МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА»
(САМАРСКИЙ УНИВЕРСИТЕТ)

А. А. АГАФОНОВ, А. С. ЮМАГАНОВ

БЕЗОПАСНОСТЬ СИСТЕМ БАЗ ДАННЫХ

Рекомендовано редакционно-издательским советом федерального государственного автономного образовательного учреждения высшего образования «Самарский национальный исследовательский университет имени академика С. П. Королева» в качестве учебного пособия для обучающихся по основной образовательной программе высшего образования по специальности 10.05.03 Информационная безопасность автоматизированных систем

С А М А Р А
Издательство Самарского университета
2023

УДК 004.056.5(075)
ББК А68я7
А235

Рецензенты: д-р техн. наук, доц. С. Б. П о п о в,
д-р техн. наук, проф. В. А. Ф у р с о в

Агафонов, Антон Александрович

А235 **Безопасность систем баз данных:** учебное пособие / *А.А. Агафонов, А.С. Юмаганов*. – Самара: Издательство Самарского университета, 2023. – 272 с.

ISBN 978-5-7883-1916-2

Учебное пособие посвящено основам безопасности систем баз данных. В пособии рассматриваются задачи построения защищенной базы данных, которая обеспечивает конфиденциальность, доступность и целостность данных пользователя. Для решения задачи построения защищенной базы данных рассматриваются вопросы, связанные с разграничением доступа к данным, вопросы управления доступом к данным, создания учетных записей и настройки процедуры аутентификации, управления привилегиями. Также рассматриваются вопросы обеспечения доступности данных, создание резервных копий баз данных, настройка репликации данных, балансировки нагрузки, секционирования и сегментирования данных, мониторинга доступности баз данных. Отдельные разделы посвящены аудиту и шифрованию данных, а также SQL-инъекциям. Рассматриваются вопросы обеспечения целостности данных с использованием механизма транзакций и встроенных средств СУБД. Для иллюстрации примеров решения данных задач используются СУБД MySQL, PostgreSQL и MongoDB.

Предназначено для студентов института информатики и кибернетики, обучающихся по специальности 10.05.03 Информационная безопасность автоматизированных систем.

УДК 004.056.5(075)
ББК А68я7

ISBN 978-5-7883-1916-2

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Системы баз данных играют ключевую роль в хранении и управлении огромными объемами информации в различных сферах деятельности, от бизнеса и государственных учреждений до личных данных пользователей. Однако, с увеличением количества данных и повышением их важности становится все более актуальным обеспечение безопасности этих систем.

В учебном пособии рассматриваются ключевые аспекты безопасности систем баз данных, связанные с защитой данных, предотвращением несанкционированного доступа и обеспечением конфиденциальности и целостности информации.

В первой части пособия представлены основные угрозы, которые могут возникать в системах баз данных, включая несанкционированный доступ, внутренние и внешние атаки, утечку данных и вредоносное программное обеспечение. Рассматриваются вопросы управления доступом к данным, аутентификации пользователей и управления привилегиями.

Далее рассматриваются вопросы обеспечения доступности данных путем настройки процедур резервного копирования, репликации, сегментирования и секционирования данных.

В заключительной части пособия рассматриваются вопросы аудита и мониторинга систем баз данных, в т.ч. с использованием сторонних сервисов, а также вопросы шифрования и обеспечения целостности данных.

Материал пособия выстроен таким образом, чтобы обучающиеся имели возможность познакомиться с основными проблемами и способами обеспечения безопасности систем баз данных на примерах СУБД MySQL, PostgreSQL и MongoDB.

1 ВВЕДЕНИЕ В БЕЗОПАСНОСТЬ СИСТЕМ БАЗ ДАННЫХ

1.1 Понятие защищенной базы данных

Актуальность проблемы защиты компьютерной информации вряд ли вызывает сомнение. Новости о взломе той или иной системе приходят практически ежедневно. При этом утрата или хищение данных из базы данных наиболее опасна, так как может затрагивать тысячи, а иногда и миллионы пользователей.

В основе многих настольных приложений и практически всех интернет-сервисов лежит некоторая база данных (БД), которая хранит данные о пользователях и сопутствующие данные предметной области (например, системы управления кадрами предприятия, системы бухгалтерского учета, интернет-магазины и т.д.). Очевидно, что разработчикам и администраторам этих систем необходимо обеспечить бесперебойное функционирование этих приложений и сервисов, а также сохранность данных и доступ к данным исключительно согласно принятой политики безопасности. При несоблюдении этих очевидных требований возможна утечка персональных данных (например, данные клиентов финансовой организации, медицинская информация о пациентах клиники и т.д.), утечка данных, составляющих коммерческую тайну (клиентская база, спецификация оборудования, которое еще не поступило в продажу и т.д.), значительное замедление либо полная неработоспособность сервиса или приложения (например, невозможность запустить приложение из-за поломки жесткого диска на сервере БД, невозможность выполнить перевод в интернет-банке из-за недоступности сервера т.д.). В любом случае подробного рода события наносят как финансовый ущерб, так и ущерб деловой репутации организации, а также могут привести к неблагоприятным последствиям для клиентов организации.

Таким образом, можно сказать, что информационная система, частью которой является система управления базами данных (СУБД), должна обеспечивать конфиденциальность, целостность и доступность данных. Более того, должны выполняться все указанные аспекты информационной безопасности, иначе при невыполнении хотя бы одного из них безопасную информационную систему построить не удастся.

Безопасность данных – это состояние защищенности, при котором обеспечиваются конфиденциальность, доступность и целостность данных [1-4]:

– конфиденциальность отвечает за обеспечение доступа к данным, только санкционированным пользователями;

– целостность исключает несанкционированное изменение структуры и содержания данных;

– доступность позволяет обеспечить доступ к данным, санкционированным пользователями, по их первому требованию.

Конфиденциальность информации – необходимость предотвращения разглашения или утечки какой-либо информации. Конфиденциальность направлена на сохранение в тайне данных, критичных для организации, и на обеспечение неприкосновенности личных данных.

Конфиденциальность информации достигается предоставлением к ней доступа с наименьшими привилегиями исходя из принципа минимальной необходимой осведомлённости (англ. need-to-know). Иными словами, авторизованное лицо должно иметь доступ только к той информации, которая ему необходима для исполнения своих должностных обязанностей.

Одной из важнейших мер обеспечения конфиденциальности является классификация информации, которая позволяет отнести её к строго конфиденциальной, предназначенной для публичного или внутреннего пользования. Шифрование информации – характерный пример одного из средств обеспечения конфиденциальности.

Целостность информации – термин, означающий, что данные не были изменены при выполнении какой-либо операции над ними, будь то передача, хранение или отображение. Осуществление деятельности в организации возможно лишь на основе достоверных данных. Иными словами, информация должна быть защищена от намеренного, несанкционированного или случайного изменения по сравнению с исходным состоянием, а также от каких-либо искажений в процессе хранения, передачи или обработки.

Помимо преднамеренных действий, во многих случаях неавторизованные изменения важной информации возникают в результате технических сбоев или человеческих ошибок по невнимательности или из-за недостаточной профессиональной подготовки. Например, к нарушению целостности ведут: случайное удаление файлов, ввод ошибочных значений, изменение настроек, выполнение некорректных команд, причём, как рядовыми пользователями, так и системными администраторами.

Для защиты целостности информации необходимо применение разнообразных мер контроля и управления изменениями информации и обрабатывающих её систем. Типичным примером таких мер является ограничение круга лиц с правами на изменения лишь теми, кому такой доступ необходим для выполнения служебных обязанностей.

Кроме того, любые изменения в ходе жизненного цикла информационных системы должны быть согласованны, протестированы на предмет обеспечения информационной целостности и внесены в систему только корректно сформированными транзакциями.

Потеря целостности грозит искажением или даже разрушением хранимых в базе данных. И то, и другое как минимум приводит к остановке работы базы данных на период восстановления данных.

Доступность информации – состояние информации, при котором субъекты, имеющие права доступа, могут реализовывать их

беспрепятственно. К правам доступа относятся: право на чтение, изменение, хранение, копирование, уничтожение информации.

Основными факторами, влияющими на доступность информационных систем, являются DoS-атаки, атаки программ-вымогателей, саботаж. Кроме того, источником угроз доступности являются непреднамеренные человеческие ошибки, отказ в обслуживании в результате превышения допустимой мощности или недостатка ресурсов оборудования и другие факторы.

Во всех случаях конечный пользователь теряет доступ к информации, необходимой для его деятельности, возникает вынужденный простой. Критичность системы для пользователя и её важность для организации в целом определяют степень воздействия времени простоя. Недостаточные меры безопасности увеличивают риск поражения вредоносными программами, уничтожения данных, проникновения извне или DoS-атак. Подобные инциденты могут сделать системы недоступными для обычных пользователей.

Конфиденциальность, целостность и доступность составляют минимальный набор требований к безопасности данных. Эти требования могут быть дополнены критериями защищенности, например, аутентичностью (т.е. подлинностью данных), апеллируемостью (т.е. подтверждением авторства), достоверностью и т.д.

1.2 Угроза информационной безопасности

Угрозой информационной безопасности информационной системы (ИС) назовем возможность воздействия на информацию, обрабатываемую в системе, приводящего к искажению, уничтожению, копированию, блокированию доступа к информации, а также возможность воздействия на компоненты информационной системы, приводящего к утрате, уничтожению или сбою функциони-

рования носителя информации или средства управления программно-аппаратным комплексом системы.

Угроза нарушения конфиденциальности данных включает в себя любое умышленное или случайное раскрытие информации, хранящейся в вычислительной системе или передаваемой из одной системы в другую. К нарушению конфиденциальности ведет как умышленное действие, направленное на реализацию несанкционированного доступа к данным, так и случайная ошибка программного или неквалифицированного действия оператора, приведшая к передаче по открытым каналам связи незащищенной конфиденциальной информации.

Угроза нарушения целостности включает в себя любое умышленное или случайное изменение информации, обрабатываемой в информационной системе или вводимой из первичного источника данных. К нарушению целостности данных может привести как преднамеренное деструктивное действие некоторого лица, изменяющего данные для достижения собственных целей, так и случайная ошибка программного или аппаратного обеспечения, приведшая к безвозвратному разрушению данных.

Потеря доступности данных – это отказ в обслуживании, вызванный преднамеренными действиями одного из пользователей (нарушителя), при котором блокируется доступ к некоторому ресурсу со стороны других пользователей (постоянно или на большой период времени). Потеря доступности может быть также вызвана умышленным или неумышленным разрушением данных.

Согласно статистике, около 35% утечек данных приходится на внешних нарушителей и около 65% выполнено с участием сотрудников организации, то же самое можно сказать и о ситуациях отказа в доступе – не всегда такие ситуации являются следствием атак извне, а иногда и вовсе вызваны сбоем оборудования или, например, недооценкой нагрузки на информационную систему. Таким

образом, для обеспечения безопасности информационной системы стоит уделять внимание не только внешним угрозам, а наряду с ними необходимо рассматривать как возможные внутренние угрозы безопасности, так и определенные правила проектирования безопасных информационных систем.

Комплексная система обеспечения информационной безопасности должна строиться с учетом средств и методов, характерных для четырех уровней информационной системы:

– уровня прикладного программного обеспечения, отвечающего за взаимодействие с пользователем;

– уровня системы управления базами данных, обеспечивающего хранение и обработку данных информационной системы;

– уровня операционной системы, отвечающего за функционирование СУБД и иного прикладного программного обеспечения;

– уровня среды доставки, отвечающего за взаимодействие информационных серверов и потребителей информации.

Система защиты должна эффективно функционировать на всех этих уровнях.

Существуют различные варианты классификации угроз информационной системе: по природе возникновения, по источнику угроз, по способу доступа к защищаемому ресурсу, по степени воздействия на систему, по расположению источника и т.д. В контексте защиты базы данных рассматривается вариант классификации по способу осуществления угроз.

Выделяются явные угрозы:

– некорректная реализация механизма защиты;

– некорректная настройка механизма защиты;

– неполнота покрытия каналов доступа к информации средствами защиты.

А также скрытые угрозы:

– нерегламентированные действия пользователя;

– ошибки и закладки в программном обеспечении.

В качестве примера на рисунке 1 продемонстрирована типичная многопользовательская компьютерная система небольшой компании [1]. Пользователи компании объединены в локальную вычислительную сеть, которая включает в себя сервер БД и предоставляет возможность доступа в интернет. Рассмотрим перечисленные угрозы на этом примере.

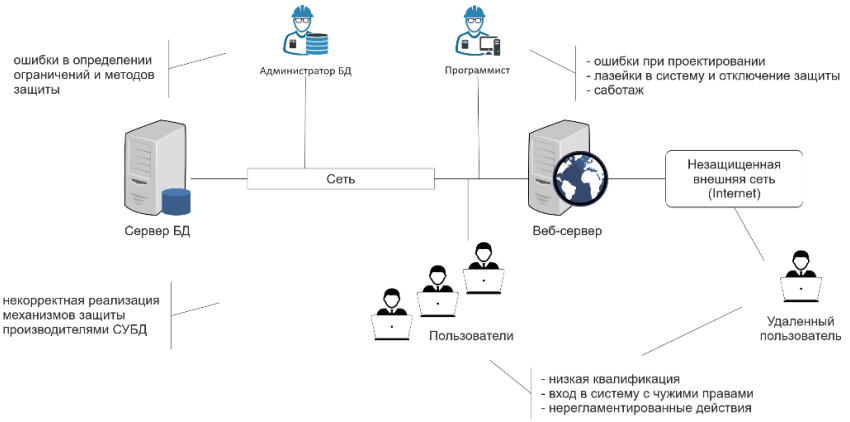

Рисунок 1 – Многопользовательская компьютерная система

Администратор отвечает за защиту компьютерной информации, пытаясь противодействовать всем явным угрозам. Но он сам выступает в качестве источника угроз, ведь его возможные ошибки в настройке механизма защиты компьютерной системы снижают уровень безопасности.

Программисты также являются источником скрытых угроз для системы. Ошибки в исходном коде программ, отключение защиты системы, например, для тестирования, или даже сознательное создание программных закладов влияют на безопасность информационной системы.

В любой компьютерной системе основным действующим лицом является не администратор или программист, а обычный *пользователь*. Низкая квалификация пользователей, обладающих при

этом широкими правами для взаимодействия с базой данных, может значительно влиять на безопасность системы. Примерами действий, представляющих потенциальную угрозу, может являться установка стороннего программного обеспечения из недоверенных источников, переход по ссылкам из спама, запуск программ-вирусов и т.д.

Более того, существуют специальные методы воздействия на СУБД, такие как SQL-инъекции, заключающиеся во внедрении в текст запроса вредоносного кода, загрузка СУБД ресурсоемкими запросами или попытка блокировки таблиц выполнением долгосрочных транзакций.

1.3 Основные принципы обеспечения безопасности

Опыт создания систем защиты позволяет выделить следующие основные принципы построения систем компьютерной безопасности, которые необходимо учитывать при их проектировании и разработке:

- системность;
- комплексность;
- непрерывность защиты;
- разумная достаточность;
- гибкость управления и применения;
- открытость алгоритмов и механизмов защиты;
- простота применения защитных мер и средств.

1.3.1 Принцип системности

Системный подход к защите компьютерных систем предполагает необходимость учета всех взаимосвязанных, взаимодействующих и изменяющихся во времени элементов, условий и факторов, значимых для понимания и решения проблемы обеспечения безопасности автоматизированной системы.

При создании системы защиты необходимо учитывать все слабые, наиболее уязвимые места системы обработки информации, а также характер, возможные объекты и направления атак на систему со стороны нарушителей (особенно высококвалифицированных злоумышленников), пути проникновения в распределенные системы и получения несанкционированного доступа к информации.

Система защиты должна строиться с учетом не только всех известных каналов проникновения и получения несанкционированного доступа к информации, но и с учетом возможности появления принципиально новых путей реализации угроз безопасности.

1.3.2 Принцип комплексности

Комплексное использование широкого спектра мер, методов и средств защиты компьютерных систем предполагает согласованное применение разнородных средств при построении целостной системы защиты, перекрывающей все существенные каналы реализации угроз и не содержащей слабых мест на стыках отдельных ее компонентов.

Комплексная защита информационной системы должна обеспечиваться физическими средствами, организационными и правовыми мерами и использовать средства защиты, реализованные как на уровне операционных систем, так и на прикладном уровне с учетом особенностей предметной области.

1.3.3 Принцип непрерывности защиты

Защита информации – это не разовое мероприятие и даже не определенная совокупность проведенных мероприятий и установленных средств защиты, а непрерывный целенаправленный процесс, предполагающий принятие соответствующих мер на всех этапах жизненного цикла автоматизированной системы, начиная с самых ранних стадий проектирования, а не только на этапе ее эксплуатации.

Разработка системы защиты должна вестись параллельно с разработкой самой защищаемой системы. Это позволит учесть требования безопасности при проектировании архитектуры и, в конечном счете, позволит создать более эффективные (как по затратам ресурсов, так и по стойкости) защищенные системы.

Большинству физических и технических средств защиты для эффективного выполнения своих функций необходима постоянная организационная (административная) поддержка (своевременная смена паролей, обеспечение правильного хранения ключей шифрования, переопределение полномочий и т.п.). Перерывы в работе средств защиты могут быть использованы злоумышленниками для анализа применяемых методов и средств защиты, для внедрения специальных программных и аппаратных «закладок» и других средств преодоления системы защиты после восстановления ее функционирования.

1.3.4 Принцип разумной достаточности

Создать абсолютно непреодолимую систему защиты принципиально невозможно. При достаточном количестве времени и средств можно преодолеть любую защиту. Поэтому имеет смысл вести речь только о некотором приемлемом уровне безопасности. Высокоэффективная система защиты стоит дорого, использует при работе существенную часть мощности и ресурсов компьютерной системы и может создавать ощутимые дополнительные неудобства пользователям. Важно правильно выбрать тот достаточный уровень защиты, при котором затраты, риск и размер возможного ущерба были бы приемлемыми (задача анализа риска).

1.3.5 Принцип гибкости системы защиты

Часто приходится создавать систему защиты в условиях большой неопределенности. Поэтому принятые меры и установленные

средства защиты, особенно в начальный период их эксплуатации, могут обеспечивать как чрезмерный, так и недостаточный уровень защиты. Естественно, что для обеспечения возможности варьирования уровнем защищенности, средства защиты должны обладать определенной гибкостью. Особенно важным это свойство является в тех случаях, когда установку средств защиты необходимо осуществлять на работающую систему, не нарушая процесса ее нормального функционирования. Кроме того, внешние условия и требования с течением времени меняются. В таких ситуациях свойство гибкости спасает от необходимости принятия кардинальных мер по полной замене средств защиты на новые.

1.3.6 Принцип открытости алгоритмов и механизмов защиты

Суть принципа открытости алгоритмов и механизмов защиты состоит в том, что защита не должна обеспечиваться только за счет секретности структурной организации и алгоритмов функционирования ее подсистем. Знание алгоритмов работы системы защиты не должно давать возможности ее преодоления (даже автору). Однако это вовсе не означает, что информация о конкретной системе защиты должна быть общедоступна.

1.3.7 Принцип простоты применения средств защиты

Механизмы защиты должны быть интуитивно понятны и просты в использовании. Применение средств защиты не должно быть связано со знанием специальных языков или с выполнением действий, требующих значительных дополнительных трудозатрат при обычной работе законных пользователей, а также не должно требовать от пользователя выполнения рутинных малопонятных ему операций (ввод нескольких паролей и имен и т.д.).

1.4 Особенности систем баз данных как объекта защиты

Отличительной особенностью систем БД от остальных видов прикладного программного обеспечения (ПО) является их двойственная природа. С этой точки зрения СУБД включает в себя два компонента: хранимые данные (собственно БД) и программы управления (СУБД).

Обеспечение безопасности хранимой информации, в частности, невозможно без обеспечения безопасного управления данными. Исходя из этой концепции, все уязвимости и вопросы безопасности СУБД можно разделить на две категории: зависимые от данных и независимые от данных.

Отметим, что уязвимости, независимые от данных (их структуры, организации и т.д.), являются характерными для всех прочих видов ПО. К этой группе можно отнести несвоевременное обновление ПО или наличие неиспользуемых функций.

Зависимыми от данных (в той или иной степени) является большое число аспектов безопасности. В частности, зависимыми напрямую можно назвать механизмы логического вывода и агрегирования данных.

В то же время многие уязвимости являются косвенно зависимыми от данных. Например, современные СУБД (считая и реляционные, и нереляционные решения) поддерживают запросы к данным с использованием некоторого языка запросов. В свою очередь, в этом качестве используются специализированные языки запросов (SQL, CQL, OQL и других), наборы доступных пользователю функций (которые, в свою очередь, тоже можно считать операторами запросного языка) или произвольные функции на языке программирования.

Уязвимости общего типа, например, SQL-инъекции, выполняются по-разному в зависимости от синтаксиса и семантики языка

запросов, которые отчасти определяются моделью данных и, следовательно, являются зависимым от данных компонентом.

Таким образом, на основании разделения уязвимостей можно выделить зависимые и независимые от данных меры обеспечения безопасности хранилищ информации. Независимыми от данных можно назвать следующие требования к безопасной системе БД:

– функционирование в доверенной среде. Под доверенной понимается информационная среда, интегрирующая совокупность защитных механизмов, которые обеспечивают обработку информации без нарушения политики безопасности. В данном случае СУБД должна функционировать в доверенной информационной системе с соответствующими методами обмена данными;

– организация физической безопасности файлов данных. Данный вопрос требует более детального изучения, так как применяемые структуры данных в различных моделях данных СУБД могут иметь значение при шифровании и защите файлов данных. Однако в первом приближении вопрос физической безопасности файлов данных сходен с вопросом физической безопасности любых других файлов пользователей и приложений;

– организация безопасной и актуальной настройки СУБД. К данному аспекту относятся такие общие вопросы обеспечения безопасности, как своевременная установка обновлений, отключение неиспользуемых модулей или применение эффективной политики паролей.

Следующие требования можно назвать зависимыми от данных:

– безопасность пользовательского слоя ПО. К этой категории относятся задачи построения безопасных интерфейсов и вызовов (в том числе с учетом интерфейса СУБД и механизма доступа к данным);

– безопасная организация данных и манипулирование ими. Данный вопрос является ключевым в системах хранения информации. Именно в эту область входят задачи организации данных с контролем целостности, обеспечение защиты от логического вывода и другие, специфичные для СУБД проблемы безопасности. Фактически эта задача включает в себя основной пул зависимых от данных уязвимостей и защиты от них.

