12. Распределенная обработка данных

Оглавление

12.1. **Режимы работы с БД** ... 1
12.2. **Терминология** ... 2
12.3. **Модели "клиент-сервер" в технологии баз данных** ... 4
12.4. **Двухуровневые модели** ... 7
12.4.1. **Модель удаленного управления данными. Модель файлового сервера** ... 7
12.4.2. **Модель удаленного доступа к данным** ... 8
12.4.3. **Модель сервера баз данных** ... 9
12.5. **Трехуровневая модель. Модель сервера приложений** ... 12
12.6. **Модели серверов баз данных** ... 13
12.7. **Типы параллелизма** ... 17

12.1. Режимы работы с БД

При размещении *БД* на персональном компьютере, который не находится в сети, *БД* всегда используется в монопольном режиме. Даже если *БД* используют несколько пользователей, они могут работать с ней только последовательно, и поэтому вопросов о поддержании корректной модификации *БД* в этом случае здесь не стоит, они решаются организационными мерами — то есть определением требуемой последовательности работы конкретных пользователей с соответствующей *БД*. Однако даже в некоторых настольных *БД* требуется учитывать последовательность изменения данных при обработке, чтобы получить корректный результат: так, например, при запуске программы балансного бухгалтерского отчета все бухгалтерские проводки — финансовые *операции* должны быть решены заранее до запуска конечного приложения.

Однако работа на изолированном компьютере с небольшой базой данных в настоящий момент уже давно нехарактерна для большинства приложений. *БД* отражает информационную модель реальной *предметной области*, она растет по объему и резко увеличивается количество задач, решаемых с ее использованием, и в соответствии с этим увеличивается количество приложений, работающих с единой базой данных. Компьютеры объединяются в *локальные сети*, и необходимость распределения приложений, работающих с единой базой данных по сети, является несомненной.

Действительно, даже когда вы строите *БД* для небольшой торговой фирмы, у вас появляется ряд специфических пользователей *БД*, которые

имеют свои бизнес-функции и территориально могут находиться в разных помещениях, но все они должны работать с единой информационной моделью организации, то есть с единой базой данных.

Параллельный *доступ* к одной *БД* нескольких пользователей, в том случае если *БД* расположена на одной машине, соответствует режиму распределенного доступа к централизованной *БД*. (Такие системы называются *системами распределенной обработки данных*.)

Если же *БД* распределена по нескольким компьютерам, расположенным в сети, и к ней возможен параллельный *доступ* нескольких пользователей, то мы имеем дело с параллельным доступом к распределенной *БД*. Подобные системы называются *системами распределенных баз данных*. В общем случае режимы использования *БД* можно представить в следующем виде (рис.12.1).

Рис. 12.1. Режимы работы с базой данных

Определим терминологию, которая нам потребуется для дальнейшей работы. Часть терминов нам уже известна, но повторим здесь их дополнительно.

12.2. Терминология

Пользователь БД — *программа* или человек, обращающийся к *БД* на языке манипуляции данными (ЯМД).

Запрос — процесс обращения пользователя к *БД* с целью ввода, получения или изменения информации в *БД*.

Транзакция — последовательность операций модификации данных в *БД*, переводящая *БД* из одного непротиворечивого состояния в другое непротиворечивое состояние.

Логическая структура БД — *определение БД* на физически независимом уровне, ближе всего соответствует концептуальной модели *БД*.

Топология БД = Структура распределенной БД - схема распределения физической *БД* по сети.

Локальная автономность — означает, что *информация локальной БД* и связанные с ней определения данных принадлежат локальному владельцу и им управляются.

Удаленный запрос — *запрос*, который выполняется с использованием модемной связи.

Возможность реализации удаленной транзакции обработка одной транзакции, состоящей из *множества SQL*-запросов на одном удаленном узле.

Поддержка распределенной транзакции допускает обработку транзакции, состоящей из нескольких запросов *SQL*, которые выполняются на нескольких узлах сети (удаленных или локальных), но каждый*запрос* в этом случае обрабатывается только на одном узле, то есть запросы не являются распределенными. При обработке одной распределенной транзакции разные локальные запросы могут обрабатываться в разных узлах сети.

Распределенный запрос — *запрос*, при обработке которого используются данные из *БД*, расположенные в разных узлах сети.

Системы распределенной обработки данных в основном связаны с первым поколением *БД*, которые строились на мультипрограммных операционных системах и использовали централизованное хранение *БД* на *устройствах внешней памяти* центральной ЭВМ и терминальный *многопользовательский режим* доступа к ней. При этом пользовательские терминалы не имели собственных ресурсов — то есть процессоров и памяти, которые могли бы использоваться для хранения и обработки данных. Первой полностью реляционной системой, работающей в многопользовательском режиме, была *СУБД SYSTEM* R, разработанная фирмой *IBM*, именно в ней были реализованы как язык манипулирования данными *SQL*, так и основные принципы синхронизации, применяемые при распределенной обработке данных, которые до сих пор являются базисными практически во всех коммерческих *СУБД*.

