции до точки сохранения (commit to <имя_точки>). На одну транзакцию может быть несколько точек сохранения (ограничение на их количество зависит от СУБД).

Для сохранения сведений о транзакциях СУБД ведёт журнал транзакций. **Журнал транзакций** – это часть БД, в которую поступают данные обо всех изменениях всех объектов БД. Журнал недоступен пользователям СУБД и поддерживается особо тщательно (иногда ведутся две копии журнала, хранимые на разных физических носителях). Форма записи в журнал изменений зависит от СУБД. Но обычно там фиксируется следующее:

- номер транзакции (номера присваиваются автоматически по возрастанию);
- состояние транзакции (завершена фиксацией или откатом, не завершена, находится в состоянии ожидания);
- точки сохранения (явные и неявные);
- команды, составляющие транзакцию, и проч.

Начало транзакции соответствует появлению первого исполняемого SQL-оператора. При этом в журнале появляется запись об этой транзакции.

По стандарту ANSI/ISO транзакция завершается при наступлении одного из следующих событий:

- Поступила команда **commit** (результаты транзакции фиксируются).
- Поступила команда **rollback** (результаты транзакции откатываются).
- Успешно завершена программа (**exit, quit**), в рамках которой выполнялась транзакция. В этом случае транзакция фиксируется автоматически.
- Программа, выполняющая транзакцию, завершена аварийно (**abort**). При этом транзакция автоматически откатывается.

Примечания:

1. Возможна работа в режиме **AUTOCOMMIT**, когда каждая команда воспринимается системой как транзакция. В этом режиме пользователи меньше задерживают друг друга, требуется меньше памяти для сегмента отката, зато результаты ошибочно выполненной операции нельзя отменить командой **rollback**.
2. В некоторых СУБД реализованы расширенные модели транзакций, в которых существуют дополнительные ситуации фиксации транзакций. Например, в СУБД Oracle команды DDL выполняются в режиме **AUTOCOMMIT**, т.е. не могут быть откачены.

Все изменения данных выполняются в оперативной памяти в буфере данных, затем фиксируются в журнале транзакций и в сегменте отката, и периодически (при выполнении контрольной точки) переписываются на диск. Процесс формирования **контрольной точки** (КТ) заключается в синхронизации данных, находящихся на диске (т.е. во вторичной памяти) с теми данными, которые находятся в ОП: все модифицированные данные из ОП переписываются во вторичную память. В разных системах процесс формирования контрольной точки запускается по-разному. Например, в СУБД Oracle КТ формируется:

- при поступлении команды commit,
- при переполнении буфера данных,
- в момент заполнения очередного файла журнала транзакций,
- через три секунды со времени последней записи на диск.

Внесение изменений в журнал транзакций всегда носит опережающий характер по отношению к записи изменений в основную часть БД (протокол WAL – Write Ahead Log). Эта стратегия заключается в том, что запись об изменении любого объекта БД должна попасть во внешнюю память журнала транзакций раньше, чем изменённый объект попадёт во внешнюю память основной части БД. Если СУБД корректно соблюдает протокол WAL, то с помощью журнала транзакций можно решить все проблемы восстановления БД после сбоя, если сбой препятствуют дальнейшему функционированию системы (например, после сбоя приложения или фонового процесса СУБД).

Таким образом, при использовании протокола WAL измененные данные почти сразу попадают в базу данных, ещё по поступления команды commit. Поэтому фиксация транзакции чаще всего заключается в следующем:

1. Изменения, внесённые транзакцией, помечаются как постоянные.
2. Уничтожаются все точки сохранения для данной транзакции.
3. Если выполнение транзакций осуществляется с помощью блокировок, то освобождаются объекты, заблокированные транзакцией (см. раздел 2.5).
4. В журнале транзакций транзакция помечается как завершенная, уничтожаются системные записи о транзакции в оперативной памяти.

А при откате транзакции вместо п.1 обычно выполняется считывание из сегмента отката прежних значений данных и переписывание их обратно в БД (остальные пункты сохранятся без изменений). Поэтому откат транзакции практически всегда занимает больше времени, чем фиксация.