1.4.1 Угрозы безопасности баз данных

В соответствии с изложенной выше задачей обеспечения информационной безопасности СУБД можно выделить следующие угрозы безопасности баз данных:

– несанкционированный доступ к данным, структуре данных или к конфигурации безопасности БД, а также удаление или повреждение данных в результате эксплуатации уязвимостей в клиентских приложениях БД (решение: администрирование прав доступа, правила написания клиентского ПО);

– несанкционированный доступ к данным, структуре данных или к конфигурации безопасности БД, а также удаление или повреждение данных в результате эксплуатации уязвимостей в клиентских приложениях БД (решение: администрирование прав доступа, правила написания клиентского ПО);

– потеря данных вследствие аппаратных или программных сбоев серверов БД случайного или преднамеренного характера. Для решения этой проблемы необходимо использовать резервное копирование данных;

– остановка или значительное снижение производительности сервера БД, приводящие к невозможности использования БД по назначению, вызванное большим количеством активных

пользователей или преднамеренными атаками (решение: репликация данных, масштабирование БД);

– снижение производительности сервера БД, приводящее к невозможности использования БД по назначению, вызванное преднамеренными действиями уполномоченных пользователей (решение: средства мониторинга и протоколирования событий);

– беспрепятственный доступ к данным в случае успешной атаки или хищения (решение: шифрование критических данных).

1.4.2 Администрирование СУБД

Для защиты СУБД от рассмотренных угроз необходимо применять комплекс мер, не всегда имеющих прямое отношение к базе данных, например, обеспечить сетевую защиту, защиту от вирусов или обеспечить безопасность операционной системы в целом. Однако многие вопросы обеспечения безопасности решаются именно путем администрирования СУБД, в частности:

– аутентификация и авторизация пользователя;
– криптографическая защита БД;
– резервное копирование данных;
– репликация и балансировка нагрузки;
– аудит событий безопасности БД;
– модернизация системного и прикладного ПО;
– доступ к данным только с использованием представлений и хранимых процедур.

1.4.2.1 Идентификация, аутентификация и авторизация

Любой пользователь (или процесс), получающий доступ к БД, на этапе создания пользовательской сессии подлежит обязательной **идентификации**. Все дальнейшие его действия так или иначе будут требовать предъявления этого идентификатора.

Одним из основных способов обеспечения конфиденциальности и целостности информации в БД выступает механизм аутентификации. **Аутентификация** – это процедура проверки подлинности пользователя (точнее, его идентификатора). Обычно пользователь подтверждает то, что он является именно тем, за кого он себя выдает, путем ввода в систему уникальной (неизвестной другим) информации о себе. В простейшем случае это символьный пароль, но возможен и более сложный подход подтверждения подлинности, в том числе биометрическая аутентификация или электронные способы аутентификации (контактные и бесконтактные смарт-карты, радиочастотные идентификаторы, USB-ключи).

Если пользователь успешно прошел процедуру аутентификации, СУБД осуществляет его авторизацию. **Авторизация** – это процедура предоставления пользователю определенных ресурсов и прав на их использование. Все дальнейшее взаимодействие пользователя с объектами БД строго регламентируется в соответствии с назначенными правами. Пользователи могут обладать разными правами на один и тот же объект, одни могут лишь просматривать данные, другие – только добавлять, третьи – осуществлять чтение, вставку и редактирование.

1.4.2.2 Криптографическая защита

В основе подавляющего большинства криптографических систем данных защитой выступает шифрование. Шифрование – это процесс преобразования открытых данных с использованием специального алгоритма, после чего эти данные не могут быть восстановлены к исходному виду без ключа дешифрования.

Большинство СУБД, помимо шифрования данных в таблицах БД, обеспечивают криптографическую защиту учетных записей пользователя, исключающую их кражу.

Защищенная СУБД должна уметь шифровать: хранящиеся в ней данные (включая служебную информацию), исходный код запросов, хранимых процедур и триггеров, данные, передаваемые к другим компьютерам по незащищенным каналам.

1.4.2.3 Резервное копирование и восстановление

Для минимизации ущерба от вероятной потери данных необходимо регулярно создавать копию БД. В такую копию входят не только собственно данные, но и служебная информация.

Резервной копией называют копию данных, которая может использоваться для восстановления данных в случае возникновения ошибки или для восстановления копии БД на другом сервере.

Сценарии полного и неполного протоколирования изменений предполагают ведение резервной копии журналов транзакций (в первом случае туда отображаются все операции с БД, а во втором – наиболее важные). Благодаря журналу транзакций возможно не просто восстановление последнего состояния БД, но и откат от этого состояния к наиболее приемлемой точке.

Хотя современные СУБД позволяют создавать резервные копии, не прекращая при этом обслуживать пользователей системы, расписание создания резервных копий следует планировать так, чтобы процедура резервирования проводилась в часы наименьших нагрузок.

Резервная копия также должна защищаться. Одним из наиболее надежных способов защиты может стать шифрование данных при создании резервных копий.

1.4.2.4 Репликация и балансировка нагрузки

Серверы баз данных могут работать совместно для обеспечения возможности быстрого переключения на другой сервер в случае отказа первого (отказоустойчивость) или для обеспечения возможности нескольким серверам БД обрабатывать один набор данных (балансировка нагрузки).

Репликацией называется набор технологий копирования и распространения данных и объектов баз данных между базами данных и последующей синхронизации баз данных для поддержания их согласованности.

Фактически использование репликации позволяет осуществить хранение копии одних и тех же данных на нескольких физических серверах в одной сети, каждую такую копию называют репликой. Такой подход позволяет решать ряд практических задач: повышение производительности системы, обеспечение отказоустойчивости, незаметное резервное копирование данных и других задач.

1.4.2.5 Аудит событий безопасности

Аудит событий безопасности БД представляет собой процесс получения и анализа данных о происходящих в системе событиях и степени их соответствия требованиям к защите данных.

В идеале сбор информации о состоянии системы безопасности БД должен осуществляться непрерывно, для этого многие СУБД автоматически ведут журнал аудита. В журнале содержится:

– описание стандартного набора событий (авторизации пользователя, доступа к тем или иным данным и операций с ними; создания, модификации и уничтожения объектов БД; выполнение нештатных SQL-команд и т. д.);

– настраиваемый перечень атрибутов в отдельной записи журнала аудита (даты и времени события, идентификатор пользователя, имя и сетевой адрес компьютера, описание события, связанные с событием объекты, признак успешного или неудачного завершения события).

Журнал аудита сам по себе должен быть защищен от несанкционированного доступа.

1.4.2.6 Безопасный доступ к данным

Рассмотренные выше способы защиты информации с помощью аутентификации, криптозащиты и резервного копирования в том или ином виде имеются во всех компьютерных системах. Но есть и способы защиты, специфичные для СУБД – это использование представлений и хранимых процедур.

Специалисты по безопасности БД рекомендуют разработчикам ограничить предоставление доступа пользователям непосредственно к таблицам и использовать представления. Использование представлений позволяет ограничить набор атрибутов таблицы и записей, доступных пользователям.

В целях защиты от SQL-инъекций доступ к хранимым в сетевых БД данным должен осуществляться не через динамический SQL, а с помощью хранимых процедур.

2 УПРАВЛЕНИЕ ДОСТУПОМ К ДАННЫМ

2.1 Основные понятия

Целью управления доступом является ограничение действий или операций, которые может выполнять легальный пользователь информационной системы. Управление доступом ограничивает ряд действий, которые пользователь может выполнять над ресурсами информационной системы непосредственно, а также то, какими программными компонентами информационной системы он может управлять (запускать, останавливать, конфигурировать). В целом управление доступом направлено на предотвращение действий, которые могут привести к нарушению безопасности. Т.е. одни пользователи базы данных могут иметь права только на чтение определенных данных, но не иметь доступ ко всем данным, другие – иметь возможность редактирования данных и т.д.

Говоря об управлении доступом, выделяют два ключевых понятия – субъект и объект доступа [5].

Субъект доступа (access subject) – лицо или процесс, действия которого регламентируются правилами разграничения доступа.

Объект доступа (access object) – единица информации автоматизированной системы, доступ к которой регламентируется правилами разграничения доступа. Объектами доступа (контроля) в СУБД является практически все, что содержит конечную информацию: таблицы (базовые или виртуальные), представления, а также более мелкие элементы данных: столбцы и строки таблиц, поля строк (значения), а также программируемые объекты базы данных.

Привилегиями доступа называют разрешение на использование определенной услуги управления данными для доступа к объ-

екту данных (например, чтение, запись или исполнение), предоставляемое идентифицированному пользователю.

Управление доступом может основываться на различных политиках, которые, в свою очередь, следуют разным принципам. Выбор политики безопасности важен, так как он влияет на гибкость, удобство и производительность системы. Принципы, на которых основаны политики, следующие:

– принцип минимальных или максимальных привилегий. В соответствии с принципом минимальных привилегий каждый модуль (процесс, пользователь или программа) должны иметь доступ к такой информации и ресурсам, которые минимально необходимы для успешного выполнения его задач. Противоположностью этому является принцип максимальных привилегий, предусматривающий максимальную доступность данных в БД. Очевидно, принцип минимальных привилегий является предпочтительным с точки зрения построения безопасной системы;

– принцип открытой или закрытой системы. В открытой системе разрешен доступ к тем объектам, для которых явно не настроен запрет. В закрытой системе доступ к объекту разрешен только при наличии явного разрешения, в противном случае доступ к объектам запрещен;

– принцип централизованного и децентрализованного администрирования. Этот принцип рассматривает вопрос, кто отвечает за управление привилегиями в модели управления доступом. В варианте централизованного администрирования привилегий доступа одна сущность контролирует доступ ко всем объектам, в то время как в децентрализованной системе различные сущности контролируют доступ к различным объектам.

Для противодействия несанкционированному доступу в большинстве современных СУБД реализована многоуровневая система обеспечения безопасности, включающая три процедуры:

– **идентификация** – сущность процедуры состоит в назначении пользователю (процессу) уникального имени;

– **аутентификация** – это процедура проверки подлинности пользователя, представившего свой идентификатор. Обычно пользователь подтверждает то, что он является именно тем, за кого он себя выдает, путем ввода в систему уникальной (неизвестной другим) информации о себе. Наиболее распространенный способ подтверждения – ввод символьного пароля;

– если пользователь успешно прошел процедуру аутентификации, то сервер осуществляет его **авторизацию** – процедуру предоставления пользователю определенных ресурсов и прав на их использование.

Диапазон полномочий, которые может получить пользователь, достаточно широк – от ограниченного доступа к отдельному столбцу до всеобъемлющих прав относительно всей базы данных. Набор прав (привилегий) назначается администратором БД. Обычно пользователь вводится в группу с заранее предопределенными ролями (администратор данных, администратор БД, пользователь БД, гость). Все дальнейшее взаимодействие пользователя с объектами БД строго регламентируется в соответствии с назначенной ролью. Пользователи могут обладать разными правами на один и тот же объект, одни могут лишь просматривать данные, другие – только добавлять, третьи – осуществлять чтение, вставку и запись.

2.2 Модель управления доступом

Модель управления доступом определяет порядок доступа субъектов к объектам. Для реализации правил и целей этой модели

используются технологии управления доступом и механизмы безопасности. Условно, модель управления доступом определяет правила, по которым объекты базы данных доступны пользователям. Модели доступа используются для разграничения доступа не только в базах данных, а вообще в информационных системах, например, в операционных системах для доступа к файлам, определения прав для выполнения программ и так далее.

В современных системах может быть реализована комбинация дискреционного (или избирательного) разграничения доступа (Discretionary Access Control, DAC), ролевого разграничения доступа (Role Based Access Control, RBAC) и мандатного (или принудительного) разграничения доступа (Mandatory Access Control, MAC). Рассмотрим эти модели более подробно.

2.2.1 Дискреционная модель управление доступом

Дискреционное управление доступом – управление доступом субъектов к объектам на основе списков управления доступом или матрицы доступа.

Дискреционное управление доступом определяет разграничение доступа между поименованными субъектами и поименованными объектами, при этом субъект с определенным правом доступа может передать это право любому другому субъекту. Правила доступа определяют для каждого пользователя и каждого объекта в системе типы доступа, разрешенные пользователю к данному объекту. Фактически такая модель доступа основана на матрице привилегий. Запрос пользователя на доступ к объекту проверяется по указанным правилам; при наличии привилегии доступа, доступ разрешается; в противном случае доступ отклоняется.

Для каждой пары (субъект – объект) должно быть задано явное и недвусмысленное перечисление допустимых типов доступа (читать, писать и т.д.), то есть тех типов доступа, которые являются

санкционированными для данного субъекта (индивида или группы индивидов) к данному ресурсу (объекту).

Как уже говорилось, эти правила можно представить в виде матрицы доступа. Пример такой матрицы продемонстрирован на рисунке 2. В качестве субъектов доступа выступают пользователи СУБД, в качестве объектов доступа – объекты БД, например, таблицы или представления. В данном примере субъект доступа «Пользователь № 1» имеет право доступа только к объекту доступа № 3 на чтение. Субъект «Пользователь № 2» имеет право доступа как к объекту доступа № 1 (на чтение и изменение), так и к объекту доступа № 2 (на чтение). Пользователь 3 не имеет прав доступа к объектам.

	Объект доступа №1	**Объект доступа №2**	**Объект доступа №3**
Пользователь №1	–	–	Чтение
Пользователь №2	Чтение, запись	Чтение	–
Пользователь №3	–	–	–

Рисунок 2 – Матрица доступа в дискреционной модели управления доступом

Использование матрицы доступа основывается на следующих предположениях:

– все субъекты и объекты доступа должны быть однозначно идентифицированы;

– для любого объекта должен быть определён пользователь-владелец;

– владелец обладает определенными правами доступа (может передавать свои права другим);

– в системе существует привилегированный пользователь, обладающий правом полного доступа к любому объекту. Это нужно для того, чтобы исключить возможность недоступных объектов.

Владелец БД является администратором, ему предоставлены все привилегии. Он имеет право регистрации новых пользователей и предоставление им привилегий.

Привилегии делятся на системные и объектные. Системные привилегии позволяют создавать и модифицировать объекты БД. Объектные привилегии позволяют использовать объекты (выполнять запросы выборки, добавления, модификации данных).

Достоинствами дискреционной модели являются наглядность, простота реализации и понимания.

В качестве недостатков дискреционной модели управления доступом применительно к базе данных можно выделить следующие.

– дискреционная модель не позволяет надежно разграничить доступ к различным строкам одной таблицы. Частичное решение проблемы – запрет доступа к таблице и разрешение доступа к отдельным представлениям, созданным на базе этой таблицы и содержащим различные условия выборки строк. Слабость такого решения очевидна: невозможно разграничить доступ к нескольким строкам таблицы, удовлетворяющим одному условию выборки;

– существуют также проблемы разграничения доступа к отдельным столбцам одной таблицы;

– третий (возможно, главный) недостаток дискреционного управления доступом – это отсутствие средств защиты от несанкционированного распространения конфиденциальной информации: ничто не препятствует пользователю, легально получившему доступ (с правом чтения) к таблице с конфиденциальной информацией, сделать эту информацию доступной другим пользователям путем вставки этой информации в другую (например, временную) общедоступную таблицу. Таким образом, субъект, способный читать данные, может передать данные другим субъектам, не авторизованным на чтение

данных, без ведома владельца этих данных. Этот недостаток делает дискреционную систему управления доступом уязвимой для вредоносных атак типа «Троянский конь».

2.2.2 Мандатная модель управления доступом

Мандатное (принудительное) управление доступом определяет разграничение доступа субъектов к объектам, основанное на назначении метки конфиденциальности для информации, содержащейся в объектах, и выдаче официальных разрешений (допуска) субъектам на обращение к информации такого уровня конфиденциальности. Метка конфиденциальности – это элемент иерархически упорядоченного набора.

Например можно рассмотреть следующие уровни: TS – сверхсекретный (top secret), S – секретный (secret), C – конфиденциальный (confidentional) и U – неклассифицированный (unclassified), где $TS > S > C > U$. Пользователям не допускается создавать объекты с уровнем безопасности ниже, чем его собственный.

Контроль доступа в системах обязательной защиты основан на следующих двух принципах:

– чтение вниз (read down): субъект может читать только те объекты, класс доступа которых равен либо меньше его класса доступа;

– запись вверх (write up): субъект может записывать только те объекты, класс доступа которых равен либо выше его класса доступа.

Удовлетворение этих принципов предохраняет конфиденциальную информацию, от копирования в объекты с более низким уровнем допуска.

Правила контроля доступа можно представить в виде диаграммы информационных потоков, изображенной на рисунке 3.

Субъект не может читать информацию объекта, метка конфиденциальности которого выше его собственной, но также ему запрещена запись информации в объекты с более низким уровнем безопасности, что не позволит такому субъекту понизить уровень секретности информации, к которой он получил легальный доступ.

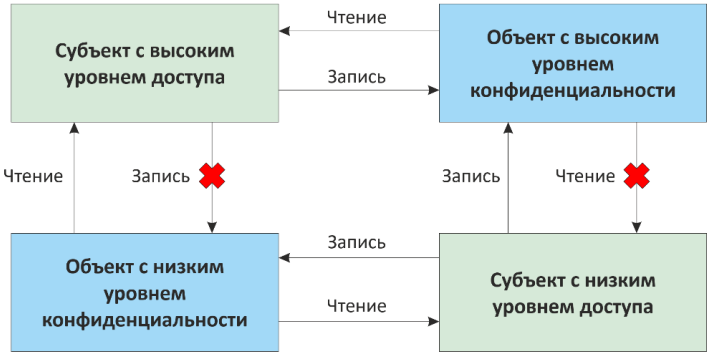

Рисунок 3 – Диаграммы информационных потоков при мандатной модели управления доступом

Хоть данная схема и предусматривает возможность субъектом с низким уровнем доступа записи данных в объект с высоким уровнем секретности, фактически это не работает, так как «секретные» файлы попросту не видны в каталоге.

Мандатная модель доступа не подвержена атаке «Троянский конь», описанной при рассмотрении недостатков дискреционной модели, т.к. согласно рассмотренной диаграмме субъект доступа, даже прочитав защищенные данные, не имеет прав на их запись в объект с низким уровнем доступа.

Из недостатков мандатной модели можно отметить снижение эффективности работы информационной системы, так как проверка доступа к объектам осуществляется не только при открытии объектов, но и перед выполнением любой операции.

Отличие от дискреционной модели состоит в том, что метки секретности хранятся вместе с объектами, а не отдельно. Здесь контролируется не операция, а потоки информации, которые могут быть от субъекта к объекту, либо от объекта к субъекту.

2.2.3 Ролевая модель управления доступом

Управление доступом на основе ролей – развитие политики избирательного управления доступом, при этом права доступа (привилегии) субъектов системы на объекты группируются с учётом специфики их применения, образуя роли. Ролью называется именованная совокупность привилегий, которые могут быть предоставлены пользователям или другим ролям.

Формирование ролей позволяет определить чёткие и понятные для пользователей правила разграничения доступа, а также реализовать гибкие механизмы управления привилегиями. В отличие от дискреционной модели привилегии доступа определяются не для субъектов, а для ролей, а субъекты в свою очередь уже получают членство в различных ролях и наследуют определенные для нее привилегии. Множества субъектов, ролей и привилегий связаны по типу «многие ко многим», что позволяет сформулировать следующие утверждения, характеризующую ролевую модель:

- один субъект может иметь несколько ролей;
- одну роль могут иметь несколько субъектов;
- одна роль может иметь несколько разрешений;
- одно разрешение может принадлежать нескольким ролям.

Разница между подходами продемонстрирована на примере, представленном на рисунке 4.

Несмотря на то, что роль является совокупностью привилегий доступа на объекты, ролевое управление доступом отличается от дискреционного. Отличие заключается в том, что порядок предоставления доступа субъекту определяется в зависимости от имеющихся

(или отсутствующих) у него ролей в каждый момент времени, что скорее характерно мандатной модели управления доступом. С другой стороны, правила ролевого разграничения доступа являются более гибкими, чем при мандатном подходе к разграничению.

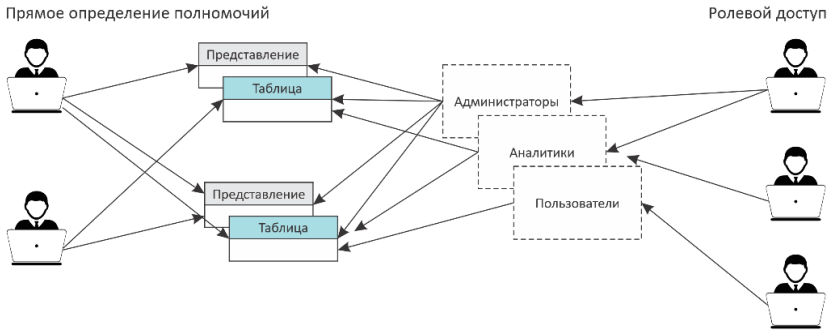

Рисунок 4 – Дискреционная и ролевая модели управления доступом

Использование ролевой модели во многом упрощает разграничение прав доступа и по своему определению наилучшим образом подходит для реализации принципа наименьших привилегий, а также четкого и прозрачного разграничения полномочий.

Так как привилегии не назначаются пользователям непосредственно и приобретаются ими только через свою роль (или роли), управление индивидуальными правами пользователя, по сути, сводится к назначению ему ролей. Это упрощает такие операции, как добавление пользователя или смена подразделения пользователем.

Так, например, можно определить роли «Администратор», «Аналитик», «Пользователь» и в дальнейшем управлять доступом, используя роли, а не адресные привилегии доступа к объектам.

В ролевой модели также вводится понятие «сессия». Сессия – это подмножество ролей, которые активировал пользователь после входа в систему.

По сути, управление доступом состоит из следующих стадий:

– для каждой роли создается набор полномочий, которые предоставляют пользователю определенные права;
– каждому пользователю предоставляется список ролей;
– после авторизации пользователя создается сессия.

Далее, в зависимости от доступных ролей в сессии, проверяются права доступа к объектам.

2.2.4 Смешанная модель управления доступом

В обычных версиях СУБД в настоящее время поддерживается смешанная модель управления доступом, основанная на ролевой и дискреционной моделях. Такой подход совмещает гибкость и удобство управления доступом на уровне ролей с возможностью точного управления на уровне пользователя.

Средства мандатной защиты предоставляются специальными (trusted) версиями СУБД:

– в Oracle Database существует подсистема Oracle Label Security (LBAC, Label-Based Access Control system);
– в PostgreSQL в версии 9.2 появилась начальная поддержка SELinux.

2.3 Управление привилегиями средствами языка SQL

Каждая СУБД должна поддерживать механизм, гарантирующий, что доступ к базе данных смогут получить только те пользователи, которые имеют соответствующее разрешение. Язык SQL включает операторы `GRANT` и `REVOKE`, предназначенные для организации разграничения доступа к объектам базы данных. Механизм защиты построен на использовании идентификаторов пользователей и предоставляемых им прав владения и привилегий.

Идентификатором пользователя называется обычный идентификатор языка SQL, применяемый для обозначения некоторого пользователя базы данных. Каждому пользователю должен быть назначен собственный идентификатор, присваиваемый администратором базы данных. Каждый выполняемый СУБД SQL-оператор выполняется от имени какого-либо пользователя. Идентификатор пользователя определяет, на какие объекты базы данных пользователь может ссылаться и какие операции с этими объектами он имеет право выполнять.