Общая тенденция движения от отдельных *mainframe*-систем к открытым распределенным системам, объединяющим компьютеры среднего класса, получила название *DownSizing*. Этот процесс оказал огромное влияние на развитие архитектур *СУБД* и поставил перед их разработчиками ряд сложных задач. Главная проблема состояла в технологической сложности перехода от централизованного управления данными на одном компьютере и *СУБД*, использовавшей собственные модели, форматы представления данных и языки доступа к данным и т. д., к распределенной обработке данных в неоднородной вычислительной среде, состоящей из соединенных в глобальную *сеть* компьютеров различных моделей и производителей.

В то же время происходил встречный процесс — *UpSizing*. Бурное развитие персональных компьютеров, появление локальных сетей также оказали серьезное влияние на эволюцию *СУБД*. Высокие темпы роста производительности и функциональных возможностей *PC* привлекли внимание разработчиков профессиональных *СУБД*, что привело к их активному распространению на платформе настольных систем.

Сегодня возобладала тенденция создания информационных систем на такой платформе, которая точно соответствовала бы ее масштабам и задачам.

Она получила название *RightSizing* (помещение ровно в тот размер, который необходим).

Однако и в настоящее время большие ЭВМ сохраняются и сосуществуют с современными открытыми системами. Причина этого проста — в свое время в аппаратное и *программное обеспечение* больших ЭВМ были вложены огромные средства: в результате многие продолжают их использовать, несмотря на морально устаревшую архитектуру. В то же время перенос данных и программ с больших ЭВМ на компьютеры нового поколения сам по себе представляет сложную техническую проблему и требует значительных затрат.

12.3. Модели "клиент-сервер" в технологии баз данных

Вычислительная модель "клиент—*сервер*" исходно связана с парадигмой открытых систем, которая появилась в 90-х годах и быстро эволюционировала. Сам термин "клиент-*сервер*" исходно применялся к архитектуре программного обеспечения, которое описывало распределение процесса выполнения по принципу взаимодействия двух программных процессов, один из которых в этой модели назывался "клиентом", а другой — "сервером". Клиентский процесс запрашивал некоторые услуги, а *серверный процесс* обеспечивал их выполнение. При этом предполагалось, что один *серверный процесс* может обслужить множество клиентских процессов.

Ранее *приложение* (пользовательская *программа*) не разделялась на части, оно выполнялось некоторым монолитным блоком. Но возникла идея более рационального использования ресурсов сети. Действительно, при монолитном исполнении используются ресурсы только одного компьютера, а остальные компьютеры в сети рассматриваются как терминалы. Но теперь, в отличие от эпохи main-фреймов, все компьютеры в сети обладают собственными ресурсами, и разумно так распределить нагрузку на них, чтобы максимальным образом использовать их ресурсы.

И как в промышленности, здесь возникает древняя как мир идея *распределения обязанностей*, разделения труда. Конвейеры Форда сделали в свое время прорыв в автомобильной промышленности, показав наивысшую *производительность* труда именно из-за того, что весь процесс сборки был разбит на мелкие и максимально простые *операции* и каждый рабочий специализировался на выполнении только одной *операции*, но эту операцию он выполнял максимально быстро и качественно.

Конечно, в вычислительной технике нельзя было напрямую использовать технологию автомобильного или любого другого механического производства, но идею использовать было можно. Однако для воплощения идеи необходимо было разработать модель разбиения единого монолитного приложения на отдельные части и определить принципы взаимосвязи между этими частями.

Основной принцип технологии "клиент—*сервер*" применительно к технологии баз данных заключается в разделении функций стандартного интерактивного приложения на 5 групп, имеющих различную природу:

- функции ввода и отображения данных (*Presentation Logic*);
- прикладные функции, определяющие основные алгоритмы решения задач приложения (*Business Logic*);
- функции обработки данных внутри приложения (*Database Logic*);
- функции управления информационными ресурсами (*Database Manager System*);
- служебные функции, играющие роль связок между функциями первых четырех групп.

Структура типового приложения, работающего с базой данных приведена на рис. 12.2.

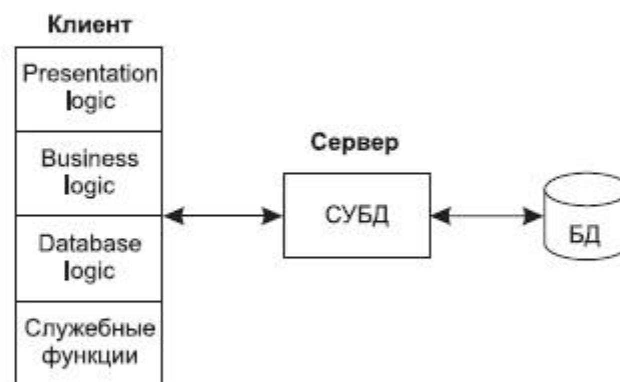

Рис. 12.2. Структура типового интерактивного приложения, работающего с базой данных

Презентационная логика (*Presentation Logic*) как часть приложения определяется тем, что *пользователь* видит на своем экране, когда работает *приложение*. Сюда относятся все интерфейсные экранные формы, которые *пользователь* видит или заполняет в ходе работы приложения, к этой же части относится все то, что выводится пользователю на экран как результаты решения некоторых промежуточных задач либо как справочная *информация*. Поэтому основными задачами презентационной логики являются:

- формирование экранных изображений;
- чтение и запись в экранные формы информации;
- управление экраном;
- обработка движений мыши и нажатие клавиш клавиатуры.