2.4. Взаимовлияние транзакций

Транзакции в многопользовательской БД должны быть изолированы друг от друга, т.е. в идеале каждая из них должна выполняться так, как будто выполняется только она одна. В реальности транзакции выполняются одновременно и могут влиять на результаты друг друга, если они обращаются к одному и тому же набору данных и, хотя бы, одна из транзакций изменяет данные.

В общем случае взаимовлияние транзакций может проявляться в виде:

- потери изменений;
- чернового чтения;
- неповторяемого чтения;
- фантомов

Потеря изменений могла бы произойти при одновременном обновлении двумя и более транзакциями одного и того же набора данных. Транзакция, закончившаяся последней, перезаписала бы результаты изменений, внесённых предыдущими транзакциями, и они были бы потеряны.

Представим, что одновременно начали выполняться две транзакции:

транзакция 1 – UPDATE СОТРУДНИКИ
SET Оклад = 39200
WHERE Номер = 1123;
транзакция 2 – UPDATE СОТРУДНИКИ
SET Должность = "старший экономист"
WHERE Номер = 1123;

Обе транзакции считали одну и ту же запись (1123, "Рудин В.П.", "экономист", 28300) и внесли каждая свои изменения: в бухгалтерии изменили оклад (транзакция 1), в отделе кадров – должность (транзакция 2). Результаты транзакции 1 будут потеряны (см. рис. 2.10).

Отношение «Сотрудник»

Номер	ФИО	Должность	Оклад	
1 123	Рудин В.П.	экономист	28300	
1 123	Рудин В.П.	экономист	**39200**	Транзакция 1
1 123	Рудин В.П.	**старший экономист**	28300	Транзакция 2

Рисунок 2.10 – Недопустимое взаимовлияние транзакций: потеря изменений

СУБД не допускает такого взаимовлияния транзакций, при котором возможна потеря изменений!

Ситуация **чернового чтения** возникает, когда транзакция считывает изменения, вносимые другой (незавершенной) транзакцией. Если эта вторая транзакция не будет зафиксирована, то данные, полученные в результате чернового чтения, будут некорректными. Транзакции, осуществляющие черновое чтение, могут использоваться только при невысоких требованиях к согласованности данных: например, если транзакция считает статистические показатели, когда отклонения отдельных значений данных слабо влияют на общий результат.

При повторяемом чтении один и тот же запрос, повторно выполняемый одной транзакцией, возвращает один и тот же набор данных (т.е. игнорирует изменения, вносимые другими завершёнными и незавершёнными транзакциями). **Неповторяемое чтение** является противоположностью повторяемого, т.е. транзакция "видит" изменения, внесённые другими (завершёнными!) транзакциями. Следствием этого может быть несогласованность результатов запроса, когда часть данных запроса соответствует состоянию БД до внесения изменений, а часть – состоянию БД после внесения и фиксации изменений.

Фантомы – это особый тип неповторяемого чтения. Возникновение фантомов может происходить в ситуации, когда одна и та же транзакция сначала производит обновление набора данных, а затем считывание этого же набора. Если считывание данных начинается раньше, чем закончится их обновление, то в результате чтения можно получить несогласованный (не обновлённый или частично обновлённый) набор данных. При последующих запросах это явление пропадает, т.к. на самом деле запрошенные данные после завершения обновления будут согласованными в соответствие со свойствами транзакции.

Для разграничения двух пишущих транзакций и предотвращения потери изменений СУБД используют механизмы блокировок или временных отметок, а для разграничения пишущей и читающих транзакций – специальные правила поведения транзакций, которые называются уровнями изоляции транзакций.

Уровни изоляции транзакций

Стандарт ANSI/ISO для SQL устанавливает различные уровни изоляции для операций, выполняемых над БД, которые работают в многопользовательском режиме. **Уровень изоляции** определяет, может ли читающая транзакция *считывать* ("видеть") результаты работы других одновременно выполняемых завершённых и/или незавершённых пишущих транзакций (табл. 2.1). Использование уровней изоляции обеспечивает предсказуемость работы приложений.