Каждой создаваемой в БД роли (так же, как и пользователю) назначается уникальный идентификатор.

Предоставление прав на защищаемый объект санкционированному пользователю или роли осуществляется с помощью инструкции GRANT. Использование указанной команды гарантирует, что пользователь или роль имеет право выполнить определенные операции с объектом. Так как в СУБД полными правами на объект всегда владеет его создатель, то и право вызова GRANT обычно предоставляется владельцу объекта. Оператор GRANT имеет следующий формат:

```
GRANT {привилегия на объект [,…] | имя роли [,…]}
ON имя объекта
TO {получатель привилегии [,…]}
[WITH GRANT OPTION | WITH ADMIN OPTION]
```

Параметр «имя объекта» может использоваться как имя таблицы базы данных, представления, имя хранимой процедуры, имя внешней процедуры, имя функции. Параметр «получатель привилегии» определяет список, состоящий из идентификаторов пользователей, групп или ролей.

Благодаря параметру WITH GRANT OPTION указанные в операторе GRANT пользователи имеют право передавать все предоставленные им в отношении указанного объекта привилегии другим

пользователям, которые, в свою очередь, будут наделены точно таким же правом передачи своих полномочий. Если данный параметр не будет указан, получатель привилегии не сможет передать свои права другим пользователям. Таким образом, владелец объекта может четко контролировать, кто получил право доступа к объекту и какие полномочия ему предоставлены.

В общем случае набор привилегий зависит от реализации СУБД и определяется производителем. Но обычно выделяют следующие виды привилегий, указанные в таблице 2.1. Ключевое слово `ALL PRIVILEGES` позволяет предоставить указанному пользователю все существующие привилегии без необходимости их перечисления.

Таблица 2.1. Виды привилегий

Привилегия	**Описание**	**Применимо к объектам**
`ALL PRIVILEGIES`	Назначить все привилегии	Ко всем объектам
`SELECT | INSERT | UPDATE | DELETE`	Право на просмотр, вставку, редактирование и удаление данных в таблице (столбце)	Таблицы, столбцы, представления (только `SELECT`)
`REFERENCES`	Право управления ограничением внешнего ключа (`FOREIGIN KEY`), право использовать столбцы в любом ограничении	Таблицы и столбцы
`USAGE`	Дает право использовать данный объект для определения другого объекта	Домены, пользовательские типы данных, наборы символов, порядки сравнения и сортировки, трансляции
`UNDER`	Право на создание подтипов или объектных таблиц	Структурные типы

Окончание табл. 2.1

Привилегия	**Описание**	**Применимо к объектам**
TRIGGER	Право на создание триггера	Таблицы
EXECUTE	Запуск на выполнение	Хранимые процедуры и функции

Среди привилегии создания/изменения объектов БД приведем в таблице 2.2 наиболее часто используемые. В реализациях могут присутствовать и другие разновидности привилегий.

Таблица 2.2. Привилегии создания/изменения объектов в БД

Привилегия	**Описание**
CREATE <тип объекта>	Создание объекта некоторого типа
ALTER <тип объекта>	Изменение структуры объекта
DROP <тип объекта>	Удаление объекта

В языке SQL для отмены привилегий, предоставленных пользователям посредством оператора GRANT, используется оператор REVOKE. С помощью этого оператора могут быть отменены все или некоторые из привилегий, полученных указанным пользователем раньше. Оператор REVOKE имеет следующий формат:

```
REVOKE [GRANT OPTION FOR] {привилегия на объект
[,…] | имя роли [,…]}
ON имя объекта
FROM {получатель привилегии [,…]}
[RESTRICT | CASCADE]
```

В таблице 2.3 приведены опции для отзыва привилегий.

Таблица 2.3. Опции для отзыва привилегий

Опция	**Описание**
ALL PRIVILEGIES	Отзываются все привилегии, предоставленные ему ранее тем пользователем, который ввел данный оператор
GRANT OPTION FOR	Отзывается только право передачи привилегии, но не сама привилегия. Без этого указания отзывается и привилегия, и право назначать привилегии.
CASCADE	Отзыв привилегий не только непосредственно у указанного пользователя, но и у всех пользователей, которым он выдавал привилегии используя опцию GRANT OPTION.
RESTRICT	Отзыв привилегий касается только непосредственно пользователя, указанного в операторе REVOKE, но при наличии у этого пользователя делегированных с помощью опции GRANT OPTION привилегий возникнет ошибка. Такой подход позволяет избежать возникновения так называемых «брошенных» привилегий.

Следует отметить, что рекомендуемые стандартом конструкции поддерживаются не всеми СУБД. В MySQL опции RESTRICT и CASCADE не поддерживаются.

Привилегии, которые были предоставлены указанному пользователю другими пользователями, не могут быть затронуты оператором REVOKE. Следовательно, если другой пользователь также предоставил данному пользователю удаляемую привилегию, то право доступа к соответствующей таблице у указанного пользователя сохранится.

В качестве примера рассмотрим ситуацию, приведенную на рисунке 5.

user1	user2	user3	user4	user5
`GRANT SELECT ON orders TO user2 WITH GRANT OPTION;`	Получение права от `user1`			
	`GRANT SELECT ON orders TO user3 WITH GRANT OPTION;`	Получение права от `user2`. Получение права от `user5`.		`GRANT SELECT ON orders TO user3 WITH GRANT OPTION;`
		`GRANT SELECT ON orders TO user4;`	Получение права от `user3`.	
`REVOKE SELECT ON orders FROM user2 CASCADE`	Отмена права	Сохранение права	Сохранение права	Сохранение права

Рисунок 5 – Пример отмены привилегий

Пусть пользователи `user1` и `user5` имели право на чтение таблицы `orders`, т.е. право `SELECT` на таблицу `orders`. Рассмотрим следующие этапы:

1) пользователь `user1` предоставил пользователю `user2` привилегию `SELECT` для таблицы `orders` с указанием опции `WITH GRANT OPTION`;
2) пользователь `user2` передал эту привилегию пользователю `user3`;
3) пользователь `user3` получил ту же привилегию от пользователя `user5`;
4) пользователь `user3` предоставил привилегию пользователю `user4`.

Когда пользователь `user1` отменяет привилегию `SELECT` для пользователя `user2`, она не может быть отменена и для пользователя `user3`, поскольку ранее он уже получил ее от пользователя

user5. Если бы пользователь user5 не предоставил данной привилегии пользователю user3, то удаление привилегии пользователя user2 имело бы следствием каскадное удаление привилегий для пользователей user3 и user4.

Разрешения, предоставленные роли или группе, наследуются их членами. Хотя пользователю может быть предоставлен доступ через членство в одной роли, роль другого уровня может иметь запрещение на действие с объектом. В таком случае возникает конфликт доступа.

При разрешении конфликтов доступа руководствуются следующим принципом: разрешение на предоставление доступа имеет самый низкий приоритет, а на запрещение доступа – самый высокий. Это значит, что доступ к данным может быть получен только явным его предоставлением при отсутствии запрещения доступа на любом другом уровне иерархии системы безопасности.

3 УЧЕТНЫЕ ЗАПИСИ. АУТЕНТИФИКАЦИЯ

Как было сказано в предыдущих разделах, для противодействия несанкционированному доступу в большинстве современных СУБД реализована многоуровневая система обеспечения безопасности, включающая три процедуры: идентификацию, аутентификацию и авторизацию.

Таким образом, прежде чем пользователь будет авторизован на доступ к определенным данным и на выполнение определенных действий в базе данных, необходимо, что бы он получил доступ к экземпляру СУБД, что в свою очередь требует, чтобы пользователь был аутентифицирован.

В случае успешной аутентификации для данного пользователя устанавливается соединение, и все действия и запросы к данным, выполняемые в рамках этого соединения, контролируются системой авторизации в соответствии с привилегиями, определенными для этого пользователя. В случае, если аутентификация не пройдена, соединение не будет установлено, у пользователя не будет возможности работать с СУБД и, соответственно, с размещенными в ней базами данных.

В контексте баз данных аутентификация пользователя может происходить на разных уровнях (различные методы аутентификации). Она может выполняться на уровне СУБД, на уровне операционной системы, либо на уровне других внешних систем аутентификации. Доступные методы аутентификации зависят от конкретной СУБД. Общей рекомендацией может являться выбор наиболее защищенного метода аутентификации из доступных для данного экземпляра СУБД.

При развертывании СУБД обычно задается пользователь, который имеет неограниченные привилегии. В MySQL это пользователь «root», в PostgreSQL – пользователь «postgres». Разумеется, этих

пользователей необходимо защищать надежными паролями и использовать исключительно в задачах администрирования сервера БД. Учетные данные этих пользователей должны быть известны только администраторам сервера баз данных, строго не рекомендуется передавать учетные данные этих пользователей каким-либо другим пользователям информационной системы, разработчикам программного обеспечения, выполнять от имени этих пользователей запросы от клиентского программного обеспечения и т.д.

Далее рассмотрим способы аутентификации на примере СУБД MySQL [6, 7], PostgreSQL [8, 9] и MongoDB [10, 11].

3.1 Аутентификация в MySQL

Для создания пользователей и ролей в MySQL используются операторы `CREATE USER` или `CREATE ROLE`, которые имеют следующий синтаксис.

```
CREATE USER [IF NOT EXISTS]
    user [auth_option]
    DEFAULT ROLE role [, role ]
    [REQUIRE {NONE | tls_option
            [[AND] tls_option] ...}]
    [WITH resource_option [resource_option] ...]
    [password_option | lock_option]

CREATE ROLE [IF NOT EXISTS] role
```

Оператор `CREATE USER` создает новые учетные записи MySQL и позволяет:

- устанавливать тип аутентификации;
- роль;
- SSL/TLS шифрование;
- ограничивать использование ресурсов;
- управлять паролями.

Оператор также определяет, будут ли учетные записи изначально заблокированы или разблокированы.

`CREATE ROLE` создает одну или несколько ролей, которые называются наборами привилегий. Созданная роль заблокирована, не имеет пароля и ей назначается подключаемый модуль аутентификации по умолчанию. Эти атрибуты роли могут быть изменены позже с помощью инструкции `ALTER USER` пользователями. После назначения пароля и разблокировки аккаунта роль может быть использована для подключения к БД, как и пользователь.

Имена учетных записей MySQL состоят из имени пользователя и имени хоста, что позволяет создавать отдельные учетные записи для пользователей с одинаковым именем пользователя, которые подключаются с разных хостов:

– в соответствии с принятой стратегией аутентификации имя пользователя указывается в строке формата `'username'@'host'`, т.е задается не только имя пользователя, но и узел, с которого он подключается к серверу. Фактически пользователи с одинаковой частью `username` но разной частью `host` – это два различных пользователя, которые могут обладать совершенно разными правами;

– часть `@'host_name'` является необязательной. Имя учетной записи, состоящее только из имени пользователя, эквивалентно `'user_name'@'%'`. Например, `'test_user'` эквивалентно `'test_user'@'%'`;

– при указании хоста кроме конкретных имен узлов и ip-адресов можно использовать значение `'localhost'`, если соединение выполняется с того же компьютера, на котором установлен сервер, а также значения `'%'` и `''`, если соединение допускается с произвольного узла;

– имя пользователя и имя хоста не нужно заключать в кавычки, если они допустимы как идентификаторы без кавычек;

– части имени пользователя и имени хоста, если они указаны в кавычках, должны быть записаны отдельно. Т.е. необходимо указывать пользователя как 'me'@'localhost', а не 'me@localhost'. Имя пользователя 'me@localhost' эквивалентно 'me@localhost'@'%'.

Чтобы узнать, под какой учетной записью вас аутентифицировал сервер, можно вызвать функцию CURRENT_USER():

```
SELECT CURRENT_USER();
```

Оператор ALTER USER изменяет учетные записи MySQL. Оператор поддерживает практически все те же параметры, что и оператор CREATE USER, и позволяет изменять аутентификацию, роль, SSL/TLS шифрование, ограничение ресурсов, управление паролями, комментарии и свойства атрибутов для существующих учетных записей:

```
ALTER USER [IF EXISTS] user [options];
```

Оператор DROP USER удаляет одну или несколько учетных записей MySQL и их привилегии:

```
DROP USER [IF EXISTS] user;
```

Создание нового пользователя с использованием команды CREATE USER добавляет новую запись в системную таблицу mysql.user. Записи из этой таблицы используются для аутентификации пользователей.

В СУБД MySQL пользователь идентифицируется по двум параметрам: имя хоста клиентского приложения и собственно имя пользователя MySQL. Проверка подлинности выполняется на основе трех столбцов (Host, User и authentication_string) таблицы user системной базы данных mysql. Сервер устанавливает соединение только в том случае, если колонки Host и User в

какой-то строке таблицы пользователей совпадают с именем клиентского хоста и именем пользователя, а также клиент предоставляет пароль, совпадающий со значением в столбце `authentication_string`.

MySQL не хранит пароли в открытом виде, непустые значения `authentication_string` в таблице `user` представляют собой зашифрованные пароли. Пароль, предоставляемый пользователем, который пытается подключиться, также зашифрован (с помощью метода хеширования паролей, реализованного плагином аутентификации учетной записи). Зашифрованный пароль используется в процессе подключения при проверке подлинности пользователя. С точки зрения MySQL, зашифрованный пароль – это и есть действительный пароль, поэтому никому не следует давать к нему доступ. В частности, не допускается давать неадминистративным пользователям доступ на чтение данных в системной базе данных `mysql`.

Если сервер аутентифицирует клиента с использованием внешней аутентификации, то пароль в колонке `authentication_string` может не использоваться, в зависимости от использования типа аутентификации.

MySQL поддерживает различные методы аутентификации с использованием плагинов:

– плагин `mysql_native_password`, выполняющий встроенную аутентификацию на основе метода хеширования пароля. Использовался до введения подключаемой аутентификации в MySQL;

– плагин `sha256_password`, выполняющий аутентификацию на основе сравнения SHA-256 хешей паролей. Плагин `caching_sha2_password` также использует SHA-256 аутентификацию, но с кешированием на стороне сервера для лучшей производительности;

– плагин на стороне клиента `mysql_clear_password`, который отправляет пароль на сервер без хеширования или шифрования. Этот подключаемый модуль используется в сочетании с подключаемыми модулями на стороне сервера, которым требуется пароль в открытом виде.

Указанные плагины доступны в свободно распространяемой версии MySQL. Коммерческая версия MySQL также поддерживает внешнюю аутентификацию:

– плагин `authentication_windows`, который выполняет аутентификацию в Windows, позволяя MySQL Server использовать собственные службы Windows для аутентификации клиентских подключений. Пользователи Windows могут подключаться из клиентских программ MySQL к серверу на основе информации в своей среде без указания дополнительного пароля;

– плагин `authentication_ldap_simple`, выполняющий аутентификацию с использованием LDAP (облегчённого протокола доступа к каталогам) путем доступа к службам каталогов, таким как `X.500`;

– плагин `authentication_kerberos`, выполняющий аутентификацию с использованием сетевого протокола аутентификации Kerberos, позволяющего передавать данные через незащищённые сети.

После успешного прохождение процедуры аутентификации сервер будет выполнять проверку назначенных пользователю привилегий на выполнение каждого поступающего запроса.

3.2 Аутентификация в PostgreSQL

В отличие от MySQL, PostgreSQL не выделяет пользователей как отдельные сущности, а использует концепцию ролей для управ-

ления разрешениями на доступ к базе данных. Роль можно рассматривать как пользователя базы данных или как группу пользователей в зависимости от того, как роль настроена. Роли могут владеть объектами базы данных (например, таблицами и функциями) и выдавать другим ролям разрешения на доступ к этим объектам, управляя тем, кто имеет доступ и к каким объектам. Кроме того, можно предоставить одной роли членство в другой роли, таким образом, одна роль может использовать права других ролей.

Концепция ролей включает в себя концепцию пользователей («users») и групп («groups»). До версии 8.1 в PostgreSQL пользователи и группы были отдельными сущностями, но теперь существуют только роли. Любая роль может использоваться в качестве пользователя, группы, и того и другого.

Роли базы данных являются глобальными для всей установки кластера базы данных (не для отдельной базы данных). Для создания роли используется команда SQL `CREATE ROLE`:

```
CREATE ROLE имя [ [ WITH ] параметр [ ... ] ]
Параметр:
      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | CONNECTION LIMIT предел_подключений
    | [ ENCRYPTED ] PASSWORD 'пароль' | PASSWORD
NULL
    | VALID UNTIL 'дата_время'
    | IN ROLE имя_роли [, ...]
    | ROLE имя_роли [, ...]
    | ADMIN имя_роли [, ...]
```

В команде имя соответствует правилам именования идентификаторов SQL: либо обычное, без специальных символов, либо в двойных кавычках. После имени можно указать параметры создаваемой роли.

В частности, можно указать параметр LOGIN, определяющий, разрешается ли новой роли вход на сервер, то есть, может ли эта роль стать начальным авторизованным именем при подключении клиента. Можно считать, что роль с атрибутом LOGIN соответствует пользователю.

Существует также команда CREATE USER, которая является синонимом команды CREATE ROLE:

```
CREATE USER имя [ [ WITH ] параметр [ ... ] ]
...
    | LOGIN | NOLOGIN
```

Единственное отличие между командами состоит в том, что для команды CREATE USER по умолчанию подразумевается использование параметра LOGIN, а в виде CREATE ROLE по умолчанию используется NOLOGIN. Более подробное описание параметров команд представлено в таблице 3.1.

Таблица 3.1. Параметры команд CREATE ROLE и CREATE USER

Параметр	Описание
SUPERUSER | NOSUPERUSER	Определяют, будет ли эта роль «суперпользователем», который может переопределить все ограничения доступа.
CREATEDB | NOCREATEDB	Определяют, сможет ли роль создавать базы данных.
CREATEROLE | NOCREATEROLE	Определяют, сможет ли роль создавать (изменять, удалять) другие роли.

Окончание табл. 3.1

Параметр	**Описание**
`INHERIT | NOINHERIT`	Будет ли роль «наследовать» права ролей, членом которых она является. Роль с атрибутом `INHERIT` может автоматически использовать любые права, назначенные всем ролям, в которые она включена, непосредственно или опосредованно. Иначе права другой роли можно использовать только после переключения на нее командой `SET ROLE`.
`LOGIN | NOLOGIN`	Определяют, разрешается ли новой роли вход на сервер, т.е. может ли эта роль стать начальным авторизованным именем при подключении клиента.
`REPLICATION | NOREPLICATION`	Определяют, будет ли роль ролью репликации с возможностью подключения к серверу в режиме репликации.
`CONNECTION LIMIT`	Определяет, сколько параллельных подключений может установить роль. Значение -1 (по умолчанию) снимает ограничение.
`PASSWORD`	Задаёт пароль роли. Если проверка подлинности по паролю не будет использоваться, этот параметр можно опустить.
`VALID UNTIL`	Устанавливает дату и время, после которого пароль роли перестаёт действовать. Если предложение отсутствует, срок действия пароля неограничен.
`IN ROLE`	Перечисляются одна или несколько существующих ролей, в которые будет включена новая роль.
`ROLE`	Перечисляются одна или несколько существующих ролей, которые автоматически становятся членами создаваемой роли.
`ADMIN`	Подобно `ROLE`, но перечисленные в нём роли включаются в новую роль с атрибутом `WITH ADMIN OPTION`, что даёт им право включать в эту роль другие роли.

Как и MySQL, PostgreSQL реализует аналогичные функции и операторы для получения информации о текущей роли и для модификации ролей. Чтобы узнать, под какой учетной записью вас аутентифицировал сервер, можно вызвать функцию CURRENT_ROLE:

```
SELECT CURRENT_ROLE;
```

Оператор ALTER ROLE изменяет роли и поддерживает те же параметры, что и оператор CREATE ROLE:

```
ALTER ROLE имя [ WITH ] параметр [ ... ];
Оператор DROP ROLE удаляет запись:
DROP ROLE [IF EXISTS] имя;
```

PostgreSQL предлагает несколько различных методов аутентификации клиентов. Метод аутентификации конкретного клиентского соединения может основываться на адресе компьютера клиента, имени базы данных, имени пользователя.

Аутентификация клиентов управляется конфигурационным файлом, который традиционно называется pg_hba.conf и расположен в каталоге с данными кластера базы данных (HBA расшифровывается как host-based authentication – аутентификации по имени узла.)

Формат файла представляет собой набор записей, по одной в строке. Каждая запись обозначает тип соединения, диапазон IP-адресов клиента, имя базы данных, имя пользователя, и способ аутентификации, который будет использован для соединения в соответствии с этими параметрами. Первая найденная запись с соответствующим типом соединения, адресом клиента, указанной базой данных и именем пользователя применяется для аутентификации, если аутентификация не пройдена, последующие записи не рассматриваются. Если же ни одна из записей не подошла, в доступе будет отказано.

В PostgreSQL поддерживаются несколько типов аутентификации:

– простая аутентификация (trust, reject). Метод «trust» безусловно доверяет пользователю и не выполняет аутентификацию. В реальной жизни имеет смысл применять разве что для локальных соединений. Метод «reject» безусловно отказывает в доступе. Можно использовать, чтобы отсечь любые соединения определенного типа или с определенных адресов (например, запретить нешифрованные соединения);

– аутентификация по паролю (password, md5, scram-sha-256, LDAP, RADIUS, PAM, BSD). Эти методы аутентификации запрашивают у пользователя пароль и проверяют его на соответствие паролю, который хранится либо в самой СУБД, либо во внешней службе. Метод «password» сравнивает пароли в открытом виде, метод «md5» сравнивает MD5-хеш пароля с MD5-хешем, хранящимся в базе. Наиболее безопасный метод «scram-sha-256» (появился в версии 10) использует протокол SCRAM и криптостойкий алгоритм SHA-256. Методы «LDAP», «RADIUS» и «PAM» используют для хранения паролей сервер LDAP, сервер RADIUS или подключаемый модуль аутентификации (Pluggable Authentication Module) соответственно;

– внешние системы аутентификации (peer, cert, GSSAPI, SSPI, Ident). Метод подразумевает аутентификацию пользователя внешним сервером аутентификации вне СУБД. В результате успешной аутентификации PostgreSQL получает: имя пользователя, идентифицированное внешней системой (внешнее имя). Внешние имена пользователей могут не совпадать с именами пользователей базы данных, в этом случае, для сопоставления имен используют файл `pg_ident.conf`. Метод peer запрашивает имя пользователя у ядра операционной системы

(ОС). Поскольку ОС уже аутентифицировала этого пользователя, СУБД доверяет ей. Метод cert использует аутентификацию на основе клиентского сертификата и предназначен только для SSL-соединений. Метод GSSAPI использует аутентификацию Kerberos по протоколу GSSAPI. Поддерживается автоматическая аутентификация. Метод SSPI использует аутентификацию Kerberos или NTLM для систем на Windows. Поддерживается автоматическая аутентификация. Метод Ident запрашивает имя пользователя ОС клиента от сервера Ident. Способ доступен только для подключений по TCP/IP.