Бизнес-логика, или логика собственно приложений (*Business processing Logic*), — это часть кода приложения, которая определяет собственно алгоритмы решения конкретных задач приложения. Обычно этот код пишется с использованием различных языков программирования, таких как C, C++, Visual-Basic.

Логика обработки данных (*Data manipulation Logic*) — это часть кода приложения, которая связана с обработкой данных внутри приложения. Дан-

ными управляет собственно *СУБД* (*DBMS*). Для обеспечения доступа к данным используются язык запросов и средства манипулирования данными стандартного языка *SQL*.

Обычно *операторы* языка *SQL* встраиваются в языки 3-го или 4-го поколения (*3GL*, *4GL*), которые используются для написания кода приложения.

Процессор управления данными (*Database Manager System Processing*) — это собственно *СУБД*, которая обеспечивает хранение и управление базами данных. В идеале *функции СУБД* должны быть скрыты от бизнес-логики приложения, однако для рассмотрения архитектуры приложения нам надо их выделить в отдельную часть приложения.

В *централизованной архитектуре* (Host-based *processing*) эти части приложения располагаются в единой среде и комбинируются внутри одной исполняемой программы.

В децентрализованной архитектуре эти задачи могут быть по-разному распределены между серверным и клиентским процессами. В зависимости от характера распределения можно выделить следующие модели распределений (рис. 12.3):

1. распределенная презентация (*Distribution presentation*, DP);
2. удаленная презентация (*Remote Presentation*, *RP*);
3. распределенная бизнес-логика (*Distributed Business Logic*, DBL);
4. распределенное управление данными (*Distributed data management*, *DDM*);
5. удаленное управление данными (*Remote data management*, *RDM*).

Рис. 12.3. Распределение функций приложения в моделях "клиент—сервер"

Эта условная классификация показывает, как могут быть распределены отдельные задачи между серверным и клиентскими процессами. В этой классификации отсутствует реализация удаленной бизнес-логики. Действительно, считается, что она не может быть удалена сама по себе полностью. Считается, что она может быть распределена между разными процессами, которые в общем-то могут выполняться на разных платформах, но должны корректно кооперироваться (взаимодействовать) друг с другом.

12.4. Двухуровневые модели

Двухуровневая модель фактически является результатом распределения пяти указанных функций между двумя процессами, которые выполняются на двух платформах: на клиенте и на сервере. В чистом виде почти никакая модель не существует, однако рассмотрим наиболее характерные особенности каждой двухуровневой модели.

12.4.1. Модель удаленного управления данными. Модель файлового сервера

Модель удаленного управления данными также называется моделью файлового сервера (*File Server*, FS). В этой модели презентационная логика и бизнес-логика располагаются на клиенте. На сервере располагаются файлы с данными и поддерживается доступ к файлам. Функции управления информационными ресурсами в этой модели находятся на клиенте.

Распределение функций в этой модели представлено на рис. 12.4.

В этой модели файлы базы данных хранятся на сервере, клиент обращается к серверу с файловыми командами, а механизм управления всеми информационными ресурсами, собственно база мета-данных, находится на клиенте.

Рис. 12.4. Модель файлового сервера

Достоинства этой модели в том, что мы уже имеем разделение монопольного приложения на два взаимодействующих процесса. При этом сервер (*серверный процесс*) может обслуживать множество клиентов, которые обращаются к нему с запросами. Собственно, СУБД должна находиться в этой модели на клиенте.

Каков алгоритм выполнения запроса клиента?

Запрос клиента формулируется в командах ЯМД. СУБД переводит этот запрос в последовательность файловых команд. Каждая файловая команда вызывает перекачку блока информации на клиента, далее на клиенте СУБД анализирует полученную информацию, и если в полученном блоке не содержится ответ на запрос, то принимается решение о перекачке следующего блока информации и т. д.

Перекачка информации с сервера на клиент производится до тех пор, пока не будет получен ответ на запрос клиента.

Недостатки:

- высокий сетевой трафик, который связан с передачей по сети множества блоков и файлов, необходимых приложению;
- узкий спектр операций манипулирования с данными, который определяется только файловыми командами;
- отсутствие адекватных средств безопасности доступа к данным (защита только на уровне файловой системы).

12.4.2. Модель удаленного доступа к данным

В модели удаленного доступа (Remote Data Access, *RDA*) база данных хранится на сервере. На сервере же находится ядро СУБД. На клиенте располагается презентационная логика и бизнес-логика приложения. Клиент обращается к серверу с запросами на языке SQL. Структура модели удаленного доступа приведена на рис. 12.5.

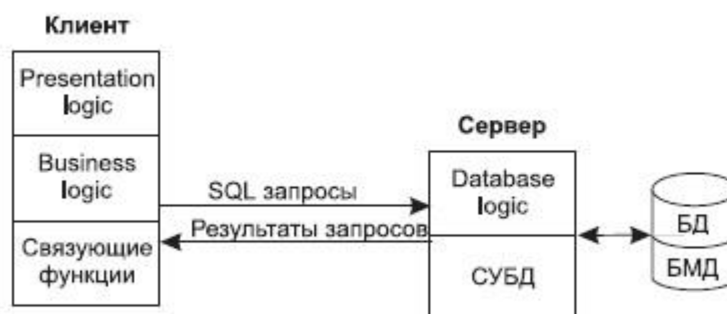

Рис. 12.5. Модель удаленного доступа (RDA)