Таблица 2.1. Уровни изоляции по стандарту ANSI / ISO

Уровень изоляции	Черновое чтение	Неповторяемое чтение	Фантомы
Read Uncommited – чтение незавершённых транзакций	да	да	да
Read Commited – чтение завершённых транзакций	нет	да	да
Repeatable Read – повторяемое чтение	нет	нет	да
Serializable – последовательное чтение	нет	нет	нет

По умолчанию в СУБД обычно установлен уровень Read Commited.

Уровень изоляции позволяет транзакциям в большей или меньшей степени влиять друг на друга: при повышении уровня изоляции повышается согласованность данных, но снижается степень параллельности работы и, следовательно, производительность системы.

2.4. Блокировки транзакций

Блокировка – это временное ограничение доступа к данным, участвующим в транзакции, со стороны других транзакций.

Блокировка относится к пессимистическим алгоритмам, т.к. предполагается, что существует высокая вероятность одновременного обращения нескольких пишущих транзакций к одним и тем же данным. Различают следующие типы блокировок:

- по степени доступности данных: разделяемые и исключающие;
- по множеству блокируемых данных: строчные, страничные, табличные;
- по способу установки: автоматические и явные.

Строчные, страничные и табличные блокировки накладываются соответственно на строку таблицы, страницу (блок) памяти и на всю таблицу целиком. Табличная блокировка приводит к неоправданным задержкам исполнения запросов и сводит на нет параллельность работы. Другие виды блокировки увеличивают параллелизм работы, но требуют накладных расходов на поддержание блокировок: наложение и снятие блокировок требует времени, а для хранения информации о наложенной блокировке нужна дополнительная память (для каждой записи или блока данных).

Разделяемая блокировка, установленная на определённый ресурс, предоставляет транзакциям право коллективного доступа к этому ресурсу. Обычно этот вид блокировок используется для того, чтобы запретить другим транзакциям производить необратимые изменения. Например, если на таблицу целиком наложена разделяемая блокировка, то ни одна транзакция не сможет удалить эту таблицу или изменить её структуру до тех пор, пока эта блокировка не будет снята. (При выполнении запросов на чтение обычно накладывается разделяемая блокировка на таблицу.)

Исключающая блокировка предоставляет право на монопольный доступ к ресурсу. Исключающая (монопольная) блокировка таблицы накладывается, например, в случае выполнения операции ALTER TABLE, т.е. изменения структуры таблицы. На отдельные записи (блоки) монопольная блокировка накладывается тогда, когда эти записи (блоки) подвергаются модификации.

Блокировка может быть **автоматической** и **явной**. Если запускается новая транзакция, СУБД сначала проверяет, не заблокирована ли другой транзакцией строка, требуемая этой транзакции: если нет, то строка автоматически

блокируется и выполняется операция над данными; если строка заблокирована, транзакция ожидает снятия блокировки. Явная блокировка, накладываемая командой LOCK TABLE языка SQL, обычно используется тогда, когда транзакция затрагивает существенную часть отношения. Это позволяет не тратить время на построчную блокировку таблицы. Кроме того, при большом количестве построчных блокировок транзакция может не завершиться (из-за возникновения взаимных блокировок, например), и тогда все сделанные изменения придётся откатить, что снизит производительность системы.

Явную блокировку также можно наложить с помощью ключевых слов *for update*, например:

```
SELECT *
FROM <имя_таблицы>
WHERE <условие>
for update;
```

При этом блокировка будет накладываться на те записи, которые удовлетворяют <условию>.

И явные, и неявные блокировки снимаются при завершении транзакции.

Блокировки могут стать причиной бесконечного ожидания и тупиковых ситуаций. **Бесконечное ожидание** возможно в том случае, если не соблюдается очерёдность обслуживания транзакций и транзакция, поступившая раньше других, всё время отодвигается в конец очереди. Решение этой проблемы основывается на выполнении правила FIFO (first input – first output): «первый пришел – первый ушел».