Примеры записей файла `pg_hba.conf` показаны на рисунке 6. Каждая запись содержит тип соединения, диапазон IP-адресов клиента, имя базы данных, имя пользователя, и способ аутентификации, который будет использован для соединения в соответствии с этими параметрами.

```
# Позволяет любому пользователю локальной системы подключаться к любой базе данных,
# используя любое имя пользователя баз данных, через Unix-сокеты
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
local   all             all                                     trust

# Позволяет любому пользователю компьютера 192.168.12.10 подключаться к базе данных "postgres",
# если он передаёт правильный пароль.
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    postgres        all             192.168.12.10/32        scram-sha-256

# Позволяет любым пользователям с компьютеров в домене example.com
# подключаться к любой базе данных, если передаётся правильный пароль.
#
# Для всех пользователей требуется аутентификация SCRAM, за исключением
# пользователя 'mike', который использует старый клиент, не поддерживающий
# аутентификацию SCRAM.
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             mike            .example.com            md5
host    all             all             .example.com            scram-sha-256
```

Рисунок 6 – Записи файла pg_hba.conf

Рассмотрим несколько примеров создания ролей в PostgreSQL. В следующих создается роль `agafonov`, для которой разрешен

вход с паролем. CREATE USER действует так же, как CREATE ROLE, но подразумевает ещё и атрибут LOGIN, поэтому эти два запроса идентичны.

```
CREATE ROLE agafonov WITH LOGIN PASSWORD 'P@ssw0rd';
CREATE USER agafonov WITH PASSWORD 'P@ssw0rd';
```

В следующем примере создается роль ivanov с паролем, действующим до конца 2022 г.:

```
CREATE ROLE ivanov WITH LOGIN PASSWORD 'pass'
  VALID UNTIL '2023-01-01';
```

В следующем примере создается роль admin, которая может создавать базы данных и управлять ролями:

```
CREATE ROLE admin WITH CREATEDB CREATEROLE;
```

Для управления базами данных и ролями рекомендуется создавать роль с правами CREATEDB и CREATEROLE, но не суперпользователя. Такой подход позволит избежать опасностей, связанных с использованием полномочий суперпользователя для задач, которые их не требуют.

Для получения информации о существующих ролях можно выполнить запросы к таблицам pg_user и pg_roles. Результаты выполнения запросов к этим таблицам показаны на рисунках 7 и 8 соответственно.

```
SELECT * FROM pg_user;
```

	username	usecreatedb	usesuper	userepl	passwd	valuntil
	postgres	true	true	true	*******	null
	agafonov	false	false	false	*******	null
	ivanov	false	false	false	*******	2023-01-01 00:00:00+04

Рисунок 7 – Результат выполнения запроса к таблице pg_user

```
SELECT * FROM pg_roles;
```

	rolname	rolsuper	rolinherit	rolcanlogin	rolpassword	rolvaliduntil
	pg_database_owner	false	true	false	*******	null
	…	…	…	…	…	…
	postgres	true	true	true	*******	null
	agafonov	false	false	false	*******	null
	ivanov	false	false	false	*******	2023-01-01 00:00:00+04

Рисунок 8 – Результат выполнения запроса к таблице pg_roles

3.3 Аутентификация в MongoDB

Свободно распространяемая версия MongoDB Community поддерживает ряд механизмов аутентификации:

– SCRAM (по умолчанию) для аутентификации пользователей по паролю. Поддерживаются методы SCRAM-SHA-1, использующий хеш-функцию SHA-1, и SCRAM-SHA-256, использующий хеш-функцию SHA-256;

– аутентификация на основе сертификата x.509 (вместо имени пользователя и пароля). Цифровой сертификат x.509 использует общепринятый стандарт инфраструктуры открытых ключей (PKI) x.509 для проверки того, что открытый ключ принадлежит тому, кто его предъявляет.

Кроме того, версии MongoDB Atlas и MongoDB Enterprise поддерживают следующие механизмы аутентификации:

– аутентификация с использованием протокола LDAP (облегчённый протокол доступа к каталогам);

– аутентификация с использованием сетевого протокола аутентификации Kerberos.

Эти механизмы позволяют MongoDB интегрироваться в существующую систему аутентификации.

Кратко рассмотрим основные правила настройки аутентификации в Mongo.

Пользователь в MongoDB создается в определенной базе данных. Эта БД является базой данных аутентификации пользователя; для этой цели можно использовать любую базу данных. База данных имени пользователя и аутентификации служит уникальным идентификатором пользователя. Однако привилегии пользователя не ограничиваются базой данных аутентификации.

Сначала создается пользователь-администратор, а затем создаются дополнительные пользователи в БД для каждого пользователя/приложения, которое обращается к системе.

Как и в любых других информационных системах, рекомендуется следовать принципу наименьших привилегий, т.е. сначала создать роли, определяющие точные права доступа, требуемые набором пользователей. Затем создать пользователей и назначить им только те роли, которые необходимы для выполнения их операций.

В отличие от рассмотренных СУБД MySQL и PostgreSQL, в MongoDB аутентификация и авторизация не активируются по умолчанию. MongoDB также не создает пользователя root или пользователя с правами администратора по умолчанию при активации аутентификации и авторизации, независимо от того, какой режим аутентификации используется. Его необходимо создать вручную.

Произведем настройку SCRAM-аутентификации для сервера, запущенного без режима репликации. Настройка аутентификации состоит из следующих шагов.

На *первом шаге* выполняется запуск экземпляра сервера MongoDB без контроля доступа и выполняется подключение к серверу.

На *втором шаге* выполняется создание пользователя-администратора в БД `admin`.

```
use admin
db.createUser( {
```

```
    user: "myUserAdmin",
    pwd: passwordPrompt(), // или указать пароль
    roles: [
   { role: "userAdminAnyDatabase", db: "admin" },
   { role: "readWriteAnyDatabase", db: "admin" }
    ]
} )
```

В запросе выполняется переключение на базу командой `use admin` и создание пользователя `myUserAdmin` с ролями `userAdminAnyDatabase` и `readWriteAnyDatabase`. Роль `userAdminAnyDatabase` позволяет этому пользователю:

- создавать пользователей;
- предоставлять или отзывать роли у пользователей;
- создавать или изменять роли.

При необходимости можно назначить пользователю дополнительные встроенные или пользовательские роли.

База данных, в которой создается пользователь (в представленном примере – `admin`), является базой данных аутентификации пользователя. Однако пользователю необходимо пройти аутентификацию в этой базе данных, так как он может иметь роли в других базах данных. База данных аутентификации пользователя не ограничивать привилегии пользователя.

На *третьем шаге*, после создания пользователя-администратора, необходимо перезапустить экземпляр MongoDB в режиме контроля доступа. Если экземпляр `mongod` запускается из командной строки, необходимо добавить параметр `–auth`:

```
mongod --auth --port 27017 --dbpath /var/lib/mon-
godb
```

Если экземпляр `mongod` запускается с помощью файла конфигурации, необходимо добавить параметр файла конфигурации

security.authorization:

```
security:
    authorization: enabled
```

На *четвертом шаге* после запуска сервера в режиме контроля допуска для аутентификации необходимо либо указать параметры при подключении, либо выполнять аутентификацию после подключения. Для этого необходимо выбрать БД аутентификации и вызвать метод `auth` с указанием имени пользователя и пароля. Пароль можно указать явно в открытом виде, либо с использованием метода `passwordPrompt`, который запросит пароль в закрытом виде:

```
use admin
db.auth("myUserAdmin", passwordPrompt())
// или указать пароль
```

После аутентификации в качестве администратора методом `db.createUser()` можно создавать дополнительных пользователей. Пользователям можно назначать любые встроенные или определяемые пользователем роли, например:

```
use db_lectures
db.createUser( {
    user: "myTester",
    pwd: passwordPrompt(), // или указать пароль
    roles: [
        { role: "readWrite", db: "db_lectures" },
        { role: "read", db: "wikipedia" }
    ]
} )
```

Чтобы переключиться на нового пользователя, можно использовать метод `db.auth`:

```
use db_lectures
db.auth("myTester", passwordPrompt()) // или пароль
```

4 УПРАВЛЕНИЕ ПРИВИЛЕГИЯМИ

Привилегиями доступа называют разрешение на использование определенной услуги управления данными для доступа к объекту данных (например, чтение, запись или исполнение), предоставляемое идентифицированному пользователю.

Ролью в свою очередь называется именованная совокупность привилегий, которые могут быть предоставлены пользователям или другим ролям.

Управление доступом на основе ролей – развитие политики избирательного управления доступом, при этом права доступа (привилегии) субъектов системы на объекты группируются с учётом специфики их применения, образуя роли.

Ролевую модель управления доступом характеризуют следующие утверждения:

- один субъект может иметь несколько ролей;
- одну роль могут иметь несколько субъектов;
- одна роль может иметь несколько разрешений;
- одно разрешение может принадлежать нескольким ролям.

Рассматриваемые СУБД MySQL, PostgreSQL, MongoDB поддерживают ролевую модель управления доступом.

Предоставление прав на защищаемый объект санкционированному пользователю или роли осуществляется с помощью инструкции `GRANT`. Оператор `GRANT` имеет следующий формат:

```
GRANT {привилегия на объект [,…] | имя роли [,…]}
ON имя объекта
TO {получатель привилегии [,…]}
[WITH GRANT OPTION | WITH ADMIN OPTION]
```

Параметр «`имя объекта`» может использоваться как имя таблицы базы данных, представления, имя хранимой процедуры, имя

внешней процедуры, имя функции. Параметр «получатель привилегии» определяет список, состоящий из идентификаторов пользователей, групп или ролей.

Для отмены привилегий, предоставленных пользователям посредством оператора GRANT, используется оператор REVOKE, который имеет следующий синтаксис:

```
REVOKE [GRANT OPTION FOR] {привилегия на объект
[,…] | имя роли [,…]}
ON имя объекта
FROM {получатель привилегии [,…]}
[RESTRICT | CASCADE]
```

В общем случае набор привилегий зависит от реализации СУБД и определяется производителем. В большинстве случаев выделяют следующие виды привилегий:

Таблица 4.1. Виды привилегий

Привилегия	**Описание**	**Применимо**
ALL PRIVILEGIES	Назначить все привилегии	Ко всем объектам
SELECT | INSERT | UPDATE | DELETE	Право на просмотр, вставку, редактирование и удаление данных в таблице (столбце)	Таблицы, столбцы, представления (только SELECT)
REFERENCES	Право управления ограничением внешнего ключа, право использовать столбцы в любом ограничении	Таблицы и столбцы
USAGE	Право использовать данный объект для определения другого объекта	Домены, пользовательские типы данных, наборы символов, порядки сравнения и сортировки, трансляции

Окончание табл. 4.1

Привилегия	**Описание**	**Применимо**
UNDER	Право на создание подтипов или объектных таблиц	Структурные типы
TRIGGER	Право на создание триггера	Таблицы
EXECUTE	Запуск на выполнение	Хранимые процедуры и функции

Стоит отметить, что данные операторы могут поддерживать различные параметры для определения доступных для выдачи разрешений, специфичных для конкретных СУБД.

4.1 Привилегии MySQL

Привилегии MySQL различаются в зависимости от контекста, в котором они применяются, и уровня их работы:

– административные привилегии позволяют пользователям управлять работой сервера MySQL. Эти привилегии являются глобальными, потому что они не связаны с конкретной БД;

– привилегии базы данных применяются к БД и ко всем объектам в ней. Эти привилегии могут быть предоставлены для определенных баз данных или глобально для всех баз;

– привилегии для объектов базы данных, таких как таблицы, индексы, представления и хранимые процедуры, могут предоставляться для конкретных объектов в базе данных, для всех объектов данного типа в базе данных (например, для всех таблиц в базе данных) или глобально для всех объекты данного типа во всех базах данных;

– привилегии столбцов обращаются к одиночным столбцам в указанной таблице.

Для просмотра привилегий пользователя можно выполнить следующую команду:

```
SHOW GRANTS FOR {CURRENT_USER | имя_пользователя};
```

Глобальные привилегии применяются ко всем базам данных на данном сервере.

Административные привилегии, такие как CREATE USER, FILE, PROCESS, RELOAD, REPLICATION CLIENT, REPLICATION SLAVE, SHOW DATABASES, SHUTDOWN и SUPER, могут предоставляться только глобально. Другие привилегии могут быть предоставлены глобально или на более определенных уровнях. Чтобы назначить глобальные привилегии, используется синтаксис ON *.*.

Рассмотрим пример. Создадим пользователя:

```
CREATE USER 'user'@'localhost' IDENTIFIED BY
'P@ssw0rd';
```

Назначим созданному пользователи глобальные привилегии:

```
GRANT ALL ON *.* TO 'user'@'localhost';
GRANT SELECT, INSERT ON *.* TO 'user'@'localhost';
```

Привилегии базы данных применяются ко всем объектам в данной базе данных.

Для назначения привилегии на уровне базы данных, используется синтаксис ON db_name.*:

```
GRANT ALL ON lectures.* TO 'user'@'localhost';
GRANT SELECT, INSERT ON lectures.* TO
'user'@'localhost';
```

Привилегии CREATE, DROP, EVENT, LOCK TABLES и REFERENCES можно задать на уровне базы данных.

Привилегии для таблицы или хранимой процедуры/функции также могут быть указаны на уровне базы данных. В этом случае они

применяются ко всем таблицам или хранимых процедур/функций в базе данных. Привилегии для таблицы применяются ко всем столбцам в данной таблице. Для назначения привилегии на уровне таблицы используется синтаксис `ON db_name.tbl_name`:

```
GRANT ALL ON lectures.movies TO 'user'@'localhost';
GRANT SELECT, INSERT ON lectures.movies TO 'user'@'localhost';
```

Допустимые значения привилегий на уровне таблицы: `ALTER`, `CREATE VIEW`, `CREATE`, `DELETE`, `DROP`, `GRANT OPTION`, `INDEX`, `INSERT`, `REFERENCES`, `SELECT`, `SHOW VIEW`, `TRIGGER` и `UPDATE`.

Привилегии уровня таблицы применяются к базовым таблицам и представлениям. Они не применяются к таблицам, созданным с помощью `CREATE TEMPORARY TABLE`, даже если имена таблиц совпадают.

Для назначения привилегий уровня столбцов для конкретной таблицы указываются конкретные столбцы:

```
GRANT SELECT (data) ON lectures.movies TO 'user'@'localhost';
```

Допустимыми привилегиями на уровне столбца являются `INSERT`, `REFERENCES`, `SELECT` и `UPDATE`.

По стандарту ролью называется именованный набор привилегий. Однако в MySQL, как и в PostgreSQL, роли не ограничиваются определением набора привилегий, а представляют собой те же учетные записи, которые могут быть присвоены другим учетным записям. В связи с этим именование ролей, как и пользователей, подразумевает указание имени и хоста. Однако есть два основных отличия: во-первых, имя роли не может быть пустым, т.е. не бывает анонимных ролей, и во-вторых, если имя хоста для роли не указано,

то оно интерпретируется как `'%'`. По этой причине имена ролей часто задаются, используя только часть имени пользователя без указания имени хоста. Создание роли с явным указанием хоста имеет смысл только в том случае, если планируется создать имя, которое будет работать и как роль, и как учетная запись, которой разрешен вход с указанного узла.

В MySQL существует такое понятие как **обязательные роли**. Сервер интерпретирует обязательную роль как роль, предоставляемую всем пользователям, т.е. без необходимости её явного предоставления какой-либо учетной записи. Указание обязательных ролей выполняется с помощью их перечисления в значении системной переменной `mandatory_roles` в конфигурационном файле сервера.

Как и учетные записи пользователей, роли могут иметь привилегии, предоставленные и отозванные у них. Учетной записи пользователя могут быть назначены роли, которые предоставляют этой учетной записи набор привилегий, связанный с каждой ролью.

Назначение ролей предоставляет удобную альтернативу назначению индивидуальных привилегий пользователям.

Для создания роли используется оператор `CREATE ROLE`, для удаления – `DROP ROLE`:

```
CREATE ROLE [IF NOT EXISTS] role;
DROP ROLE [IF EXISTS] role;
```

Роли, предоставленные учетной записи пользователя, могут быть активными или неактивными в текущем сеансе. Если предоставленная роль активна – применяются ее привилегии, в противном случае – соответственно нет. Чтобы определить, какие роли активны в текущем сеансе, используется функция `CURRENT_ROLE()`:

```
SELECT CURRENT_ROLE();
```

По умолчанию предоставление роли учетной записи не приводит к автоматическому активированию роли в сеансах учетной записи.

Оператор `SET ROLE` изменяет действующие привилегии текущего пользователя в текущем сеансе, указывая, какие из предоставленных ему ролей активны. Предоставленные роли включают роли, явно предоставленные пользователю, и обязательные роли, указанные в значении системной переменной `mandatory_roles`.

Чтобы указать, какие роли должны становиться активными каждый раз, когда пользователь подключается к серверу и проходит аутентификацию или когда пользователь выполняет оператор `SET ROLE DEFAULT` во время сеанса, используется выражение `SET DEFAULT ROLE`.

Установка ролей осуществляется следующим образом:

– установка активной роли в текущем сеансе;

```
SET ROLE {DEFAULT | NONE | ALL | ALL EXCEPT role [,
role] | role [, role] }
```

– установка активной роли по умолчанию.

```
SET DEFAULT ROLE {NONE | ALL | role [, role] } TO
user [, user]
```

MySQL предопределяет набор фиксированных ролей, соответствующих конкретным задачам. Разрешения, предоставленные фиксированным ролям, не могут быть изменены. В таблице 4.2 представлены роли, поддерживаемые MySQL. Назначение этих ролей доступно через раздел администрирования MySQL Workbench.

Таблица 4.2. Роли, поддерживаемые MySQL

Роль	**Описание**
`DBA`	Разрешено выполнять любые действия на сервере.
`MaintenanceAdmin`	Предоставляет права на администрирование сервера.
`ProcessAdmin`	Предоставляет права на доступ, мониторинг и завершение клиентских процессов.
`UserAdmin`	Предоставляет права на создание пользователей и сброс паролей.
`SecurityAdmin`	Предоставляет права на управления учетными записями, а также предоставление и отзыв серверных привилегий.
`MonitorAdmin`	Предоставляет права на для мониторинга сервера.
`DBManager`	Предоставляет полные права на все базы данных.
`DBDesigner`	Предоставляет права на создание и модификацию схемы любой базы данных.
`ReplicationAdmin`	Предоставляет права на настройку и управление репликацией данных.
`BackupAdmin`	Предоставляет минимальные права, необходимые для выполнения резервного копирования любой базы данных.

4.2 Привилегии PostgreSQL

В отличие от MySQL, в PostgreSQL существует понятие «**владельца**» объекта. Когда в базе данных создаётся объект, ему назначается владелец. Владельцем обычно становится роль, с которой был выполнен оператор создания. Для большинства типов объектов в исходном состоянии только владелец (или суперпользователь) может выполнять с объектом любые операции. Чтобы разрешить использовать его другим ролям, нужно назначить им привилегии.

Существует несколько типов прав: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`, `REFERENCES`, `TRIGGER`, `CREATE`, `CONNECT`,

TEMPORARY, EXECUTE и USAGE. Набор прав, применимых к определённому объекту, зависит от типа объекта (таблица, функция и т.д.).

Неотъемлемое право изменять или удалять объект имеет только владелец объекта. Объекту можно назначить нового владельца с помощью команды ALTER для соответствующего типа объекта, например:

```
ALTER TABLE имя_таблицы OWNER TO новый_владелец;
```

Суперпользователь может делать это без ограничений, а обычный пользователь – только если он является одновременно текущим владельцем объекта (или членом роли владельца) и членом новой роли.

В PostgreSQL пользователь и роль – это синонимы. Для получения имени пользователя в текущем контексте выполнения используется команда CURRENT_ROLE. Можно также использовать команду CURRENT_USER. Для получения имени пользователя сеанса используется команда SESSION_USER. Синтаксис команд имеет следующий вид:

```
CREATE ROLE имя [ [ WITH ] параметр [ ... ] ]
ALTER ROLE имя [ WITH ] параметр [ ... ];
DROP ROLE [IF EXISTS] имя;
```

Просмотр текущей роли и сессионной роли.

```
SELECT CURRENT_ROLE, SESSION_USER;
```

Команда SET ROLE используется для установки идентификатора текущего пользователя в рамках сеанса:

```
SET ROLE role | NONE
```

Указывая определённое имя роли, текущий пользователь должен являться членом этой роли. Если пользователь сеанса является суперпользователем, он может выбрать любую роль.

Команда RESET ROLE устанавливает в качестве идентификатора текущего пользователя значение, заданное во время подключения параметрами командной строки. Если же такое значение не задано, в качестве идентификатора текущего пользователя так же устанавливается идентификатор текущего пользователя сеанса:

```
RESET ROLE
```

Часто бывает удобно сгруппировать пользователей для упрощения управления правами: права можно выдавать и забирать для всей группы, как и в MySQL. В PostgreSQL для этого создаётся роль, представляющая группу, а затем членство в этой группе выдаётся ролям индивидуальных пользователей.

Для настройки групповой роли сначала нужно создать саму роль:

```
CREATE ROLE групповая_роль;
```

Обычно групповая роль не имеет атрибута LOGIN, т.е. не рассматривается как учетная запись подключения к БД, хотя при желании его можно установить.

После того как групповая роль создана, в неё можно добавлять или удалять членов, используя команды GRANT и REVOKE:

```
GRANT групповая_роль TO роль1, …;
REVOKE групповая_роль FROM роль1, …;
```

Членом роли может быть и другая групповая роль, потому что в действительности нет никаких различий между групповыми и не групповыми ролями. При этом база данных не допускает замыкания членства по кругу. Также не допускается управление членством роли PUBLIC в других ролях.

Члены групповой роли могут использовать её права двумя способами. Во-первых, каждый член группы может явно выполнить

SET ROLE, чтобы временно «стать» групповой ролью. В этом состоянии сеанс базы данных использует полномочия групповой роли вместо оригинальной роли, под которой был выполнен вход в систему. При этом для всех создаваемых объектов базы данных владельцем считается групповая, а не оригинальная роль. Во-вторых, роли, имеющие атрибут INHERIT, автоматически используют права всех ролей, членами которых они являются, в том числе и унаследованные этими ролями права.