Преимущества данной модели:

- перенос компонента представления и прикладного компонента на клиентский компьютер существенно разгрузил сервер БД, сводя к минимуму общее число процессов в операционной системе;
- сервер БД освобождается от несвойственных ему функций; процессор или процессоры сервера целиком загружаются операциями обработки данных, запросов и транзакций. (Это становится возможным, если отказаться от терминалов, не располагающих ресурсами, и заменить их компьютерами, выполняющими роль клиентских станций, которые обладают собственными локальными вычислительными ресурсами);
- резко уменьшается загрузка сети, так как по ней от клиентов к серверу передаются не запросы на ввод-вывод в файловой терминологии, а запросы на SQL, и их объем существенно меньше. В ответ на запросы клиент получает только данные, релевантные запросу, а не блоки файлов, как в FS-модели.

Основное достоинство *RDA*-модели — унификация интерфейса "клиент-сервер", стандартом при общении приложения-клиента и сервера становится язык SQL.

Недостатки:

- все-таки запросы на языке SQL при интенсивной работе клиентских приложений могут существенно загрузить сеть;
- так как в этой модели на клиенте располагается и презентационная логика, и бизнес-логика приложения, то при повторении аналогичных функций в разных приложениях код соответствующей бизнес-логики должен быть повторен для каждого клиентского приложения. Это вызывает излишнее дублирование кода приложений;
- сервер в этой модели играет пассивную роль, поэтому функции управления информационными ресурсами должны выполняться на клиенте. Действительно, например, если нам необходимо выполнять контроль страховых запасов товаров на складе, то каждое приложение, которое связано с изменением состояния склада, после выполнения операций модификации данных, имитирующих продажу или удаление товара со склада, должно выполнять проверку на объем остатка, и в случае, если он меньше страхового запаса, формировать соответствующую заявку на поставку требуемого товара. Это усложняет клиентское приложение, с одной стороны, а с другой — может вызвать необоснованный заказ дополнительных товаров несколькими приложениями.

12.4.3. Модель сервера баз данных

Для того чтобы избавиться от недостатков модели удаленного доступа, должны быть соблюдены следующие условия:

1. Необходимо, чтобы БД в каждый момент отражала текущее *состояние предметной области*, которое определяется не только собственно данными, но и связями между объектами данных. То есть данные,

которые хранятся в БД, в каждый момент времени должны быть непротиворечивыми.

2. БД должна отражать некоторые правила предметной области, законы, по которым она функционирует (*business rules*). Например, завод может нормально работать только в том случае, если на складе имеется некоторый достаточный запас (страховой запас) деталей определенной номенклатуры, деталь может быть запущена в производство только в том случае, если на складе имеется в наличии достаточно материала для ее изготовления, и т. д.

3. Необходим постоянный контроль за состоянием БД, отслеживание всех изменений и адекватная реакция на них: например, при достижении некоторым измеряемым параметром критического значения должно произойти отключение определенной аппаратуры, при уменьшении товарного запаса ниже допустимой нормы должна быть сформирована заявка конкретному поставщику на поставку соответствующего товара.

4. Необходимо, чтобы возникновение некоторой ситуации в БД четко и оперативно влияло на ход выполнения прикладной задачи.

5. Одной из важнейших проблем СУБД является контроль типов данных. В настоящий момент СУБД контролирует синтаксически только стандартно-допустимые типы данных, то есть такие, которые определены в DDL (*data definition language*) — языке описания данных, который является частью SQL. Однако в реальных предметных областях у нас действуют данные, которые несут в себе еще и семантическую составляющую, например, это координаты объектов или единицы различных метрик, например рабочая неделя в отличие от реальной имеет сразу после пятницы понедельник.

Данную модель поддерживают большинство современных СУБД: *Informix*, Ingres, Sybase, Oracle, MS SQL Server. Основу данной модели составляет механизм хранимых процедур как средство программирования SQL-сервера, *механизм триггеров* как механизм отслеживания текущего состояния информационного хранилища и механизм ограничений на пользовательские типы данных, который иногда называется механизмом поддержки доменной структуры. Модель сервера баз данных представлена на рис. 12.6.

Рис. 12.6. Модель активного сервера БД

В этой модели бизнес-логика разделена между клиентом и сервером. На сервере бизнес-логика реализована в виде хранимых процедур — специальных программных модулей, которые хранятся в БД и управляются непосредственно СУБД. Клиентское приложение обращается к серверу с командой запуска хранимой процедуры, а сервер выполняет эту процедуру и регистрирует все изменения в БД, которые в ней предусмотрены. Сервер возвращает клиенту данные, релевантные его запросу, которые требуются клиенту либо для вывода на экран, либо для выполнения части бизнес-логики, которая расположена на клиенте. Трафик обмена информацией между клиентом и сервером резко уменьшается.

Централизованный контроль в модели сервера баз данных выполняется с использованием механизма триггеров. Триггеры также являются частью БД.

Термин "триггер" взят из электроники и семантически очень точно характеризует механизм отслеживания специальных событий, которые связаны с состоянием БД. Триггер в БД является как бы некоторым тумблером, который срабатывает при возникновении определенного события в БД. Ядро СУБД проводит мониторинг всех событий, которые вызывают созданные и описанные триггеры в БД, и при возникновении соответствующего события сервер запускает соответствующий триггер. Каждый триггер представляет собой также некоторую программу, которая выполняется над базой данных. Триггеры могут вызывать хранимые процедуры.