Тупиковые ситуации (deadlocks) возникают при взаимных блокировках транзакций, которые выполняются на пересекающихся множествах данных (см. рис. 2.11). Здесь приведён пример взаимной блокировки трех транзакций T_i на отношениях R_j. Транзакция T_1 заблокировала данные B_1 в отношении R_1 и ждёт освобождения данных B_2 в отношении R_2, которые заблокированы транзакцией T_2, ожидающей освобождения данных B_3 в отношении R_3, заблокированных транзакцией T_3, которая не может продолжить выполнение из-за транзакции T_1. Если не предпринимать никаких дополнительных действий, то эти транзакции никогда не завершатся, т.к. они вечно будут ждать друг друга.

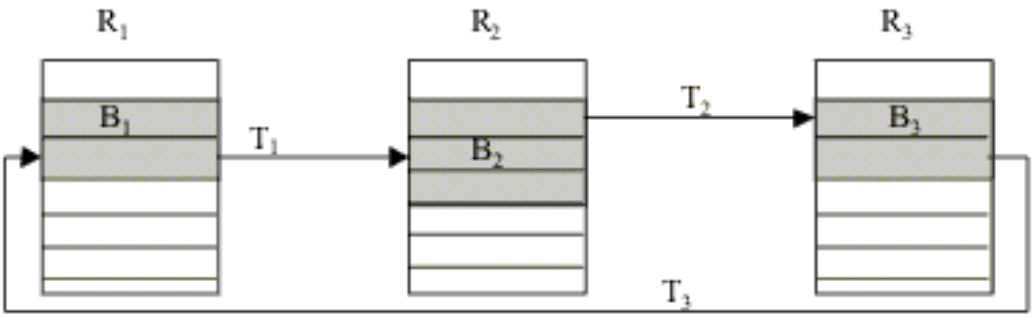


Рисунок 2.11 – Взаимная блокировка трех транзакций

Существует много стратегий разрешения проблемы взаимной блокировки, в частности:

1. Транзакция запрашивает сразу все требуемые блокировки. Такой метод снижает степень параллелизма в работе системы. Также он не может применяться в тех случаях, когда заранее неизвестно, какие данные потребуются, например, если выборка данных из одной таблицы осуществляется на основании данных из другой таблицы, которые выбираются в том же запросе.
2. СУБД отслеживает возникающие тупики и отменяет одну из транзакций с последующим рестартом через случайный промежуток времени. Этот метод требует дополнительных накладных расходов.
3. Вводится **таймаут** (time-out) – максимальное время, в течение которого транзакция может находиться в состоянии ожидания. Если транзакция находится в состоянии ожидания дольше таймаута, считается, что она находится в состоянии тупика, и СУБД инициирует её откат с последующим рестартом через случайный промежуток времени.

Использование временных отметок относится к оптимистическим алгоритмам разграничения транзакций. Для их эффективного функционирования необходимо, чтобы вероятность одновременного обращения нескольких пишущих транзакций к одним и тем же данным была невелика.

Временная отметка – это уникальный идентификатор, который СУБД создаёт для обозначения относительного момента запуска транзакции. Временная отметка может быть создана с помощью системных часов или путём присвоения каждой следующей транзакции очередного номера (SCN – system change number). Каждая транзакция T_i имеет временную отметку t_i, и каждый элемент данных в БД (запись или блок) имеет две отметки: $t_{read}(x)$ – временная отметка транзакции, которая последней считала элемент x, и $t_{write}(x)$ – временная отметка транзакции, которая последней записала элемент x.

При выполнении транзакции T_i система сравнивает отметку t_i и отметки $t_{read}(x)$ и $t_{write}(x)$ элемента ***x*** для обнаружения конфликтов:

1. Для читающей транзакции T_i : если $t_i < t_{write}(x)$, то элемент данных ***x*** перезаписан более поздней транзакцией, и его значение может оказаться несогласованным с теми данными, которые эта транзакция уже успела прочитать.
2. Для пишущей транзакции:

– если $t_i < t_{read}(x)$, то элемент данных x считан более поздней транзакцией. Если транзакция T изменит значение элемента x, то в другой транзакции может возникнуть ошибка.