Привилегии доступа к объектам базы данных для ролей также определяются с помощью операторов GRANT и REVOKE. Синтаксис этих операторов позволяет указывать как отдельные, так и все возможные привилегии, как на отдельные объекты, так и группы объектов и т.д., например:

```
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public
TO app_role;
GRANT SELECT(first_name, last_name) ON actor TO
app_role;
GRANT INSERT, UPDATE, DELETE ON actor TO app_role;
```

Доступ роли к объектам базы данных определяется выданными привилегиями, но есть два исключения:

- суперпользователи. Для них проверки разграничения доступа не выполняются и соответственно они имеют доступ ко всем объектам;
- владельцы объектов. Они сразу получают полный набор привилегий для этого объекта, правда эти привилегии могут быть отозваны.

Приведем примеры управления ролями и привилегиями. Сначала создадим роли:

```
CREATE ROLE app_user WITH LOGIN PASSWORD '1';
CREATE ROLE app_developer;
```

```
CREATE ROLE app_read;
CREATE ROLE app_write;
CREATE ROLE app_read_write NOINHERIT;
```

Роль app_user определена с атрибутом LOGIN и паролем, что позволяет использовать ее как учетную запись для подключения к базе данных. Роли app_user, app_developer, app_read и app_write по умолчанию используют модификатор INHERIT, т.е. могут автоматически использовать в базе данных любые права, назначенные всем ролям, в которые они включена, непосредственно или опосредованно. Роль app_read_write определена с атрибутом NOINHERIT, т.е. правами можно будет пользоваться только после выполнения команды SET ROLE и переключения на эту роль.

Далее назначим ролям привилегии:

```
GRANT SELECT ON film TO app_developer;
GRANT SELECT(first_name, last_name) ON actor TO
app_read;
GRANT INSERT, UPDATE, DELETE ON actor TO app_write;
```

Роль app_developer имеет права на чтение из таблицы film, роль app_read – права на чтения столбцов first_name и last_name из таблицы actor, роль app_write – права на модификацию данных из таблицы actor.

Далее назначим роли app_read и app_write роли app_read_write, и роли app_user назначим роли app_developer и app_read_write:

```
GRANT app_developer TO app_user;
GRANT app_read, app_write TO app_read_write;
GRANT app_read_write TO app_user;
```

Для проверки доступных привилегий подключимся к базе данных от имени `app_user`. Эта роль может использоваться как учетная запись, так как она определена с атрибутом `LOGIN`.

Для проверки текущей роли и сессионной роли выполним запрос:

```
SELECT CURRENT_ROLE, SESSION_USER;
```

Результат выполнения запроса показан на рисунке 9.

	current_role	session_user
	app_user	app_user

Рисунок 9 – Результат выполнения запроса

Чтобы в текущем сеансе переключиться на роль app_read, можно воспользоваться командой SET ROLE. Результат выполнения запроса получения текущей роли после ее смены показан на рисунке 10:

```
SET ROLE app_read;
SELECT CURRENT_ROLE, SESSION_USER;
```

	current_role	session_user
	app_read	app_user

Рисунок 10 – Результат запроса после смены текущей роли

В PostgreSQL также существуют фиксированные роли, показанные в таблице 4.3.

Таблица 4.3. Фиксированные роли в PostgreSQL

Роль	**Разрешаемый доступ**
pg_read_all_data	Читать все данные (таблицы, представления, последовательности), как будто роль имеет права SELECT на эти объекты и права USAGE на все схемы, но при этом явным образом такие права ей не назначены.
pg_write_all_data	Записывать все данные (таблицы, представления, последовательности), как будто роль имеет права INSERT, UPDATE и DELETE на эти объекты и права USAGE на все схемы, но при этом явным образом такие права ей не назначены.
pg_read_all_settings	Читать все конфигурационные переменные, даже те, что обычно видны только суперпользователям.
pg_read_all_stats	Читать все представления pg_stat_* и использовать различные расширения, связанные со статистикой.
pg_database_owner	Никакого. Неявным образом включает в себя владельца текущей базы данных.

В отличие от MySQL, эти роли могут быть явно назначены другим ролям с использованием команды GRANT, например, команда, разрешающая роли app_user чтение всех данных:

```
GRANT pg_read_all_data TO app_user;
```

В дополнение к стандартной системе прав SQL, управляемой командой GRANT, в PostgreSQL на уровне таблиц можно определить политики защиты строк, ограничивающие для роли наборы строк, которые могут быть возвращены, добавлены, изменены или удалены командами SQL. Это называется также защитой на уровне

строк (RLS, Row-Level Security). По умолчанию таблицы не имеют политик, так что, если система прав SQL разрешает пользователю доступ к таблице, все строки в ней одинаково доступны для чтения или изменения.

Когда для таблицы включается защита строк, все обычные запросы к таблице на выборку или модификацию строк должны разрешаться политикой защиты строк. Команда включения защиты строк для таблицы:

```
ALTER TABLE таблица ENABLE ROW LEVEL SECURITY;
```

Если политика для таблицы не определена, применяется политика запрета по умолчанию, так что никакие строки в этой таблице нельзя увидеть или модифицировать. При создании политики можно также задать, для каких ролей она будет работать (по умолчанию – для всех) и для каких операторов (по умолчанию – также для всех).

Суперпользователи и роли с атрибутом BYPASSRLS всегда обращаются к таблице, минуя систему защиты строк.

Оператор CREATE POLICY позволяет определить политику защиты строк для указанной таблицы:

```
CREATE POLICY имя ON имя таблицы
[ FOR { ALL | SELECT | INSERT | UPDATE | DELETE }
]
[ TO { имя роли | PUBLIC| CURRENT_USER } [,…] ]
[ USING ( выражение USING ) ]
[ WITH CHECK ( выражение WITH CHECK) ]
```

Для строк определяются предикаты (выражения логического типа), существующие строки таблицы проверяются по выражению, указанному в USING, тогда как строки, которые могут быть созданы командами INSERT или UPDATE проверяются по выражению, ука-

занному в WITH CHECK. Если значение выражения истинно, то разрешается просмотр или модификация строки. Если роль не задана, то политика применяется ко всем ролям. Несколько политик на одну таблицу могут комбинироваться с помощью логического «или».

Так как роли могут владеть объектами баз данных и иметь права доступа к объектам других ролей, удаление роли не сводится к немедленному выполнению DROP ROLE. Сначала должны быть удалены или переданы другим владельцам все объекты, принадлежащие роли; также должны быть отозваны все права, данные роли.

Владение объектами можно передавать в индивидуальном порядке, применяя команду ALTER, например:

```
ALTER TABLE название OWNER TO роль;
```

Кроме того, для переназначения какой-либо другой роли владения сразу всеми объектами, принадлежащих удаляемой роли, можно применить команду REASSIGN OWNED. После того как все ценные объекты будут переданы новым владельцам, все оставшиеся объекты, принадлежащие удаляемой роли, могут быть удалены с помощью команды DROP OWNED.

DROP OWNED также удаляет все права, которые даны целевой роли для объектов, не принадлежащих ей.

Так как REASSIGN OWNED такие объекты не затрагивает, обычно необходимо запустить и REASSIGN OWNED, и DROP OWNED, чтобы полностью ликвидировать зависимости удаляемой роли. При этом порядок следования операторов важен.

При попытке выполнить DROP ROLE для роли, у которой сохраняются зависимые объекты, будут выданы сообщения, сообщающие, какие объекты нужно передать другому владельцу или удалить.

4.3 Привилегии MongoDB

MongoDB также поддерживает ролевую модель управления доступом, предоставляет доступ к данным и командам посредством авторизации на основе ролей, поддерживает **встроенные роли**, обеспечивающие различные уровни доступа, обычно необходимые в СУБД. Также можно дополнительно создавать определяемые пользователем роли.

Роль предоставляет привилегии для выполнения указанных действий над ресурсом. Каждая привилегия либо явно указывается в роли, либо наследуется от другой роли, либо и то, и другое.

Привилегия состоит из указанного ресурса и действий, разрешенных для ресурса.

Ресурс – это база данных, коллекция, набор коллекций или кластер. Если ресурсом является кластер, выполняемые действия влияют на состояние системы, а не на конкретную базу данных или коллекцию.

Роль может включать в свое определение одну или несколько существующих ролей, в этом случае роль наследует все привилегии включенных ролей.

Для просмотра привилегий для роли можно выполнить команду `rolesInfo` с указанием имени роли и базы данных, а также указанием параметров `showPrivilegies` и `showBuiltinRoles` для отображения привилегий и встроенных ролей:

```
db.runCommand(
   { rolesInfo : { role: <name>, db: <db> },
     showPrivileges: true,
     showBuiltinRoles: true
   })
```

MongoDB предоставляет встроенные **пользовательские** роли и роли администратора базы данных в каждой БД. Все остальные встроенные роли доступны только в базе данных `admin`. Встроенные роли пользователя БД включают в себя роли `read` и `readWrite`:

– роль `read` предоставляет возможность чтения данных для всех несистемных коллекций и коллекции `system.js`. Роль предоставляет доступ для чтения, разрешая выполнять следующие действия: выполнение запросов `find`, получение статистики `dbStats`, вычисление хеш-значений `dbHash`, получение списка коллекций `listCollections` и индексов `listIndexes` и другие операции;

– роль `readWrite` предоставляет все привилегии роли `read`, а также возможность изменять данные во всех несистемных коллекциях и коллекции `system.js`. Роль позволяет выполнять следующие действия над этими коллекциями: создание коллекции `createCollection`, удаление коллекций `dropCollection`, создание индексов `createIndex`, удаление индексов `dropIndex`, добавление `insert`, редактирование `update` и удаление записей `remove` и т.д.

Каждая БД включает следующие **роли администратора** БД:

– роль `dbAdmin` предоставляет возможность выполнять административные задачи: задачи, связанные с управлением схемой, индексированием и сбором статистики. Эта роль не предоставляет привилегий для управления пользователями и ролями;

– роль `dbOwner` сочетает в себе привилегии, предоставляемые ролями `readWrite`, `dbAdmin` и `userAdmin`. Владелец базы данных может выполнять любые административные действия с базой данных;

– роль `userAdmin` предоставляет возможность создавать и изменять роли и пользователей в текущей базе данных. Поскольку роль `userAdmin` позволяет пользователям предоставлять любые привилегии любому пользователю, включая себя, эта роль также косвенно предоставляет доступ суперпользователя либо к базе данных, либо, если она ограничена базой данных администратора, к кластеру. Более того, необходимо учитывать последствия предоставления роли `userAdmin` для безопасности: пользователь с этой ролью для базы данных может назначать себе любые привилегии в этой базе данных. Предоставление роли `userAdmin` в базе данных администратора имеет дополнительные последствия для безопасности, поскольку косвенно предоставляет доступ суперпользователя к кластеру.

Также база данных `admin` включает роли для **администрирования всей системы**, а не только одной базы данных:

– роль `clusterAdmin` обеспечивает максимальный доступ к управлению кластером. Эта роль объединяет в себе привилегии, предоставляемые ролями `clusterManager`, `clusterMonitor` и `hostManager`. Кроме того, роль позволяет удалять базу данных;

– роль `clusterManager` обеспечивает действия по управлению и мониторингу в кластере;

– роль `clusterMonitor` предоставляет доступ только для чтения к инструментам мониторинга;

– роль `hostManager` предоставляет возможность мониторинга и управления серверами;

– роли `backup` и `restore` предоставляют минимальные привилегии, необходимые для резервного копирования данных и восстановления данных из резервной копии.

На уровне таблицы `admin` также поддерживаются роли для чтения, записи данных и управления ролями пользователей во всех базах данных.

MongoDB предоставляет ряд встроенных ролей. Однако если эти роли не могут описать требуемый набор привилегий, можно создавать новые роли. MongoDB поддерживает следующий набор методов для управления ролями (таблица 4.4).

Таблица 4.4. Методы управления ролями в MongoDB

Оператор	**Описание**
`db.createRole()`	Создает роль и указывает ее привилегии.
`db.dropRole()`	Удаляет пользовательскую роль.
`db.dropAllRoles()`	Удаляет все пользовательские роли, связанные с базой данных.
`db.getRole()`	Возвращает информацию об указанной роли.
`db.getRoles()`	Возвращает информацию обо всех пользовательских ролях в базе данных.
`db.grantPrivilegesToRole()`	Назначает привилегии пользовательской роли.
`db.revokePrivilegesFromRole()`	Удаляет указанные привилегии из пользовательской роли.
`db.grantRolesToRole()`	Указывает роли, от которых пользовательская роль наследует привилегии.
`db.revokeRolesFromRole()`	Удаляет унаследованные роли из роли.
`db.updateRole()`	Обновляет пользовательскую роль.

Рассмотрим пример назначения ролей пользователям. Первым шагом произведем подключение к СУБД с учетной записью пользователя-администратора myUserAdmin. Далее выберем БД db_lectures и создадим пользователя lecturesReader с запросом пароля и без назначенных ролей:

```
use db_lectures
db.createUser( {
    user: "lecturesReader",
    pwd: passwordPrompt(), // или указать пароль
    roles: []
} )
```

Для назначения прав пользователю используется команда grantRolesToUser. В примере пользователю lecturesReader назначается роль read в БД db_lectures:

```
db.grantRolesToUser(
    "lecturesReader",
    [ { role: "read", db: "db_lectures" } ]
)
```

5 РЕЗЕРВНОЕ КОПИРОВАНИЕ

5.1 Основные понятия. Типы резервного копирования

Резервное копирование баз данных является важной частью любой комплексной стратегии обеспечение безопасности базы данных с точки зрения резервирования и восстановления данных. Резервное копирования БД имеет свои особенности и в определенных ключевых аспектах отличается от резервного копирования других типов файлов. Нельзя рассчитывать на то, что простое копирование файла базы всегда позволит восстановить ее при необходимости из копии, не принимая во внимание особенности, связанные с правами доступа к данным, состоянием базы данных и т.д.

В качестве основных причин необходимости резервного копирования можно выделить следующие:

– аварийное восстановление данных, необходимое из-за выхода из строя аппаратного обеспечения, физического уничтожения сервера, например, в случае пожара или стихийного бедствия, потери данных из-за несанкционированного доступа и т.д.;

– ошибки пользователей, например, удаление таблицы или данных. В частности, пользователь может забыть указать условия удаления строк в предложении `WHERE` оператора `DELETE`, что приведет к полному удалению данных из таблицы;

– аудит: иногда появляется необходимость получения данных или схемы базы данных в некоторый момент времени в прошлом;

– тестирование: периодическое обновление данных на тестовом сервере с использованием последних рабочих данных позволит проводить тестирование на реальных данных. При наличии резервной копии можно легко восстановить копию данных на тестовом сервере.

5.1.1 Классификация резервного копирования по резервируемым данным

Полное резервное копирование баз данных – это тип резервного копирования, который создает копию всей базы данных. Это самый очевидный и самый надежный тип резервного копирования базы данных. Однако он требует больше всего времени и ресурсов. Поэтому, как правило, полное резервное копирование баз данных выполняется реже всего.

При **дифференциальном (или разностном) копировании** выполняется резервное копирование только тех данных, которые изменились в базе данных с момента последнего **полного** резервного копирования. Так, например, если полное резервное копирование выполняется по понедельникам, а во все остальные дни недели – разностное, то во вторник в резервную копию будут помещены изменения, которые произошли с понедельника по вторник, в среду – с понедельника по среду и т.д. Этот способ гораздо быстрее, чем полное резервное копирование (конечно если запланировано регулярное полное резервное копирование и размер дифференциальных резервных копий не приближается к размеру полных резервных копий), но сопряжен с дополнительным риском того, что произойдет что-то непредвиденное, что приведет к невозможности восстановления базы данных.

Инкрементное резервное копирование аналогично дифференциальному, за исключением того, что в качестве момента времени, к которому относятся измененные данные, будет использоваться дата последней резервной копии, инкрементной или полной. Таким образом, при восстановлении из инкрементной резервной копии может потребоваться восстановить последнюю полную резервную копию, а также одну или несколько инкрементных резервных копий, чтобы добраться до текущего момента.

Для таких СУБД, которые поддерживают журналы транзакций, можно создавать инкрементные резервные копии этих журналов и использовать их для восстановления базы данных. В журнале транзакций фиксируются все транзакции и производимые ими в базе изменения, и его можно использовать для восстановления базы данных на определенный момент времени. При выполнении резервной копии журнала транзакций происходит его усечение, что является необходимым для всех БД с активированным журналом транзакций, в противном случае, за счет журнала транзакций размер базы данных будет расти и довольно быстро, вплоть до завершения места на диске. В журнале транзакций, если он активирован, фиксируются все транзакции, с момента его последнего усечения.

Так, например, если полное резервное копирование выполняется по понедельникам, а во все остальные дни недели – резервное копирование журнала транзакций, то во вторник в резервную копию будут записаны те изменения, которые произошли с понедельника по вторник, в среду будут записаны изменения со вторника по среду и т.д. Такое резервное копирование будет выполняться быстрее, чем дифференциальное, но придется пожертвовать скоростью восстановления.

Если сбой произойдет, например, в четверг, то при использовании дифференциального резервного копирования потребуются две резервные копии: за понедельник (полная) и за четверг (разностная). В той же ситуации при использовании инкрементного резервного копирования журналов транзакций потребуются полная копия за понедельник и копии журналов транзакций за вторник, среду и четверг.

5.1.2 Классификация резервного копирования по способу создания

По способу создания резервные копии данных можно разделить на две большие группы: физические и логические.

Физическая резервная копия – это копия исходных двоичных данных, часто создаваемая на уровне операционной системы. Любой метод резервного копирования, включающий копирование данных без использования построчного доступа к БД, считается методом физического резервного копирования. Такой тип резервного копирования подходит для любых баз данных, но рекомендуем для критически важных баз, которые необходимо быстро восстанавливать после сбоев.

Логическая резервная копия – это набор команд SQL, содержащий логическую структуру базы данных (запросы `DDL`) и ее содержимое (запросы `INSERT`). Резервное копирование выполняется посредством сканирования таблицы с прохождением каждой строки данных. Такой тип резервного копирования подходит для небольших баз данных, не требовательных к скорости восстановления.

5.1.2.1 Достоинства и недостатки физического и логического копирования данных

5.1.2.1.1 Физическое резервное копирование

Время создания физических резервных копий ниже, чем логических, т.к. в процессе создания резервной копии происходит только запись данных содержащихся в файлах БД, в то время как при создании логической копии происходит преобразование физических данных к их логической структуре и запись соответствующих команд SQL в файл.

Время восстановления физических резервных копий также ниже, чем логических, т.к. фактически происходит перезапись файлов базы данных данными из копии, в то время как при восстановлении логической копии происходит последовательное исполнение SQL-скрипта для создания объектов БД и вставки данных.

В то же время, для выполнения физической резервной копии сервер как правило должен быть остановлен. Существуют подход к резервному копированию файловой системы, который заключается в создании «целостного снимка» каталога с данными, если это поддерживает файловая система. Типичная процедура включает создание «замороженного снимка» тома, содержащего базу данных, затем копирование всего каталога с этого снимка на устройство резервного копирования, и наконец освобождение замороженного снимка. При этом сервер базы данных может не прекращать свою работу. Однако резервная копия, созданная таким способом, содержит файлы базы данных в таком состоянии, как если бы сервер баз данных не был остановлен штатным образом.

Объем физических и логических копий сильно зависит от структуры базы данных. С одной стороны, логическая копия представляет собой SQL скрипт в некомпактном текстовом виде, где каждой записи в каждой таблице соответствует оператор `INSERT`, что, конечно, больше, чем размер физических файлов. С другой стороны, логические копии могут не содержать индексов, т.к. достаточно команд их пересоздания.

Физические резервные копии обладают низкой степенью переносимости, как правило базы данных из этих копий могут быть восстановлены только на сервере с аналогичной архитектурой, под управлением той же ОС и в той же версии СУБД, в то время как логические копии позволяют восстановить базу данных на сервере с иной архитектурой, под управлением другой ОС и в другой версии СУБД, при наличии совместимости на уровне команд. Поскольку СУБД, как правило, не имеют обратной совместимости по форматам резервных копий, иногда использование логической резервной копии остается единственным вариантом переноса базы данных на сервер с более ранней версией СУБД.

5.1.2.1.2 Логическое резервное копирование

Логическое резервное копирование не требует остановки сервера. Логические копии могут быть относительно легко перенесены на компьютер/БД с другой архитектурой системы.

Основные недостатки использования логического резервного копирования заключаются во времени работы. Время создания логической резервной копии существенно выше, чем физической. Время восстановления из логической резервной копии также выше, так как требуется загружать и интерпретировать команды добавления данных и перестраивать индексы.

Важно, что и физические резервные копии, и логические могут восстанавливаться с ошибками. Однако физическая копия либо восстановится, либо нет, а логическая копия может восстановиться не вполне корректно. Ошибки могут возникать в части скрипта, описывающей вставку данных, и обычно причиной тому служат различные проблемы в строковых данных (например, проблемы с кодировкой). Соответственно, строки данных, при вставке которых произошла ошибка, в восстанавливаемую базу данных не попадут, что в свою очередь может привести к нарушению ссылочной целостности. Поэтому при восстановлении из логических копий желательно вести лог ошибок и анализировать его.

Логическое резервное копирование обычно больше подходит для решения задач клонирования и переноса баз данных, для решения задачи восстановления баз данных после сбоев желательно использовать физические резервные копии, если используемая СУБД предоставляет такие возможности.

5.1.3 Классификация резервного копирования по доступности сервера

Автономным резервным копированием называют процесс создания резервной копии на остановленном сервере, т.е. на момент

резервного копирования работа с БД не ведется, файлы БД не изменяются, и все данные находятся в согласованном состоянии. Благодаря этому можно быстро скопировать файлы, не беспокоясь о сохранении состояния на текущий момент, пока другие процессы читают и записывают данные. Такие виды резервного копирования проще в реализации, но требуют остановки сервера, что часто невозможно.

Оперативным (горячим) резервным копированием называют процесс создания резервной копии, не требующий остановки сервера, т.е. на момент резервного копирования пользователи продолжают работу с БД. Кроме того, при оперативном резервном копировании возрастает нагрузка на сервер. Такой вид резервного копирования сложнее, т.к. после копирования файлов данных требуется еще и анализ журнала транзакций, но является единственным доступным видом резервирования для критических систем.

5.2 Факторы планирования резервного копирования

5.2.1 Показатели RTO и RPO

Расписание для каждого типа резервного копирования будет зависеть от показателей **RTO** и **RPO** информационной системы. С этими показателями желательно определиться до этапа планирования стратегии резервного копирования и восстановления баз данных. Без понимания этих показателей есть вероятность неправильно оценить риски, сопряженные с отсутствием доступа к данным, и спланировать неподходящую стратегию резервирования и восстановления.

RPO (Recovery Point Objective) – это максимальный период времени, за который могут быть потеряны данные в результате инцидента. Допустим, имеется информационная система и показатель RPO для нее определен в один час. Это значит, что при аварии мы

готовы к тому, что после восстановления системы допускается потеря данных не более, чем за один последний час. В определенных ситуациях количество потерянных данных может быть и меньше, но ни при каких условиях не более одного часа. Этот показатель говорит о том, как часто необходимо выполнять резервирование данных, и какие технологии применять, чтобы выполнять этот показатель. Теоретически этот показатель может быть равен 0, но на практике организовать это достаточно сложно. Это можно организовать, только, если запись идет как минимум в два разных хранилища.