Механизм использования триггеров предполагает, что при срабатывании одного триггера могут возникнуть события, которые вызовут срабатывание других триггеров. Этот мощный инструмент требует тонкого и согласованного применения, чтобы не получился бесконечный цикл срабатывания триггеров.

В данной модели сервер является активным, потому что не только клиент, но и сам сервер, используя *механизм триггеров*, может быть инициатором обработки данных в БД.

И хранимые процедуры, и триггеры хранятся в словаре БД, они могут быть использованы несколькими клиентами, что существенно уменьшает дублирование алгоритмов обработки данных в разных клиентских приложениях.

Для написания хранимых процедур и триггеров используется расширение стандартного языка SQL, так называемый *встроенный SQL*.

Недостатком данной модели является очень большая загрузка сервера. Действительно, сервер обслуживает множество клиентов и выполняет следующие функции:

- осуществляет мониторинг событий, связанных с описанными триггерами;
- обеспечивает автоматическое срабатывание триггеров при возникновении связанных с ними событий;

- обеспечивает исполнение внутренней программы каждого триггера;
- запускает хранимые процедуры по запросам пользователей;
- запускает хранимые процедуры из триггеров;
- возвращает требуемые данные клиенту;
- обеспечивает все *функции СУБД*: доступ к данным, контроль и поддержку целостности данных в БД, контроль доступа, обеспечение корректной параллельной работы всех пользователей с единой БД.

Если мы переложили на сервер большую часть бизнес-логики приложений, то требования к клиентам в этой модели резко уменьшаются. Иногда такую модель называют моделью с "*тонким клиентом*", в отличие от предыдущих моделей, где на клиента возлагались гораздо более серьезные задачи. Эти модели называются моделями с "*толстым клиентом*".

Для разгрузки сервера была предложена трехуровневая модель.

12.5. Трехуровневая модель. Модель сервера приложений

Эта модель является расширением двухуровневой модели и в ней вводится дополнительный промежуточный уровень между клиентом и сервером. *Архитектура* трехуровневой модели приведена на рис. 12.7. Этот промежуточный уровень содержит один или несколько серверов приложений.

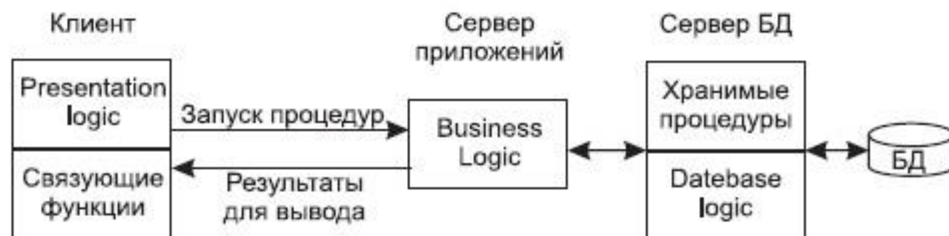

Рис. 12.7. Модель сервера приложений

В этой модели компоненты приложения делятся между тремя исполнителями:

- *Клиент* обеспечивает логику представления, включая графический пользовательский интерфейс, локальные редакторы; клиент может запускать локальный код приложения клиента, который может содержать обращения к *локальной БД*, расположенной на компьютере-клиенте. Клиент исполняет коммуникационные функции front-end части приложения, которые обеспечивают доступ клиенту в локальную или глобальную сеть. Дополнительно реализация взаимодействия между клиентом и сервером может включать в себя управление распределенными транзакциями, что соответствует тем случаям, когда клиент также является клиентом менеджера распределенных транзакций.

- *Серверы приложений* составляют новый промежуточный уровень архитектуры. Они спроектированы как исполнения общих незагружаемых функций для клиентов. Серверы приложений поддерживают функции клиентов как частей взаимодействующих рабочих групп, поддерживают сетевую доменную операционную среду, хранят и исполняют наиболее общие правила бизнес-логики, поддерживают каталоги с данными, обеспечивают обмен сообщениями и поддержку запросов, особенно в распределенных транзакциях.

- *Серверы баз данных* в этой модели занимаются исключительно функциями СУБД: обеспечивают функции создания и ведения БД, поддерживают целостность реляционной БД, обеспечивают функции хранилищ данных (warehouse services). Кроме того, на них возлагаются функции создания резервных копий БД и восстановления БД после сбоев, управления выполнением транзакций и поддержки устаревших (унаследованных) приложений (legacy application).

Отметим, что эта модель обладает большей гибкостью, чем двухуровневые модели. Наиболее заметны преимущества модели сервера приложений в тех случаях, когда клиенты выполняют сложные аналитические расчеты над базой данных, которые относятся к области OLAP-приложений. (On-line *analytical processing*.) В этой модели большая часть бизнес-логики клиента изолирована от возможностей встроенного *SQL*, реализованного в конкретной *СУБД*, и может быть выполнена на стандартных языках программирования, таких как C, C++. Это повышает *переносимость* системы, ее *маситабируемость*.

Функции промежуточных серверов могут быть в этой модели распределены в рамках глобальных транзакций путем поддержки XA-протокола (X/Open *transaction interface protocol*), который поддерживается большинством поставщиков *СУБД*.