– если $t_i < t_{write}(x)$, то элемент x перезаписан более поздней транзакцией, и транзакция T пытается поместить в БД устаревшее значение элемента ***x***.

Во всех случаях обнаружения конфликта система перезапускает текущую транзакцию T_i с более поздней временной отметкой. Если конфликта нет, то транзакция выполняется. Очевидно следующее: если разные транзакции часто обращаются к одним и тем же данным одновременно, то транзакции часто будут перезапускаться, и эффективность такого механизма будет невелика.

Для увеличения эффективности выполнения запросов некоторые СУБД используют алгоритм многовариантности. Этот алгоритм позволяет обеспечивать согласованность данных при чтении, не блокируя эти данные.

Согласованность данных для операции чтения заключается в том, что все значения данных должны относиться к тому моменту, когда начиналась эта операция. Для этого можно предварительно запретить другим транзакциям изменять эти данные до окончания операции чтения, но это снижает степень параллельности работы системы.

При использовании алгоритма многовариантности каждый блок данных хранит номер последней транзакции, которая модифицировала данные, хранящиеся в этом блоке (SCN – system change number). И каждая транзакция имеет свой SCN. При чтении данных СУБД сравнивает номер транзакции и номер считываемого блока данных:

– если блок данных не модифицировался с момента начала чтения, то данные считываются из этого блока;

– если данные успели измениться, то система обратится к сегменту отката и считает оттуда значения данных, относящиеся к моменту начала чтения.

Недостатком этого метода является возможность возникновения ошибки при чтении данных, если старые значения данных в сегменте отката будут

перезаписаны. При этом будет выдано сообщение об ошибке и операцию чтения придётся перезапускать вручную. Для устранения подобных проблем можно увеличить размер сегмента отката или разбить одну большую операцию чтения на несколько (но при этом согласованность данных обеспечиваться не будет).

Использование блокировок гарантирует сериальность планов выполнения смеси транзакций за счет общего замедления работы – конфликтующие транзакции ожидают, когда транзакция, первой заблокировавшая некоторый объект, не освободит его. Без блокировок не обойтись, если все транзакции *изменяют* данные. Но если в смеси транзакций присутствуют как транзакции, изменяющие данные, так и *только читающие* данные, можно применить альтернативный механизм обеспечения сериальности, свободный от недостатков метода блокировок. Этот метод состоит в том, что транзакциям, читающим данные, предоставляется как бы «своя» версия данных, имевшаяся в момент начала читающей транзакции. При этом транзакция не накладывает блокировок на читаемые данные, и, поэтому, не блокирует другие транзакции, изменяющие данные. Такой механизм называется ***механизм выделения версий*** и заключается в использовании журнала транзакций для генерации разных версий данных.

Журнал транзакций предназначен для выполнения операции отката при неуспешном выполнении транзакции или для восстановления данных после сбоя системы. Журнал транзакций содержит старые копии данных, измененных транзакциями.

Кратко суть метода состоит в следующем:

- Для каждой транзакции (или запроса) запоминается текущий системный номер SCN. Чем позже начата транзакция, тем больше ее SCN.
- При записи страниц данных на диск фиксируется SCN транзакции, производящей эту запись. Этот SCN становится текущим системным номером страницы данных.
- Транзакции, только читающие данные, не блокируют ничего в базе данных.
- Если транзакция А читает страницу данных, то SCN транзакции А сравнивается с SCN читаемой страницы данных.
- Если SCN страницы данных меньше или равен SCN транзакции А, то транзакция А читает эту страницу.
- Если SCN страницы данных больше SCN транзакции А, то это означает, что некоторая транзакция В, начавшаяся позже транзакции А, успела изменить

или сейчас изменяет данные страницы. В этом случае транзакция A просматривает журнал транзакция назад в поиске первой записи об изменении нужной страницы данных с SCN меньшим, чем SCN транзакции A. Найдя такую запись, транзакция A использует старый вариант данных страницы.