RTO (Recovery Time Objective) – это промежуток времени, в течение которого система может оставаться недоступной в случае аварии. Допустим, в центре обработки данных произошел пожар, но необходимо, чтобы система была снова доступна для работы через два часа – это и есть показатель RTO. Необходимо планировать резервирование и восстановление данных так, чтобы за указанный промежуток восстановить работоспособность информационной системы на резервном оборудовании или площадке. Это можно реализовать с помощью различных технологий, однако в любом случае необходимо обеспечить этот показатель при инциденте.

5.2.2 Возможность тестирования резервных копий

Также на планирование резервного копирования данных оказывает влияние то, какие имеются возможности тестирования резервных копий, чтобы проверить, что данные действительно могут быть восстановлены. При минимальной возможности тестирования резервных копий в первую очередь следует рассматривать стратегию полного резервного копирования БД по причине его максимальной надежности. Однако, если имеется возможность детального тестирования дифференциальных резервных копий или резервных копий журнала транзакций, можно запланировать преимущественное выполнение таких типов резервирования, а полное

резервное копирование выполнять нечасто, например, раз в неделю или в месяц.

5.2.3 Характеристики базы данных

Объем хранимых данных и режим их использования также влияют на планирование стратегии резервного копирования. Если данных меняются достаточно редко, то можно обойтись только полным резервным копированием, которое выполняется достаточно часто. Очевидно, что чем больше объем данных, тем медленнее будет происходить их полное резервное копирование, и, если режим доступа к данным подразумевает внепиковый период (например, ночью), рекомендуется запланировать полное резервное копирование базы данных именно на это время. При высокой интенсивности изменения данных желательно подключать периодическое разностное копирование или копирование журнала транзакций.

5.2.4 Ограничения на ресурсы

Ограничения на ресурсы, такие как оборудование, персонал, объем хранилища резервных копий и его физическая безопасность также влияют на стратегии резервного копирования. Если с аппаратными ограничениями все достаточно прозрачно, то ограничения на персонал, например, могут выражаться в том, что нет возможности держать специалиста, ответственного за тестирование резервных копий, и это в некотором смысле определяет возможную стратегию резервирования данных.

5.3 Стратегии резервного копирования

Стратегия резервного копирования определяется рассмотренными выше факторами, и для высоконагруженных многопользовательских систем с высокой стоимостью времени простоя такие стратегии индивидуальны, вплоть до аппаратной части.

Единственной стратегии резервного копирования, которая подходила бы всем пользователям, не существует. Стратегия, хорошо работающая в системе с одним пользователем, может оказаться непригодной для системы, обслуживающей сто пользователей. Аналогично, стратегия, разработанная для системы, в которой каждый день меняется множество файлов, окажется неэффективной в системе, в которой данные меняются редко.

Однако существуют некоторые базовые стратегии, применимые в большинстве случаев для массовых информационных систем:

– «простая» стратегия. Это стратегия ежедневного полного резервного копирования, обычно ночью. При этом обычно хранятся все резервные копии за последнюю неделю или две, а также резервные копии на начало месяца (чтобы можно было при необходимости восстановить базу данных и использовать ее для формирования отчетности). Такая стратегия максимально проста в реализации, не требует тестовых серверов для проверки восстановления и имеет хороший показатель RTO. Что касается показателя RPO, то нетрудно понять, что он может достигать восьми часов (при учете использование БД только в рабочее время). Фактически, в худшем случае допускается потеря данных за весь рабочий день. Несмотря на простоту и надежность, такая стратегия может быть рекомендована только для БД с низкой интенсивностью изменения данных;

– «сбалансированная» стратегия. Этот вариант стратегии использует все три вида резервного копирования. Стратегия заключается в том, что полное резервное копирование выполняется раз в неделю (например, в воскресенье), разностное резервное копирование выполняется каждую ночь, и несколько раз в день выполняется резервное копирование журналов транзакций. В отличии от «простой» стратегии, реализация этой

стратегии требует тестирования резервных копий, т.к. использует потенциально ненадёжные типы резервного копирования. Также «сбалансированная» стратегия несколько хуже по показателю RTO, но может быть значительно лучше по показателю RPO, который меньше и зависит от частоты резервных копий журнала транзакций в течение рабочего дня. Такая стратегия может быть рекомендована как для больших баз данных (десятки и сотни гигабайт), так и для тех баз данных, которые активно изменяются.

Но, конечно, в зависимости от факторов, влияющих на планирования резервирования данных, в конкретных случаях могут применяться и другие стратегии, реализующие некие промежуточные варианты.

5.4 Общие рекомендации по резервированию данных

Резервные копии не следует хранить на том же физическом хранилище, что и файлы исходной базы данных. В таком случае, если возникнет какая-либо проблема с диском, будут потеряны и база данных, и ее резервная копия. Следует помнить, что физические устройства, на которых хранятся резервные копии, также подвержены износу. Поэтому рекомендуется хранить несколько резервных копий на разных физических дисках или в хранилищах с зеркалированием данных. При организации хранения резервных копий также стоит помнить о том, что резервная копия – это абсолютно та же база данных, для которой необходимо учитывать вопросы конфиденциальности данных и уделить надлежащее внимание безопасности хранилища.

Автоматизацию процессов резервного копирования необходимо настроить по заданному расписанию. В этом случае исключа-

ется человеческий фактор, и можно быть уверенным, что все резервные копии, необходимые для восстановления в случае инцидента, сделаны и актуальны. Также необходимо проверять, что запланированные задачи по созданию резервных копий успешно выполняются.

Рекомендуется использовать тестовый сервер для проверки процедуры восстановления резервных копий. Даже если процесс резервного копирования завершился успешно, есть шанс, что восстановление данных из этой резервной копии невозможно. Единственный способ убедиться в корректности резервной копии – это выполнить процесс восстановления в реальном сценарии, на тестовом сервере. Это минимизирует вероятность ошибки восстановления в случае инцидента. Кроме того, проведение восстановления на тестовом сервере позволяет оценить показатель RTO и, соответственно, адекватность стратегии резервирования данных в целом.

Следующая рекомендация – резервирование системных БД. Как правило, в любой СУБД критические данные хранятся в системных базах данных (в MySQL это БД `mysql`, в PostgreSQL – БД `postgres`). В системных базах данных хранятся параметры конфигурации сервера, параметры конфигурации отдельных баз данных, права доступа к объектам и т.д. В связи с этим рекомендуется резервировать не только пользовательские БД, но и служебные, так как при возникновении инцидента, если повреждены системные базы данных, невозможно будет восстановить доступ к данным в разумное время, не имея их резервных копий.

Не всегда риск прекращения нормального функционирования базы данных и клиентского программного обеспечения сопряжен с какими-либо инцидентами. Действия по изменению структуры базы данных, массовое изменение данных, операции массового импорта данных из сторонних источников, ввод новой функциональ-

ности в клиентское ПО и т.д. – всё это может привести к несогласованности данных и, соответственно, к невозможности штатного функционирования БД и клиентских приложений. В связи с чем рекомендуется выполнять внеплановое полное резервное копирование данных в указанных случаях.

В добавок, рекомендуется использовать все предоставляемые СУБД доступные параметры верификации при выполнении резервного копирования. Это позволит быть уверенным в том, что все создаваемые резервные копии будут созданы корректно и последовательны с точки зрения транзакций.

5.5 Резервное копирование в MySQL

Развитая система физического резервного копирования MySQL Enterprise Backup в настоящий момент предлагается только в составе коммерческой версии MySQL Enterprise Edition. Эта система поддерживает высокопроизводительное резервирование данных в оперативном или горячем режимах, инкрементное резервное копирование, частичное и выборочное резервное копирование, шифрование данных алгоритмом AES-256, компрессию резервных копий и другие функции, свойственные подсистемам резервного копирования современных СУБД. Физическое резервное копирование выполняется значительно быстрее, чем логическое копирование с использование команды `mysqldump`.

Создание логических резервных копий доступно с использованием инструмента `mysqldump`, который позволяет выполнить оперативное логическое резервное копирование, не блокируя таблицы.

Свободно распространяемая версия СУБД MySQL предлагает только один способ полного физического резервного копирования:

копирование файлов базы данных средствами ОС. Файловая организация базы данных MySQL (для движка InnoDB) подразумевает следующий набор файлов:

- файлы данных (ibdata* и. ibd);
- лог-файлы (ib_logfile*);
- файл конфигурации (my.cnf).

Для копирования указанных файлов требуется остановить сервер. Такой способ не всегда приемлем, так как резервирование производится в автономном режиме, однако альтернативой является логическая резервная копия, восстановление из которой происходит чрезвычайно медленно.

Другим способом получения копии данных является формирование текстовых файлов с данными. Для создания текстового файла, содержащего данные таблицы, можно использовать команду:

```
SELECT * INTO OUTFILE 'file_name' FROM table_name;
```

Файл создается на узле сервера MySQL, а не на узле клиента. Для создания структуры таблиц необходимо дополнительно создавать файлы, которые содержат команды `CREATE TABLE`.

Следующим способом создания текстовых файлов с данными является использование утилиты `mysqldump` с параметром `--tab`.

В СУБД MySQL поддерживается инкрементное резервное копирование, которое основано на копировании бинарных лог-файлов. Такой лог файл является журналом транзакций, который содержит «события», описывающие изменения базы данных, такие как операции создания таблицы или изменения данных таблицы. Команды из журнала транзакции могут использоваться для изменения состояния базы данных.

Также одним из способов является создание резервных копий с использованием реплик. Если возникают проблемы с производительностью исходного сервера при создании резервных копий, одна из стратегий, которая может помочь – настроить репликацию. Репликация – это механизм синхронизации содержимого нескольких копий объекта. После получения точной копии базы данных на другом сервере можно выполнять резервное копирование на реплике, а не на источнике.

5.5.1 Логическое резервное копирование в MySQL

Для создания логической резервной копии базы данных в комплекте с MySQL Server поставляется утилита `mysqldump`, которая формирует последовательность операторов SQL, при выполнении которых восстанавливаются исходные определения объектов базы данных и данных таблиц.

Утилита поддерживает два типа выходных данных:

– без параметра `--tab` операторы SQL записываются в стандартный выходной поток. Выходные данные состоят из операторов `CREATE` для создания объектов (баз данных, таблиц, хранимых процедур и т.д.) и операторов `INSERT` для загрузки данных в таблицы. Вывод можно сохранить в файле и загрузить с помощью `mysql` для воссоздания выгруженных объектов;

– с параметром `--tab mysqldump` создает два выходных файла для каждой выгруженной таблицы. Сервер записывает один файл в виде текста с разделителями табуляции, по одной строке на строку таблицы. Этот файл называется `tbl_name.txt` в выходном каталоге. Оператор `CREATE TABLE` записывается в файл с именем `tbl_name.sql` в выходном каталоге.

По умолчанию `mysqldump` записывает результат в стандартный выходной поток. Результат можно сохранить в файл следующим образом:

```
$> mysqldump [arguments] > file_name
```

Утилита может быть использована для создания резервной копии всех БД сервера или заданных БД:

```
$> mysqldump --all-databases > dump.sql
$> mysqldump --databases db1 db2 db3 > dump.sql
```

В случае создания резервной копии одной БД можно опустить опцию
`--databases`:

```
$> mysqldump test > dump.sql
```

В этом случае дамп не содержит инструкций `CREATE DATABASE` или `USE`:

– при загрузке копии необходимо указать имя базы данных по умолчанию, чтобы сервер знал, какую базу данных необходимо восстанавливать;

– при загрузке копии можно указать имя базы данных, отличное от исходного имени, что позволит повторно загрузить данные в другую БД;

– БД для загрузки данных должна быть предварительно создана.

Утилита `mysqldump` поддерживает более 100 параметров, основные из них представлены в таблице 5.1.

Таблица 5.1. Основные параметры утилиты mysqldump

Параметры	**Описание**
`--host`	Адрес MySQL сервера
`--port`	Порт MySQL сервера
`--user`	Имя пользователя MySQL сервера
`--password`	Пароль пользователя MySQL сервера
`--routines`	Включать в резервную копию хранимые процедуры и функции

Окончание табл. 5.1

Параметры	Описание
`--single-transaction`	Создать резервную копии в рамках одной транзакции
`--source-data`	Включить в резервную копию позицию и имя бинарного лог-файла сервера на момент создания копии
`--flush-logs`	Создать новый бинарный лог-файл перед созданием резервной копии

Параметры `--source-data` и `--flush-logs` необходимы для настройки инкрементного резервного копирования с использованием бинарных логов.

Для восстановления БД из резервной копии необходимо использовать файл с дампом в качестве входных данных для клиента `mysql`. Если файл дампа был создан с опцией `--all-databases` или `--databases`, то он содержит операторы `CREATE DATABASE` и `USE`, и нет необходимости указывать базу данных по умолчанию, в которую следует загружать данные:

```
$> mysql --user root --password < dump.sql
```

Из `mysql` то же действие может быть выполнено с использованием команды `source`:

```
mysql> source dump.sql
```

Если файл представляет собой дамп одной базы данных, не содержащий операторов `CREATE DATABASE` и `USE`, сначала необходимо создать базу данных, затем указать ее при импорте данных:

```
$> mysql db1 < dump.sql
```

Другой способ – подключиться к mysql, создать БД командой `CREATE DATABASE`, выбрать ее как используемую и выполнить импорт командой `source`.

5.5.2 Инкрементное резервное копирование в MySQL

В MySQL поддерживается инкрементное резервное копирование, которое основано на копировании двоичных журналов. Двоичный журнал содержит события, которые описывают изменения в БД (например, создание таблиц, изменения в данных таблицы и т.д.) и представляет собой упорядоченную последовательность одноименных файлов (обычно это `server_name-bin`) с возрастающим числовым расширением (SERV-bin.000001, SERV-bin.000002, SERV-bin.000003 и т.д.). Запись событий ведется в текущий журнал – последний файл в последовательности. Новый файл двоичного журнала создаются сервером автоматически при каждом перезапуске сервера, по достижении текущим журналом предельного размера или при выполнении сброса журнала с помощью команды `FLUSH LOGS`.

Создание инкрементной резервной копии состоит из следующих шагов. Каждый раз, когда необходимо сделать инкрементную резервную копию (содержащую все изменения, произошедшие со времени последнего полного или инкрементного резервного копирования), необходимо сбросить двоичный журнал с помощью команды `FLUSH LOGS` и скопировать все двоичные журналы, начиная с момента последнего полного или инкрементного резервного копирования в хранилище резервных копий. Этот набор двоичных журналов и является инкрементной резервной копией. Соответственно, каждый раз при создании полной резервной копии также необходимо сбрасывать журнал, используя команду `FLUSH LOGS` или утилиту `mysqldump` с параметром `--flush-logs`.

Для восстановления из инкрементной резервной копии сначала необходимо восстановить полную резервную копию, а затем последовательно восстановить изменения из файлов журнала.

Для восстановления инкрементной резервной копии используется утилита `mysqlbinlog`, которая используется для обработки журналов:

```
$> mysqlbinlog [arguments] log_file
```

Для просмотра журнала в консоли или записи содержимого в текстовый файл можно выполнить следующие команды:

```
$> mysqlbinlog AGAFONOV-bin.000024
$> mysqlbinlog AGAFONOV-bin.000024 > incremental.log
```

Сохраненный файл можно использовать для восстановления данных:

```
$> mysql --user root --password < incremental.log
```

Можно выполнить восстановление без использования промежуточного файла. Если необходимо восстановить из нескольких файлов журнала, лучше это делать одной командой:

```
$> mysqlbinlog AGAFONOV-bin.000024 AGAFONOV-bin.000025 | mysql -u root -p
```

Можно также выполнить восстановление на конкретный момент времени:

```
$> mysqlbinlog --stop-datetime="2022-11-01 9:59:59" AGAFONOV-bin.000024 | mysql -u root –p
```

5.6 Резервное копирование в PostgreSQL

PostgreSQL поддерживает три разных подхода к резервному копированию данных:

- логическое копирование – выгрузка в SQL;
- физическое копирование на уровне файлов;
- непрерывное архивирование, являющееся по сути инкрементным резервным копированием.

5.6.1 Логическое резервное копирование в PostgreSQL

Логическое резервное копирование в СУБД PostgreSQL выполняется с помощью утилиты `pg_dump`. Эта программа генерирует файл с командами SQL, которые при выполнении на сервере пересоздадут объекты базы данных в том же самом состоянии, в котором они были на момент выгрузки. Результат работы программы записывается в стандартный поток вывода. Резервное копирование выполняется в оперативном режиме.

Синтаксис команды имеет следующий вид:

`pg_dump [`***параметр подключения...***`] [`***параметр...***`] [`***имя_бд***`]`

Список основных параметров представлен в таблице 5.2.

Таблица 5.2. Параметры pg_dump

Параметр	**Описание**
`--host, -h`	Адрес сервера
`--port, -p`	Порт сервера
`--username, -U`	Имя пользователя, под которым производится подключение
`--password, -W`	Принудительно запрашивать пароль перед подключением
`--format, -F`	Формат вывода копии (plain, custom, directory, tar)

Утилита поддерживает большое число параметров, среди которых можно особо отметить формат вывода резервной копии. По умолчанию используется формат plain – формируется текстовый SQL-скрипт. Формат custom позволяет выгрузить данные в специальном архивном формате, пригодном для дальнейшего использования утилитой `pg_restore`. Наряду с форматом directory является наиболее гибким форматом, позволяющим вручную выбирать и сортировать восстанавливаемые объекты. Вывод в этом формате

по умолчанию сжимается. Directory выгружает данные в виде каталога, tar выгружает данные в формате tar, без сжатия.

Простейший вариант вызова утилиты выглядит следующим образом:

```
$> pg_dump имя_базы > файл_дампа
```

Пример создания резервной копии БД `pagila` с подключение от пользователя `postgres`:

```
$> pg_dump -U postgres pagila > pagila.sql
```

Текстовые файлы, созданные `pg_dump`, используются для последующего чтения программой `psql`. Общий вид команды для восстановления дампа:

```
$> psql имя_базы < файл_дампа,
```

где `файл_дампа` — это файл, содержащий вывод команды `pg_dump`. База данных, заданная параметром `имя_базы`, не будет создана данной командой, поэтому ее необходимо создать самостоятельно перед запуском `psql`. Программа `psql` принимает параметры, указывающие сервер, к которому осуществляется подключение, и имя пользователя, подобно `pg_dump`.

Дампы, выгруженные не в текстовом формате, восстанавливаются утилитой `pg_restore`:

```
$> pg_restore -d имя_базы файл_дампа
```

По умолчанию, если происходит ошибка SQL, программа `psql` продолжает выполнение. Если же запустить `psql` с установленной переменной `ON_ERROR_STOP`, это поведение поменяется, и `psql` завершится с кодом 3 в случае возникновения ошибки SQL:

```
$> psql --set ON_ERROR_STOP=on имя_базы < файл_дампа
```

В любом случае возможно только частичное восстановление базы данных. В качестве альтернативы можно указать, что весь дамп должен быть восстановлен в одной транзакции, так что восстановление либо полностью выполнится, либо полностью отменится. Включить данный режим можно, передав `psql` аргумент `-1` или `--single-transaction`. Выбирая этот режим, стоит учитывать, что даже незначительная ошибка может привести к откату восстановления, которое могло продолжаться несколько часов. Однако это всё же может быть предпочтительней, чем вручную вычищать сложную базу данных после частично восстановленного дампа.

Благодаря способности `pg_dump` и `psql` писать и читать каналы ввода/вывода, можно скопировать базу данных непосредственно с одного сервера на другой:

```
pg_dump -h host1 имя_базы | psql -h host2 имя_базы
```

Программа `pg_dump` выгружает только одну базу данных в один момент времени и не включает в дамп информацию о ролях и табличных пространствах (так как это информация уровня кластера, а не самой базы данных). Для удобства создания дампа всего содержимого кластера баз данных предоставляется программа `pg_dumpall`, которая делает резервную копию всех баз данных кластера, а также сохраняет данные уровня кластера. Базовое использование этой команды имеет вид:

```
pg_dumpall > файл_дампа
```

Полученную копию можно восстановить с помощью `psql`:

```
psql -f файл_дампа postgres
```

Для резервного копирования больших баз данных можно использовать сжатые дампы, разбивать выводимые данные на небольшие файлы средствами Unix или использовать специальный формат дампа `pg_dump`.

5.6.2 Резервное копирование на уровне файлов в PostgreSQL

Альтернативной стратегией резервного копирования является непосредственное копирование файлов, в которых PostgreSQL хранит содержимое базы данных

Однако существуют два ограничения, которые делают этот метод непрактичным или как минимум менее предпочтительным по сравнению с `pg_dump`:

– чтобы полученная резервная копия была работоспособной, сервер баз данных должен быть остановлен. Запрещение всех подключений к серверу не даст необходимого результата из-за того, что tar и подобные средства не получают мгновенный снимок состояния файловой системы, кроме того, в сервере есть внутренние буферы;

– копирование и восстановление отдельных таблиц или баз данных в соответствующих файлах или каталогах также невозможно, потому что информацию, содержащуюся в этих файлах, нельзя использовать без файлов журналов транзакций, которые содержат состояние всех транзакций. Без этих данных файлы таблиц непригодны к использованию. Таким образом, копирование на уровне файловой системы будет работать, только если выполняется полное копирование и восстановление всего кластера баз данных.

Ещё один подход к резервному копированию файловой системы заключается в создании «целостного снимка» каталога с данными, если это поддерживает файловая система. Типичная процедура включает создание «замороженного снимка» тома, содержащего базу данных, затем копирование всего каталога с данными (а не его отдельных частей) из этого снимка на устройство резервного копирования, и наконец освобождение замороженного снимка.

5.6.3 Непрерывное архивирование в PostgreSQL

Другое возможное решение для создания физической резервной копии – использование механизма непрерывного архивирования и восстановление на момент времени (Point-in-Time Recovery, PITR). Такие резервные копии не могут пострадать от изменений файловой системы в процессе резервного копирования. Для этого требуется включить непрерывное архивирование только на время резервного копирования, для восстановления применяется процедура восстановления из непрерывного архива.

В процессе работы PostgreSQL ведётся журнал предзаписи (Write-Ahead Log, WAL), который расположен в подкаталоге `pg_wal/` каталога с данными кластера баз данных. В этот журнал записываются все изменения, вносимые в файлы данных. Прежде всего, журнал необходим для безопасного восстановления после аварийной остановки сервера. В этом случае целостность СУБД может быть восстановлена в результате «воспроизведения» записей, зафиксированных после последней контрольной точки.

Однако наличие журнала делает возможным использование инкрементной стратегии копирования баз данных. Можно сочетать резервное копирование на уровне файловой системы с копированием файлов WAL.