12.6. Модели серверов баз данных

В период создания первых *СУБД* технология "клиент-*сервер*" только зарождалась. Поэтому изначально в архитектуре систем не было адекватного механизма организации взаимодействия процессов типа "клиент" и процессов типа "*сервер*". В современных же *СУБД* он является фактически основополагающим и от эффективности его реализации зависит эффективность работы системы в целом.

Рассмотрим эволюцию типов организации подобных механизмов. В основном этот механизм определяется структурой реализации серверных процессов, и часто он называется архитектурой сервера баз данных.

Первоначально, как мы уже отмечали, существовала модель, когда *управление данными* (*функция* сервера) и взаимодействие с пользователем были совмещены в одной программе. Это можно назвать нулевым этапом развития серверов *БД*.

Затем функции управления данными были выделены в самостоятельную группу — *сервер*, однако модель взаимодействия пользователя с сервером соответствовала парадигме "*один-к-одному*" (рис. 12.8), то есть *сервер* обслуживал запросы только одного пользователя (клиента), и для обслуживания нескольких клиентов нужно было запустить эквивалентное число серверов.

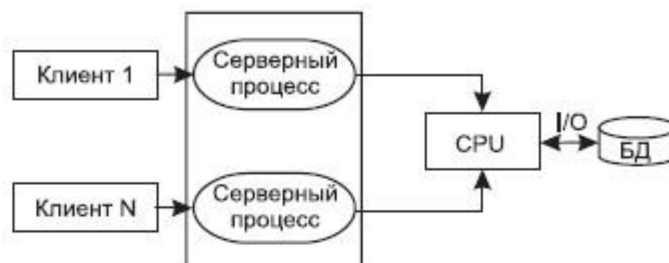

Рис. 12.8. Взаимодействие пользовательских и клиентских процессов в модели "один-к-одному"

Выделение сервера в отдельную программу было революционным шагом, который позволил, в частности, поместить *сервер* на одну машину, а программный *интерфейс* с пользователем — на другую, осуществляя взаимодействие между ними по сети. Однако необходимость запуска большого числа серверов для обслуживания *множества* пользователей сильно ограничивала возможности такой системы.

Для обслуживания большого числа клиентов на сервере должно быть запущено большое количество одновременно работающих серверных процессов, а это резко повышало требования к ресурсам ЭВМ, на которой запускались все серверные процессы. Кроме того, каждый *серверный процесс* в этой модели запускался как независимый, поэтому если один клиент сформировал *запрос*, который был только что выполнен другим серверным процессом для другого клиента, то *запрос* тем не менее выполнялся повторно. В такой модели весьма сложно обеспечить взаимодействие серверных процессов. Эта модель самая простая, и исторически она появилась первой.

Проблемы, возникающие в модели "*один-к-одному*", решаются в архитектуре "систем с выделенным сервером", который способен обрабатывать запросы от многих клиентов. *Сервер* единственный обладает монополией на *управление данными* и взаимодействует одновременно со многими клиентами (рис. 12.9). Логически каждый клиент связан с сервером отдельной нитью ("*thread*"), или потоком, по которому пересылаются запросы. Такая *архитектура* получила название многопотоковой односерверной ("multi-threaded").

Она позволяет значительно уменьшить нагрузку на операционную систему, возникающую при работе большого числа пользователей ("trashing").

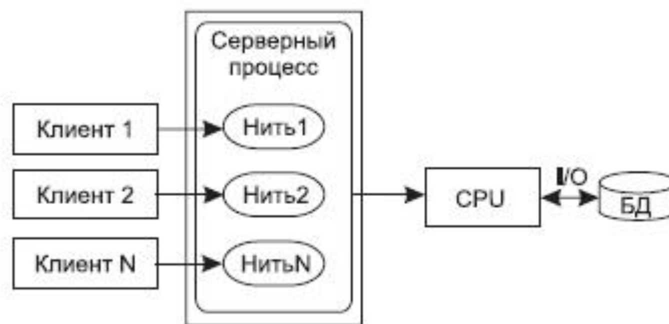

Рис. 12.9. Многопотоковая односерверная архитектура

Кроме того, возможность взаимодействия с одним сервером многих клиентов позволяет в полной мере использовать разделяемые объекты (начиная с открытых файлов и кончая данными из системных каталогов), что значительно уменьшает потребности в памяти и общее число процессов операционной системы. Например, системой с архитектурой "*один-к-одному*" будет создано 100 копий процессов *СУБД* для 100 пользователей, тогда как системе с многопотоковой архитектурой для этого понадобится только один *серверный процесс*.

Однако такое решение имеет свои недостатки. Так как *сервер* может выполняться только на одном процессоре, возникает естественное ограничение на применение *СУБД* для мультипроцессорных платформ. Если*компьютер* имеет, например, четыре процессора, то *СУБД* с одним сервером используют только один из них, не загружая оставшиеся три.

В некоторых системах эта проблема решается вводом промежуточного диспетчера. Подобная *архитектура* называется архитектурой виртуального сервера ("*virtual server*") (рис. 12.10).

В этой архитектуре клиенты подключаются не к реальному серверу, а к промежуточному звену, называемому диспетчером, который выполняет только функции диспетчеризации запросов к актуальным серверам. В этом случае нет ограничений на использование многопроцессорных платформ. Количество актуальных серверов может быть согласовано с количеством процессоров в системе.

Однако и эта *архитектура* не лишена недостатков, потому что здесь в систему добавляется новый слой, который размещается между клиентом и сервером, что увеличивает трату ресурсов на поддержку баланса загрузки актуальных серверов ("*load balancing*") и ограничивает возможности управления взаимодействием "клиент—*сервер*". Во-первых, становится невозможным направить *запрос* от конкретного клиента конкретному серверу, во-вторых, серверы становятся равноправными — нет возможности устанавливать приоритеты для обслуживания запросов.

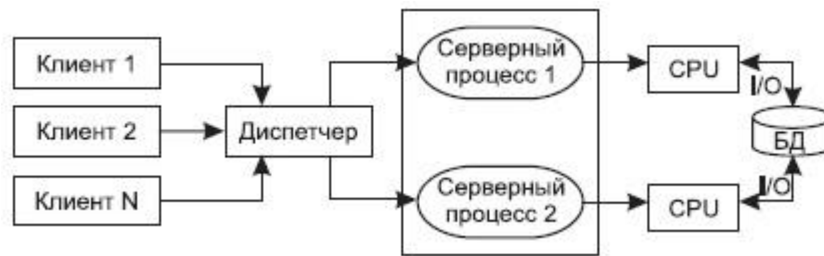

Рис. 12.12. Архитектура с виртуальным сервером

Подобная организация взаимодействия *клиент-сервер* может рассматриваться как аналог банка, где имеется несколько окон кассиров, и специальный банковский служащий — *администратор* зала (*диспетчер*) направляет каждого вновь пришедшего посетителя (клиента) к свободному кассиру (актуальному серверу). Система работает нормально, пока все посетители равноправны (имеют равные приоритеты), однако стоит лишь появиться посетителям с высшим приоритетом, которые должны обслуживаться в специальном окне, как возникают проблемы. Учет приоритета клиентов особенно важен в системах оперативной обработки транзакций, однако именно эту возможность не может предоставить *архитектура* систем с диспетчеризацией.

Современное решение проблемы *СУБД* для мультипроцессорных платформ заключается в возможности запуска нескольких серверов *базы данных*, в том числе и на различных процессорах. При этом каждый из серверов должен быть многопотоковым. Если эти два условия выполнены, то есть основания говорить о многопотоковой архитектуре с несколькими серверами, представленной на рис. 12.11.

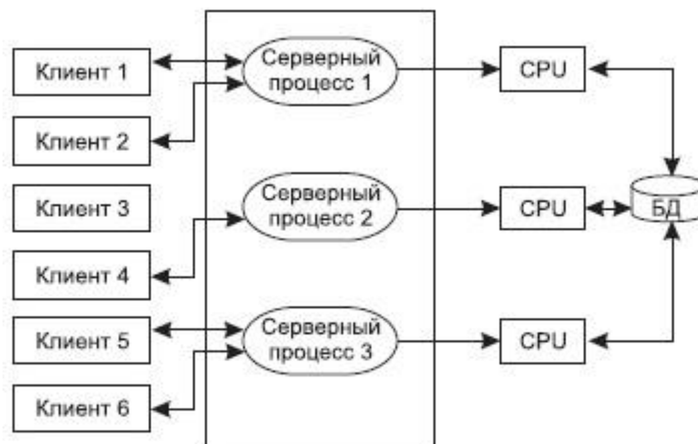

Рис. 12.11. Многопотоковая мультисерверная архитектура

Она также может быть названа многонитевой мультисерверной архитектурой. Эта *архитектура* связана с вопросами распараллеливания выполнения одного пользовательского запроса несколькими серверными процессами.

Существует несколько возможностей распараллеливания выполнения запроса. В этом случае пользовательский *запрос* разбивается на ряд подзапросов, которые могут выполняться параллельно, а результаты их выполнения потом объединяются в общий результат выполнения запроса. Тогда для обеспечения оперативности выполнения запросов их подзапросы могут быть направлены отдельным серверным процессам, а потом полученные результаты объединены в общий результат (рис. 12.12). В данном случае серверные процессы не являются независимыми процессами, такими, как рассматривались ранее. Эти серверные процессы принято называть нитями (threads), и управление нитями *множества* запросов пользователей требует дополнительных расходов от *СУБД*, однако при оперативной обработке информации в хранилищах данных такой подход наиболее перспективен.

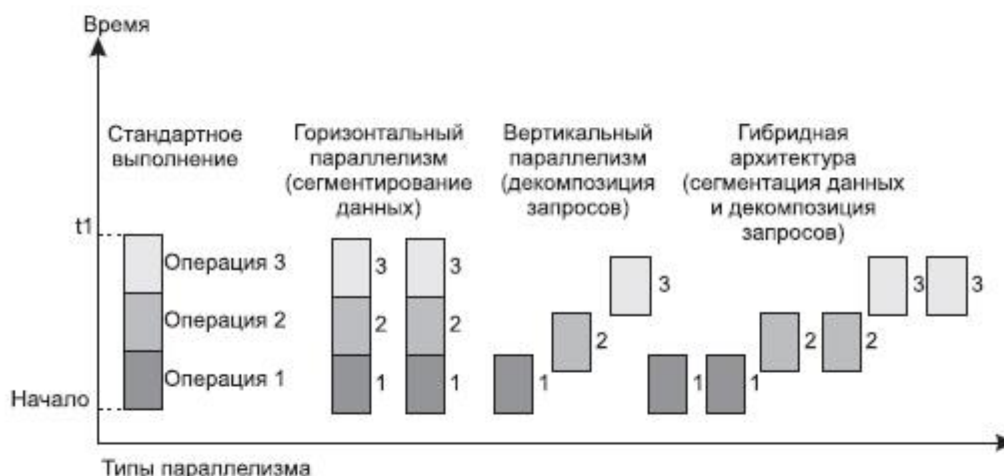

Рис. 12.12. Многонитевая мультисерверная архитектура

12.7. Типы параллелизма

Рассматривают несколько путей распараллеливания запросов.