Если потребуется восстановить данные, можно восстановить копию файлов, а затем воспроизвести журнал из скопированных файлов WAL, и, таким образом, привести систему в нужное состояние.

По сути, этот подход аналогичен использованию бинарных журналов в MySQL.

5.6.3.1 Непрерывное архивирование WAL-файлов

Описанный подход с использованием непрерывного архивирования WAL-файлов более сложен для администрирования, чем любой из описанных раннее способов резервного копирования, но он имеет значительные преимущества:

– в качестве начальной точки для восстановления необязательно иметь полностью согласованную копию на уровне файлов. Внутренняя несогласованность копии будет исправлена при воспроизведении журнала;

– поскольку при воспроизведении можно обрабатывать неограниченную последовательность файлов WAL, непрерывную резервную копию можно получить, просто продолжая архивировать файлы WAL. Это особенно ценно для больших баз данных, полные резервные копии которых делать как минимум неудобно;

– воспроизводить все записи WAL до самого конца нет необходимости. Воспроизведение можно остановить в любой точке и получить целостный снимок базы данных на выбранный момент времени. Таким образом, данная технология поддерживает восстановление на момент времени: можно восстановить состояние базы данных на любое время с момента создания резервной копии;

– если непрерывно передавать последовательность файлов WAL другому серверу, получившему данные из базовой копии того же кластера, получается система «тёплого» резерва. В любой момент можно запустить второй сервер, и он будет иметь практически текущую копию баз данных.

Также следует отметить, что программы `pg_dump` и `pg_dumpall` не создают копии на уровне файловой системы и не могут применяться как часть решения по непрерывной архивации. Создаваемые ими копии являются логическими и не содержат информации, необходимой для воспроизведения WAL.

Как и обычное резервное копирование файловой системы, этот метод позволяет восстанавливать только весь кластер баз данных целиком, но не его части. Кроме того, для архивов требуется большое хранилище: базовая резервная копия может быть объёмной,

а нагруженные системы будут генерировать многие мегабайты трафика WAL, который необходимо архивировать. Тем не менее этот метод резервного копирования является предпочтительным во многих ситуациях, где необходима высокая надёжность.

Для успешного восстановления с применением непрерывного архивирования необходима непрерывная последовательность заархивированных файлов WAL, начинающаяся не позже, чем с момента начала копирования. Так что для начала необходимо настроить процедуру архивирования файлов WAL до того, как необходимо получить первую базовую копию.

В абстрактном смысле, запущенная СУБД PostgreSQL производит неограниченно длинную последовательность записей WAL. СУБД физически делит эту последовательность на файлы сегментов WAL, которые обычно имеют размер 16 МБ (хотя размер сегмента может быть изменён при инициализации). Файлы сегментов получают цифровые имена, которые отражают их позицию в абстрактной последовательности WAL. Когда архивирование WAL не применяется, система обычно создаёт только несколько файлов сегментов и затем перезаписывает их. Предполагается, что файлы сегментов, содержимое которых предшествует последней контрольной точке, уже не представляют интереса и могут быть переработаны.

При архивировании данных WAL необходимо считывать содержимое каждого файла-сегмента, как только он заполняется, и сохранять эти данные, прежде чем файл-сегмент будет переработан и использован повторно.

В зависимости от применения и доступного аппаратного обеспечения, возможны разные способы сохранения данных: можно скопировать файлы-сегменты в смонтированный каталог на другом сервере, записать их на диск и т.д. PostgreSQL позволяет админи-

стратору указать команду оболочки или библиотеку архивирования, которая будет запускаться для копирования файла завершённого сегмента в нужное место. Её действие может заключаться как в выполнении простых команд оболочки, включая `copy`, так и в вызове функции на языке C.

Чтобы включить архивирование WAL, необходимо установить в параметре конфигурации `wal_level` уровень `replica` или выше, в `archive_mode` – значение `on` и задать команду архивирования в параметре `archive_command`. На практике эти параметры всегда задаются в файле `postgresql.conf`.

В `archive_command` символы `%p` заменяются полным путём к файлу, подлежащему архивации, а `%f` заменяются только именем файла. Простейшая команда для архивации имеет следующий вид:

```
archive_command       =       'test       !       -f
/mnt/server/archivedir/%f
          && cp %p /mnt/server/archivedir/%f' # Unix
archive_command = 'copy "%p"
          "C:\\server\\archivedir\\%f"' # Windows
```

Для Windows команда будет копировать архивируемые сегменты WAL в каталог `C:\server\\archivedir\`.

Команда или функция архивирования вызывается, только когда сегмент WAL заполнен до конца. Таким образом, если сервер постоянно генерирует небольшой трафик WAL (или есть продолжительные периоды, когда это происходит), между завершением транзакций и их безопасным сохранением в архиве может образоваться большая задержка. Чтобы ограничить время жизни неархивированных данных, можно установить `archive_timeout`, чтобы сервер переключался на новый файл сегмента WAL как минимум с заданной частотой.

5.6.3.2 Непрерывное архивирование. Базовая копия

Проще всего получить базовую резервную копию, используя программу pg_basebackup. Эта программа выполняет оперативное копирование и сохраняет базовую копию в виде обычных файлов или в архиве tar. Для запуска утилиты указываются параметры подключения к БД и параметры создания резервной копии.

pg_basebackup [***параметр-подключения...***] [***параметр...***]

Простейший вариант вызова выглядит следующим образом:

```
$> pg_basebackup -D каталог
$> pg_basebackup -U postgres -D "E:\\backup"
```

После выполнения этой команды в каталоге E\backup будет создана физическая резервная копия.

PostgreSQL не поддерживает встроенные средства для шифрования резервных копий, поэтому при необходимости следует защищать архивные данные внешними средствами шифрования.

Процесс создания базовой резервной копии записывает файл истории резервного копирования, который сохраняется в каталоге архивации WAL файлов. Данный файл получает имя по имени файла первого сегмента WAL, который потребуется для восстановления скопированных файлов. Файл истории резервного копирования – это текстовый файл, в который записывается метка, которая была передана pg_basebackup, а также время и текущие сегменты WAL в момент начала и завершения создания резервной копии.

5.6.3.3 Непрерывное архивирование. Восстановление данных

Для восстановления данных из полной резервной копии (полученной с помощью утилиты pg_basebackup) и инкрементных копий (файлы журналов WAL) следует выполнить следующее:

1) остановить сервер баз данных, если он запущен;
2) скопировать весь текущий каталог кластера баз данных и все табличные пространства во временный каталог на случай, если они вам понадобятся. Как минимум, сохранить содержимое подкаталога `pg_wal` каталога кластера, так как он может содержать журналы, не попавшие в архив перед остановкой системы;
3) удалить все содержимое директории сервера, на котором будет выполняться восстановление данных;
4) скопировать содержимое папки, в которую была выполнена полная резервная копия, в директорию данных сервера;
5) удалить все файлы из `pg_wal/`; они восстановились из резервной копии файлов и поэтому, скорее всего, будут старее текущих;
6) если на шаге 2 были сохранены незаархивированные файлы с сегментами WAL, необходимо скопировать их в `pg_wal/`. Сегменты WAL, которые не найдутся в архиве, система будет искать в `pg_wal/`; благодаря этому можно использовать последние незаархивированные сегменты. Однако файлы в `pg_wal/` будут менее предпочтительными, если такие сегменты окажутся в архиве;
7) для восстановления данных из инкрементных копий (архивных WAL-файлов) в конфигурационном файле сервера написать команду восстановления, например (для Windows):

   ```
   restore_command = 'copy "PATH\\%f" "%p"'
   ```

   где `PATH` – это путь к каталогу с архивными файлами, который был настроен в переменной `archive_command`. Для восстановления данных до заданного момента времени можно использовать параметр `recovery_target_time`;

8) создать пустой файл в директории сервера с названием `recovery.signal`;
9) запустить сервер. Сервер запустится в режиме восстановления и начнёт считывать необходимые ему архивные файлы WAL. По завершении процесса восстановления сервер удалит файл `recovery.signal` (чтобы предотвратить повторный запуск режима восстановления), а затем перейдёт к обычной работе с базой данных.

5.7 Резервное копирование в MongoDB

В MongoDB существует несколько вариантов резервного копирования кластеров:

– MongoDB Atlas, официальный сервис облачных вычислений MongoDB, предоставляет как непрерывное резервное копирование, так и снимки облачных провайдеров. Непрерывное резервное копирование требует инкрементного резервного копирования данных в кластере, поэтому резервные копии обычно отстают от операционной системы всего на несколько секунд. Снимки облачного провайдера предоставляют локализованное хранилище резервных копий с использованием функционала снимков кластера (например, Amazon Web Services, Microsoft Azure или Google Cloud Platform). Лучшее решение для резервного копирования для большинства сценариев – непрерывное резервное копирование;

– MongoDB также обеспечивает возможность резервного копирования с помощью Cloud Manager и Ops Manager. Cloud Manager – это служба резервного копирования, мониторинга и автоматизации для MongoDB. Ops Manager – это локальное решение, функциональность которого аналогична Cloud Manager;

– для администраторов, управляющих кластерами MongoDB напрямую, существует несколько стратегий резервного копирования, включающих в т.ч. физическое и логическое резервное копирование данных.

5.7.1 Физическое резервное копирование в MongoDB

Первая стратегия – это создание снимка файловой системы. Для резервного копирования путем создания снимка файловой системы используются инструменты системного уровня, такие как LVM – подсистема операционных систем Linux и OS/2, позволяющая использовать разные области одного жёсткого диска и/или области с разных жёстких дисков как один логический том.

Снимки создают образ всего диска или тома. Если выполнять резервное копирование всей системы не требуется, следует рассмотреть возможность изоляции файлов данных MongoDB, журнала (если это применимо) и конфигурирования на одном логическом диске, который не содержит никаких других данных.

Чтобы обеспечить согласованность резервной копии, база данных должна быть заблокирована, а все записи в базу данных должны быть приостановлены во время процесса резервного копирования.

Чтобы сбросить записи на диск и «заблокировать» базу данных, необходимо выполнить метод `db.fsyncLock()`. Чтобы разблокировать базу данных после создания моментального снимка, используется метод `db.fsyncUnlock()`.

Вторая стратегия – это копирование файлов данных. Схема похожа на резервное копирование этого типа в MySQL и PostgreSQL – база данных блокируется командой `fsyncLock`, затем происходит копирование файлов с данными средствами операционной системы, затем база разблокируется командой `fsyncUnlock`.

5.7.2 Логическое резервное копирование в MongoDB

Логическое резервное копирование выполняется с использованием утилиты `mongodump`. Достоинства и недостатки использования этой утилиты остаются теми же, что и при выполнении логического копирования в рассмотренных раннее SQL СУБД:

– медленная работа как при получении резервной копии, так и при восстановлении из нее по сравнению с физическим резервным копированием;

– существует ряд проблем, связанных с резервным копированием набора реплик: помимо данных необходимо также фиксировать состояние набора реплик, чтобы обеспечить точный снимок развертывания.

Однако у логического резервного копирования есть некоторые преимущества:

– это хороший способ для резервного копирования отдельных баз данных, коллекций и даже подмножеств коллекций;

– нет требования остановки сервера.

Для резервного копирования данных из экземпляра `mongod` используется команда `mongodump`.

В следующем примере выполняется сохранение логической резервной копии в выходной каталог `/data/backup/` (предполагается, что сервер работает локально на порту по умолчанию 27017):

```
$> mongodump --out=/data/backup/
```

Резервная копия данных организована в папки и подпапки по базам данных и коллекциям. Фактические данные хранятся в файлах с расширением `.bson`, которые содержат каждый документ коллекции в формате BSON. Файлы `.bson` можно просматривать с помощью утилиты `bsondump`, которая поставляется в комплекте с MongoDB.

Чтобы ограничить объем данных, включенных в дамп базы данных, можно указать параметры `--db` и `--collection`.

В следующем примере создается копия коллекции с именем `inventory` из базы данных `db_lectures`:

```
$> mongodump --collection=inventory --db=db_lectures
```

Чтобы запустить `mongodump` для создания резервной копии MongoDB с включенным контролем доступа, у пользователи должны быть привилегии, предоставляющие выполнять метод `find` для каждой резервной копии базы данных. Встроенная роль `backup` предоставляет необходимые привилегии для выполнения резервного копирования любых баз данных.

```
$> mongodump --host=mongodb1.example.net --
port=37017
      --username=user --authenticationData-
base=admin
      --out=/opt/backup/mongodump-2022-11-02
```

Поддерживается также запись резервной копии в архив, сжатие данных и другие параметры.

Для восстановления из резервной копии `mongodump` используется утилита `mongorestore`:

```
$> mongorestore --port=<порт> <путь к дампу>
```

Чтобы восстановить данные в MongoDB с включенным контролем доступа, пользователю необходимо назначить роль `restore`, которая предоставляет необходимые привилегии для восстановления данных из резервных копий.

```
$> mongorestore --host=mongodb1.example.net
      --port=37017 --username=user
      --authenticationDatabase=admin
      /opt/backup/mongodump-2022-11-02
```

MongoDB также предоставляет возможности резервного копирования и восстановления набора реплик и разделенного кластера.

6 РЕПЛИКАЦИЯ. БАЛАНСИРОВКА НАГРУЗКИ

6.1 Понятие репликации

Серверы баз данных могут работать совместно для обеспечения возможности быстрого переключения на другой сервер в случае отказа первого (т.е. обеспечивать отказоустойчивость) или для обеспечения возможности нескольким серверам БД обрабатывать один набор данных (выполняя балансировку нагрузки.

Репликацией называется набор технологий копирования и распространения данных и объектов баз данных между базами данных и последующей синхронизации баз данных для поддержания их согласованности.

Серверы баз данных только для чтения могут быть совмещены достаточно легко, а серверы с доступом на чтение и запись совместить уже сложнее. Это объясняется тем, что данные только для чтения достаточно один раз разместить на каждом сервере, а запись на любой из серверов должна распространиться на все остальные серверы, чтобы будущие запросы на чтение возвращали согласованные результаты. Если запись данных допускается на нескольких серверах, то еще одной проблемой становится разрешение конфликтов изменения – когда два и более участника репликации изменяют одну и ту же строку данных. Использование механизмов репликации позволяет решить проблемы синхронизации данных между серверами БД.

Фактически использование репликации позволяет осуществить хранение копии одних и тех же данных на нескольких физических серверах в одной сети, каждую такую копию называют репликой. Такой подход позволяет решать ряд практических задач:

– повышение производительности системы. Один сервер может не справляться с нагрузкой, вызываемой одновременными операциями чтения и записи в базе данных. Наличие реплик позволяет распределять запросы между ними, тем самым снизив нагрузку на каждый отдельный сервер;
– обеспечение отказоустойчивости. В случае отказа или необходимости обслуживания одного из серверов можно быстро восстановить работоспособность системы, переведя его нагрузку на одну из реплик на время его бездействия;
– незаметное резервирование данных. Даже если процедура резервного копирования не требует остановки сервера или блокировки таблиц – это в любом случае достаточно ресурсоемкий процесс для сервера. При наличии реплики выполнять резервное копирование можно на ней – это позволит резервировать данные без снижения производительности всей системы;
– серверы аналитики. Ресурсоемкие задачи анализа данных можно выполнять на выделенной реплике без влияния на производительность всей системы;
– снижение транспортных издержек. В распределенных системах значительный вклад в общее время обработки запросов вносит время передачи данных по сети. Уменьшить это время можно, направляя запросы клиентов на наименее удаленные реплики.

6.2 Типы репликации

Проблема синхронизации является главным препятствием для совместной работы серверов. Так как единственного решения, устраняющего проблему синхронизации во всех случаях, не существует, то используются различные решения, которые подходят к

проблеме по-разному и минимизируют её влияние в разных рабочих условиях. Можно выделить синхронную, асинхронную и полусинхронную репликацию.

При **синхронной репликации** транзакция, модифицирующая данные, считается подтверждённой только тогда, когда все серверы подтвердят эту транзакцию. При синхронной репликации гарантировано то, что при отказе ведущего сервера не произойдёт потери данных, а также то, что все серверы возвращают согласованные данные вне зависимости от того, к какому серверу был запрос. Синхронная репликация – это максимально надежный вариант, но и максимально медленный.

Асинхронная репликация. При таком типе репликации транзакция, модифицирующая данные, считается подтверждённой тогда, когда она подтверждена локально на том сервере, на котором эти изменения происходили. При асинхронной репликации высока вероятность потери данных при отказе ведущего сервера, а также нет никаких гарантий согласованности данных в произвольный момент времени, т.к. изменения могут дойти до других серверов с большой задержкой или не дойти вовсе. Асинхронная репликация – это максимально быстрый вариант, но и максимально ненадежный.

При **полусинхронной репликации** транзакция, модифицирующая данные, считается подтверждённой тогда, когда хотя бы один из других серверов подтвердит получение (но не применение) изменений. Такой тип репликации является неким компромиссным решением. При полусинхронной репликации вероятность потери данных при отказе ведущего сервера крайне мала (только если откажет и сервер, подтвердивший получение изменений), но и гарантии согласованности данных в произвольный момент времени также нет, т.к. изменения могут быть применены с задержкой. Полусинхронная репликация – это более быстрый тип репликации, чем синхронная и более надежный, чем асинхронная.

6.3 Виды топологии репликации

В контексте репликации у серверов могут быть две роли: ведущий сервер и подчиненный (ведомый) сервер. При репликации данные копируются с ведущего сервера на подчиненные серверы. Количество серверов с этими ролями определяет различные виды топологии репликации:

- при репликации с одним ведущим сервером данные всегда отправляются на один конкретный узел;
- при репликации с несколькими ведущими серверами может существовать несколько узлов, играющих роль ведущих, и каждый ведущий узел должен сохранять данные в кластере;
- при репликации без ведущих сервером ожидается, что все узлы могут принимать данные при записи.

6.3.1 Репликация с одним ведущим сервером

Репликация с одним ведущим сервером – это наиболее распространенная топология репликации, в которой один ведущий сервер реплицирует изменения всем подчиненным серверам. В такой топологии клиенты всегда посылают запросы на запись ведущему серверу, а запросы на чтение могут распределяться между несколькими подчиненными серверами. Схема репликации с одним ведущим сервером представлена на рисунке 11.

Как показано на рисунке, клиенты выполняют запросы обновления данных `INSERT`, `UPDATE`, `DELETE` только на один ведущий узел, который распределяет изменения по подчиненным серверам. При этом запросы `SELECT` на чтение данных могут выполняться с ведомых узлов.

Метод репликации с одним ведущим узлом очень популярен благодаря своей простоте. Основным преимуществом такой топо-

логии наличия одного лидера является то, что можно избежать конфликтов, вызванных параллельными записями, т.е. не возникает конфликтов согласованности в данных, так как все операции записи исходят из одного узла. Кроме того, если предположить, что все операции детерминированные, они приведут к одинаковым результатам на каждом узле.

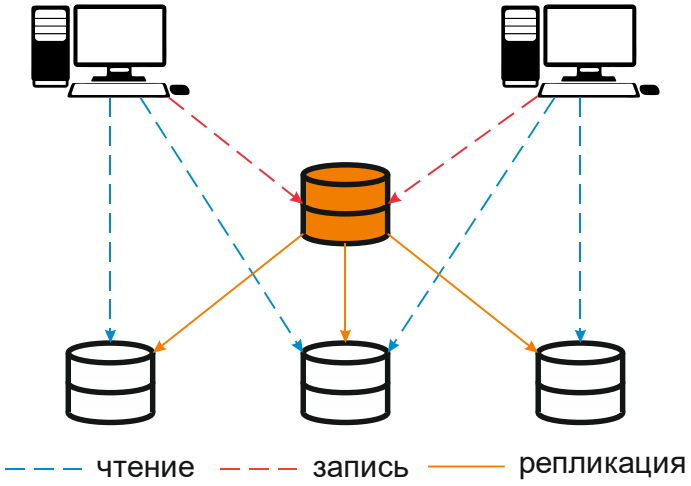

Рисунок 11 – Схема репликации с одним ведущим сервером

Проблемой данной топологии может оказаться то, что один ведущий сервер не способен обработать все запросы на запись данных. Однако большинство приложений выполняет значительно больше запросов на чтение, чем на запись, поэтому возникновение такой ситуации маловероятно. Другая проблема заключается в том, что в случае значительно удаленных серверов возможна ситуация, когда чтение будет происходить быстро – из локальной копии данных на ближайшем подчиненном сервере, а запись с задержкой – на удаленном ведущем сервере.

Метод репликации с одним ведущим узлом используется многими базами данных, такими как PostgreSQL, MongoDB, MySQL и другими.

6.3.2 Репликация с несколькими ведущими серверами

Основной причиной для рассмотрения топологии репликации с несколькими ведущими серверами является наличие более одного узла, обрабатывающего запросы записи, что позволяет решить проблемы, возникающие при топологии с одним ведущим сервером. Схема топологии продемонстрирована на рисунке 12.

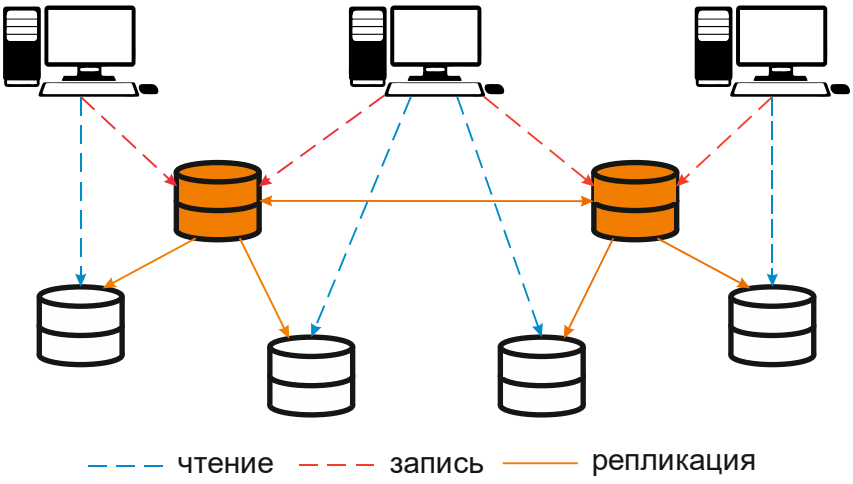

Рисунок 12 – Схема репликации с несколькими ведущими серверами

Если приложению необходимо обрабатывать большое число запросов на запись, то возможно распределить эти запросы между несколькими ведущими серверами. Также, если в случае с удаленными серверами задержка записи неприемлема, то появляется возможность обрабатывать запросы на запись на наименее удаленном от клиента ведущем сервере. Еще одним вариантом использования

такой топологии является поддержка офлайн-клиентов, которые ведут свою локальную копию базы данных, которая периодически синхронизируется с остальными копиями этой базы данных.