Горизонтальный параллелизм. Этот параллелизм возникает тогда, когда хранимая в БД информация распределяется по нескольким физическим устройствам хранения — нескольким дискам. При этом информация из одного отношения разбивается на части по горизонтали (рис. 12.13). Этот вид параллелизма иногда называют распараллеливанием или сегментацией данных. И параллельность здесь достигается путем выполнения одинаковых операций, например фильтрации, над разными физическими хранимыми данными. Эти операции могут выполняться параллельно разными процессами, они независимы. Результат выполнения целого запроса складывается из результатов выполнения отдельных операций.

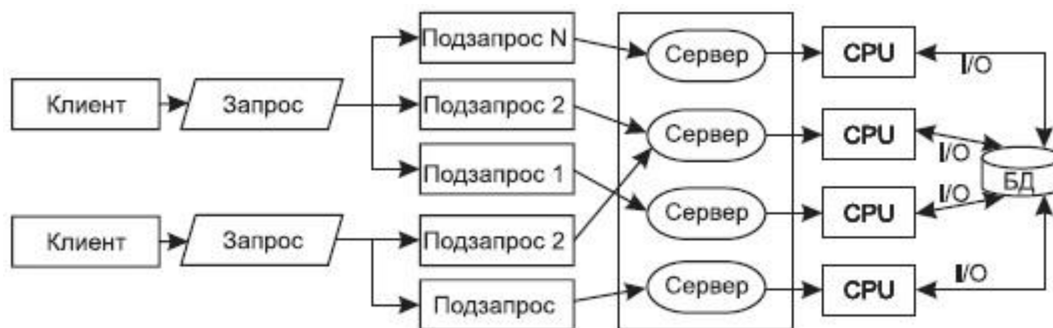

Рис. 12.13. Выполнение запроса при горизонтальном параллелизме

Время выполнения такого запроса при соответствующем сегментировании данных существенно меньше, чем время выполнения этого же запроса традиционными способами одним процессом.

Вертикальный параллелизм. Этот параллелизм достигается конвейерным выполнением операций, составляющих запрос пользователя. Этот подход требует серьезного усложнения в модели выполнения реляционных операций ядром СУБД. Он предполагает, что ядро СУБД может произвести декомпозицию запроса, базируясь на его функциональных компонентах, и при этом ряд подзапросов может выполняться параллельно, с минимальной связью между отдельными шагами выполнения запроса.

Действительно, если мы рассмотрим, например, последовательность операций реляционной алгебры:

```
R5=R1 [A,C]
R6=R2 [A,B,D]
R7 = R5[A > 128]
R8 = R5[A]R6,
```

то операции первую и третью можно объединить и выполнить параллельно с операцией два, а затем выполнить над результатами последнюю четвертую операцию.

Общее время выполнения подобного запроса, конечно, будет существенно меньше, чем при традиционном способе выполнения последовательности из четырех операций (рис. 12.13).

И третий вид параллелизма является гибридом двух ранее рассмотренных (рис. 12.14).

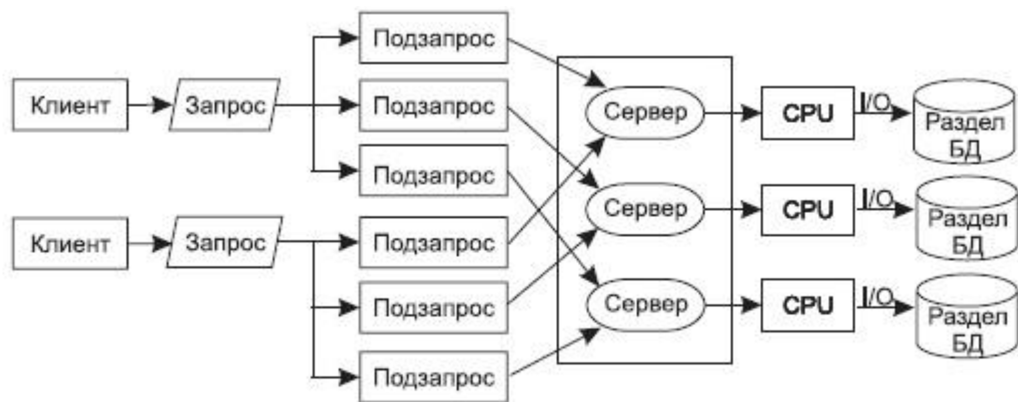

Рис. 12.14. Выполнение запроса при гибридном параллелизме

Наиболее активно применяются все виды параллелизма в OLAP-приложениях, где эти методы позволяют существенно сократить время выполнения сложных запросов над очень большими объемами данных.