Основная проблема топологии репликации с несколькими ведущими серверами заключается в возникновении конфликтов изменения данных и необходимости их разрешения. Конфликты в такой топологии могут возникать только при условии асинхронной репликации. Теоретически, имея несколько ведущих серверов синхронной репликации, можно избежать конфликтов, но такой подход не имеет большого смысла, так как в этом случае фактически утрачивается масштабируемость запросов на запись – то, ради чего данная топология и рассматривается.

Если структура данных позволяет разделить их на непересекающиеся множества и гарантировать, что каждый ведущий сервер работает на запись только со своей частью данных, а остальные данные использует на чтение, то возникновения конфликтов можно избежать. К сожалению, не все информационные системы позволяют разделить свои данные подобным образом, и тогда приходится применять механизмы разрешения конфликтов.

В случае, когда нельзя избежать потенциальных конфликтов записи, необходимо решить, как ими управлять, когда они все же произойдут. Нельзя просто применить на сервере операции записи, относящиеся к одной и той же строке данных в том порядке, в котором они поступают на этот сервер, так как это может привести к рассогласованию данных. Это связано с тем, что операции передаются по сети, и на разные серверы они приходят с различной задержкой и, соответственно, в разном порядке. Для разрешения подобных ситуаций в различных СУБД существуют различные механизмы разрешения конфликтов: одни присваивают каждой операции записи временные метки и затем всегда применяется по-

следняя запись (стратегия Last Write Wins), другие позволяют применять пользовательский код разрешения конфликтов и т.д.

Репликация с несколькими ведущими узлами обычно реализуется с помощью внешних инструментов, таких как Tungsten Replicator для MySQL, BDR для PostgreSQL.

6.3.3 Репликация без ведущих серверов

Топология репликации без ведущих серверов характеризуется отсутствием ведущих серверов и тем фактом, что каждая реплика (или сервер) может обрабатывать запросы на запись. Схема построения изображена на рисунке 13.

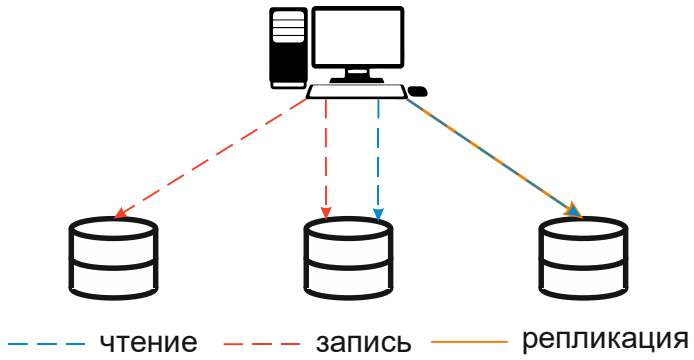

Рисунок 13 – Схема репликации без ведущих серверов

Основная идея заключается в том, что клиент посылает запрос на запись одновременно нескольким репликам, и как только он получает подтверждение от некоторых из них (в предельном случае от одной), считается, что запись прошла успешно и клиент может продолжать работу. Эта, казалось бы, простая идея позволяет значительно уменьшить последствия сбоев в работе серверов в сравнении с предыдущей топологией. Если в топологии с несколькими ведущими серверами сервер, которому послали запрос на запись, в силу

каких-то причин не подтверждает его выполнения в течение заданного таймаута, то необходимо начать процедуру отработки отказа для выбора другого ведущего сервера и посылки ему повторного запроса. В данной же топологии нет необходимости в процедуре отработки отказа, поскольку все реплики равноправны, и не надо ждать подтверждения выполнения запроса от одного конкретного сервера.

В этой же идее кроется и определенная проблема: пусть запрос на запись будет завершен удачно на двух репликах из трех, а на одной не завершится. В этом случае будут две реплики с новым значением и одна со старым, и запросы чтения к разным репликам будут возвращать разные значения. В отсутствии ведущего сервера, обеспечивающего синхронизацию, проблема рассогласования данных решается следующим образом:

1) клиент посылает запросы на чтение также на несколько реплик одновременно;
2) реплики возвращают свои локальные значения и некоторый номер версии данных, который используется клиентом, чтобы решить, какое из полученных значений является актуальным;
3) в это же время выполняется и синхронизация значений между репликами: если при чтении данных обнаруживается (по номеру версии), что на некотором узле значение не актуально, клиент посылает этому узлу запрос на запись с актуальным значением. Такой механизм называют чтением с восстановлением (Read Repair).

При выборе числа реплик, вовлеченных в процессы чтения и записи, необходимо учитывать режим работы с данными, а также кворумы чтения и записи, т.е. минимальное количество узлов чтения или записи, требуемое для обеспечения согласованности данных.

Riak и Cassandra являются примерами баз данных, использующих стратегии репликации без лидера.

6.4 Балансировка нагрузки

Репликацию можно использовать для масштабирования в разных ситуациях. Основная цель – создать и поддерживать резервную базу данных на случай выхода системы из строя. Особенно это относится к физической репликации. Однако репликацию можно использовать и для повышения производительности.

Рассмотрим систему, рассчитанную на обработку очень большого количества запросов на чтение. Например, это может быть приложение, реализующее оконечную точку HTTP API, которая призвана поддержать автозавершение на веб-сайте. Всякий раз, как пользователь вводит символ в форму, система ищет в базе данных объекты, названия которых начинаются с введенной строки. Таких запросов может быть очень много, потому что одновременно работает много пользователей, каждый из которых порождает все новые запросы. Чтобы справиться с большим числом запросов, база данных должна задействовать несколько процессорных ядер.

То же относится к системе, которая должна обрабатывать несколько «тяжелых» запросов одновременно. Даже если запросов немного, но каждый из них сложен, задействование максимально возможного количества процессоров может дать выигрыш в производительности. Особенно если выполнение запросов распараллеливается.

В ситуациях, когда одна база данных не справляется с нагрузкой, можно развернуть несколько баз данных, настроить репликацию в них главной базы и организовать приложение так, чтобы оно направляло разные запросы разным базам. Приложение можно научить запрашивать данные у разных баз, но для этого необходима

специальная архитектура уровня доступа к данным, например, такая, как показано на рисунке 14.

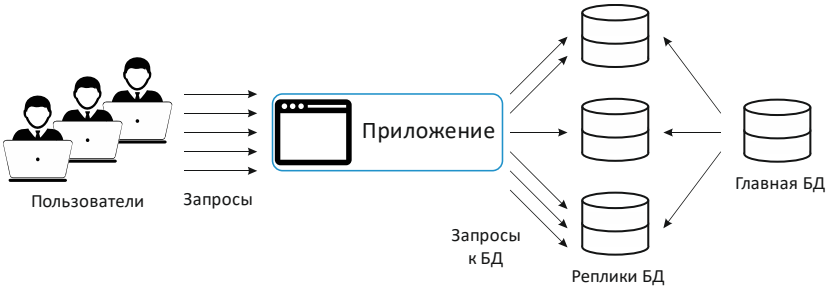

Рисунок 14 – Архитектура уровня доступа к данным через разные БД

Другой вариант – использование балансировщика нагрузки между БД, например, MySQL Proxy для MySQL, Pgpool-II для PostgreSQL и т.д. Такие программы обычно предоставляют SQL-интерфейс, так что приложения могут работать как настоящий сервер БД. Балансировщик направляет запросы базам, у которых в данный момент меньше всего запросов, т.е. балансирует нагрузку (рисунок 15).

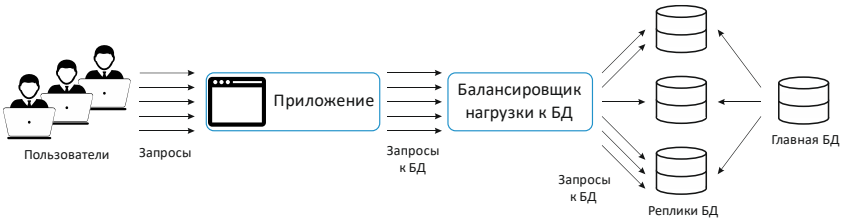

Рисунок 15 – Архитектура уровня доступа
к данным через балансировщик нагрузки

Еще одна возможность – масштабировать приложение вместе с базами данных так, чтобы каждый экземпляр приложения подключался к своему экземпляру базы данных. В таком случае поль-

зователь должен подключаться к одному из нескольких экземпляров, что легко реализовать с помощью балансировщика HTTP-нагрузки, как показано на рисунке 16.

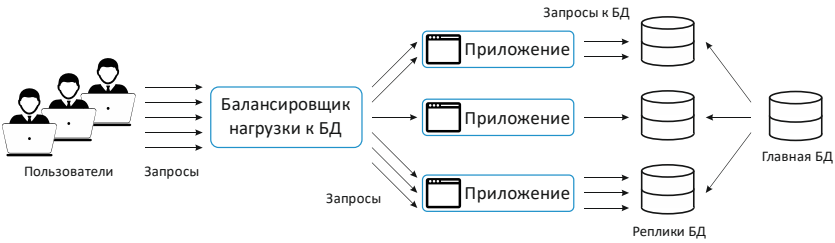

Рисунок 16 – Архитектура уровня доступа
к данным через балансировщик HTTP-нагрузки

6.5 Механизмы репликации в MySQL

В модели репликации MySQL выделяют два ключевых физических уровневых компонента:

– ведущий сервер (source или master) – сервер, предоставляющий данные;

– подчиненный сервер (replica или slave) – сервер, копирующий себе данные, предоставляемые ведущим сервером.

Для краткости изложения ведущий сервер будем называть мастером, а подчиненный – репликой. Фактически, в MySQL реализован один тип репликации, основанный на распространении данных двоичных журналов транзакций.

6.5.1 MySQL. Репликация двоичных журналов

Этот тип репликации основан на том, что все изменения данных, происходящие на мастере, после первоначальной синхронизации мастера и реплик повторяются на репликах (но не наоборот).

Поэтому запросы на изменение данных выполняются только на мастере, а запросы на чтение данных могут выполняться как на репликах, так и на мастере. Процесс репликации на одной реплике не влияет на работу других реплик и практически не влияет на работу мастера. Таким образом, в MySQL это тип репликации с одним ведущим сервером.

Первоначальная синхронизация данных выполняется с помощью резервной копии данных мастера. Репликация данных производится при помощи двоичного журнала, ведущегося на мастере. На каждом подчиненном сервере есть так называемый канал репликации, который служит для передачи информации об изменениях данных от мастера к реплике. С каждым каналом репликации ассоциирован приемник данных (поток ввода/вывода), один или несколько исполнителей команд (поток исполнения SQL) и журнал репликации (Relay Log). С помощью приемника данных информация об изменениях из двоичного журнала на мастере копируется в журнал репликации на подчиненном сервере. Далее, с помощью исполнителя команд события в журнале репликации воспроизводятся на данных реплики, тем самым происходит согласование данных мастера и реплик. В целях повышения безопасности процесса репликации есть возможность шифрования двоичных журналов и журналов репликации алгоритмом AES, а также использования зашифрованного соединения для передачи данных двоичного журнала.

Сервером также поддерживается репликация с множеством источников (Multi-Source Replication) которая позволяет подчиненному серверу параллельно получать транзакции от нескольких ведущих. При таком типе репликации на подчиненном сервере создаётся несколько каналов репликации, по одному для каждого мастера, что позволяет реплицировать данные независимо.

При репликации передаются не сами измененные данные, а только информация, необходимая для воспроизведения этих изменений. Существуют разные способы передачи этой информации об изменениях. В настоящее время в MySQL поддерживаются три формата ведения двоичного журнала:

– протоколирование выражений (Statement-based Logging). При использовании этого формата события в журнале представляют собой SQL-выражения, которые были выполнены при модификации данных;

– протоколирование записей (Row-based Logging). В этом случае события в журнале представляют собой изменения в отдельных записях таблицы данных в двоичном формате;

– смешанное протоколирование (Mixed Logging). Часть событий представлено в виде SQL-операторов, часть – в виде изменений отдельных записей.

Каждый режим протоколирования имеет свои плюсы и минусы. Протоколирование выражений, как правило, более компактно, особенно когда речь идет о массовых изменениях данных, например, `UPDATE` запрос, изменяющий тысячи записей. Также, поскольку файлы журнала содержат все выражения, изменявшие данные, такой журнал может использоваться для аудита БД. С другой стороны, при воспроизведении изменений придется повторять вычисление ресурсоемких запросов на реплике, что не лучшим образом повлияет на скорость синхронизации. Кроме того, SQL-выражения могут содержать ряд недетерминированных функций, например `NOW()`, `RAND()`, `USER()`, и исполнение этих выражений на мастере и на реплике может привести к разным результатам и, соответственно, к рассогласованию данных.

С протоколированием записей обратная ситуация – такой режим безопасен с точки зрения согласованности данных, так как реплицируются непосредственно измененные строки данных, и они

одинаковы, независимо от сервера, на котором эти изменения воспроизводятся. С другой стороны, использование этого режима приводит к быстрому росту объема двоичного журнала.

Режим смешанного протоколирования является некоторой золотой серединой, объединяющей плюсы обоих режимов. По умолчанию в этом режиме используется протоколирование выражений, но в определенных случаях, задаваемых настройками, применяется режим протоколирования записей.

Данный тип репликации может быть сконфигурирован для выполнения как в асинхронном, так и в полусинхронном режимах, полностью синхронного решения для репликации в настоящий момент MySQL не предлагает.

6.5.2 MySQL. Пример настройки репликации

Рассмотрим пример настройки процесса репликации. Допустим, что уже созданы два сервера, которые могут работать на разных компьютерах в сети или на разных портах на одном компьютере. Настройка репликации состоит из следующих шагов:

– в файлы конфигурации серверов (мастера и ведомых) необходимо добавить название БД для репликации и уникальный id сервера (например, 1 для главного сервера и 2 – для ведомого):

```
server-id=1
replicate-do-db=database_name
```

– на главном сервере необходимо создать нового пользователя, через которого будет осуществляться репликация, и выдать ему соответствующие права;

– выполнить настройку репликации на ведомом сервер. Для этого на сервере необходимо выполнить команду:

```
CHANGE MASTER TO MASTER_HOST = "localhost", MAS-
TER_PORT=3307, MASTER_USER = "replication", MAS-
TER_PASSWORD = "password";
```

В команде указывается адрес главного сервера репликации и данные пользователя главного сервера, через которого осуществляется репликация;

– выполнить перенос БД главного сервера на ведомый сервер. Для этого с помощью утилиты `mysqldump` необходимо создать резервную копию БД, включив в нее информацию о бинарном лог-файле (параметр `--source-data`). Восстановить резервную копия на ведомом сервере;

– перезапустить ведомый сервер в режиме «только чтение». Для этого в файл конфигурации ведомого сервера необходимо добавить строку `super_read_only=1` для того, чтобы осуществлять запись в БД мог только главный сервер репликации. Репликацию можно запустить вручную, для этого необходимо запустить сервер с параметром `--skip-slave-start`, чтобы репликация не начиналась автоматически с момента запуска сервера, затем запустить репликацию на ведомом сервере командой `START SLAVE;`

6.6 Механизмы репликации в PostgreSQL

В модели репликации PostgreSQL, как и в MySQL, выделяют два ключевых физических уровневых компонента:

– ведущий сервер (source / master) – сервер, предоставляющий данные;

– подчиненный сервер (replica / slave) – сервер, копирующий себе данные, предоставляемые ведущим сервером.

В PostgreSQL реализовано два типа репликации:

– физическая (или потоковая) репликация – это трансляция журнала упреждающей записи (Write-Ahead Log Shipping);

– логическая репликация.

6.6.1 Физическая репликация в PostgreSQL

Физическая репликация основана на передаче на реплику изменений в виде записей журнала упреждающей записи. Это очень эффективный механизм, но он требует, чтобы между серверами была двоичная совместимость (основная версия сервера, операционная система, аппаратная платформа). Физическая репликация всегда однонаправленна: в ней может существовать только один мастер и произвольное число реплик. Следовательно, это тип репликации с одним ведущим сервером.

Механизм репликации очень похож на оперативное резервирование данных, с той разницей, что резервная копия, восстановленная на реплике, работает в режиме постоянного восстановления (`standby_mode=on`) и непрерывно читает и применяет новые сегменты WAL, поступающие с мастера. Таким образом, реплика постоянно поддерживается в почти актуальном состоянии и в случае сбоя есть сервер, готовый продолжить работу. Если реплика не допускает клиентских подключений, она называется сервером «теплого резерва». Однако можно сделать и сервер «горячего резерва», тогда в процессе восстановления реплика будет допускать подключения, но естественно только на чтение данных.

Есть два способа доставки журналов от мастера к реплике:

– потоковая репликация. В этом случае реплика подключается к мастеру по протоколу репликации и читает поток записей WAL. За счет этого при потоковой репликации отставание реплики сведено к минимуму или даже к нулю при синхронном режиме. При использовании потоковой репликации есть опасность, что мастер удалит сегмент WAL, данные из которого еще не переданы на реплику, для уверенности стоит применять потоковую репликацию вместе с архивом WAL;

– репликация с архивом WAL. При использовании архива специальный процесс на мастере записывает заполненные сег-

менты журнала в архив. Если реплика не может получить очередную журнальную запись по протоколу репликации, она попробует прочитать ее из архива. В принципе, репликация может работать и с одним только архивом, без потоковой репликации, но в этом случае реплика всегда будет отставать от мастера на время заполнения сегмента.

Физический тип репликации по умолчанию работает в асинхронном режиме, но возможно настроить ведомый сервер на работу в синхронном режиме. Причем в случае нескольких реплик часть могут быть синхронными (резервный мастер), часть – асинхронными (масштабирование чтения).

При асинхронной репликации запрос немедленно выполняется на мастере, а соответствующие данные из WAL передаются к серверам-репликам в фоне. Недостаток асинхронной репликации заключается в том, что при внезапном выходе из строя мастера (например, из-за сгоревшего диска) часть данных будет потеряна, так как они не были переданы на реплики.

При использовании синхронной репликации данные сначала записываются в WAL как минимум одной реплики, после чего транзакция выполняется уже на мастере. Запросы на запись выполняются медленнее в результате возникающих сетевых задержек (которые, однако, внутри одного датацентра обычно меньше типового времени выполнения запроса). Кроме того, чтобы запросы на запись продолжили выполняться в результате выхода из строя одной из реплик, при использовании синхронной репликации рекомендуется использовать по крайней мере две реплики. Преимуществом такого подхода является б*о*льшая надежность.

Несколько реплик можно поддерживать без создания дополнительной нагрузки на ведущий сервер, используя технологию каскадной репликации. Суть этой технологии состоит в том, что одна реплика передает записи WAL другой реплике и так далее. При

необходимости иметь копию данных на некоторый момент в прошлом можно сконфигурировать реплику, которая применяет записи WAL не сразу, а через установленный интервал времени.

6.6.2 Логическая репликация в PostgreSQL

Основным недостатком физической репликации является требование двоичной совместимости, что накладывает определенные ограничения на её использование. В случае, когда невозможно использовать физическую репликацию, альтернативой является использование логической репликации.

Логическая репликация – это метод репликации объектов, данных и изменений в них, использующий «репликационные идентификаторы». В логической репликации используется модель публикаций/подписок с одним или несколькими подписчиками, которые подписываются на одну или несколько публикаций на публикующем узле. Подписчики получают данные из публикаций, на которые они подписаны, и могут затем повторно опубликовать данные для организации каскадной репликации или более сложных конфигураций.

Логическая репликация таблицы обычно начинается с создания снимка данных в публикуемой базе данных и копирования её подписчику, при этом таблицы-приёмники на стороне подписчика должны уже существовать (обычно это делается при первоначальной синхронизации с использованием резервных копий). После этого изменения на стороне публикующего сервера передаются подписчику в реальном времени, когда они происходят. Подписчик применяет изменения в том же порядке, что и узел публикации, так что для публикаций в рамках одной подписки гарантируется транзакционная целостность. Этот метод репликации данных иногда называется транзакционной репликацией.

Под **публикацией** в логической репликации понимается набор изменений, выделяемых в таблице или в группе таблиц. Публикация определяется только в рамках одной базы данных. Сервер, на котором определяется публикация, называется публикующим. Публикации могут ограничивать набор публикуемых изменений, выбирая любое сочетание операций из `INSERT`, `UPDATE` и `DELETE`. По умолчанию реплицируются все типы операций модификации данных. Чтобы можно было реплицировать операции `UPDATE` и `DELETE`, в публикуемой таблице должен быть настроен «репликационный идентификатор» для нахождения соответствующих строк для изменения или удаления на стороне подписчика, команды `INSERT` могут выполняться вне зависимости от такого идентификатора. По умолчанию репликационным идентификатором является первичный ключ таблицы, при необходимости репликационным идентификатором можно назначить другое уникальное поле. Если в таблице нет подходящего ключа, в качестве репликационного идентификатора можно задать «`full`», что будет означать, что ключом будет вся строка, однако такой подход является очень неэффективным и должен применяться, только если нет другого решения. У каждой публикации может быть множество подписчиков.

Подписка – это принимающая сторона логической репликации. Узел, на котором определяется подписка, называется подписчиком. В свойствах подписки определяется набор публикаций, данные из которых подписчик будет получать. База данных подписчика работает так же, как и экземпляр любой другой базы PostgreSQL, и может быть публикующей для других баз, если в ней определены собственные публикации. В одной паре публикующий сервер/подписчик могут быть определены несколько подписок, но при этом публикуемые объекты в разных подписках не должны пересекаться. При синхронизации таблицы публикации сопоставляются с табли-

цами подписчика по полностью заданным именам таблиц. Репликация в таблицы с другими именами на стороне подписчика не поддерживается. Столбцы таблиц также сопоставляются по именам. Другой порядок столбцов в целевой таблице допускается, но типы столбцов должны совпадать. Целевая таблица может содержать дополнительные столбцы, отсутствующие в публикуемой таблице. Они будут заполнены установленными для них значениями по умолчанию.

Логическая репликация работает подобно обычным операциям модификации данных в том смысле, что данные на подписчике будут изменены, даже если они изменялись на нём автономно. Если входящие данные нарушают какие-либо ограничения, репликация остановится. Эта ситуация называется конфликтом. При репликации операций `UPDATE` или `DELETE` отсутствие данных не вызывает конфликта, так что такие операции просто пропускаются. В случае конфликта выдаётся ошибка и репликация останавливается; разрешить возникшую проблему пользователь должен вручную. Подробности конфликта можно найти в журнале сервера-подписчика. Разрешение конфликта может заключаться либо в изменении данных на стороне подписчика, чтобы они не конфликтовали с приходящим изменением, либо в пропуске транзакции, конфликтующей с существующими данными.

Технически информация об измененных строках извлекается и декодируется из имеющегося журнала WAL на публикующем сервере, а затем пересылается процессом wal sender подписчику по протоколу репликации в формате, независимом от платформы и версии сервера. Процесс logical replication worker на подписчике принимает информацию и применяет изменения. Применение изменений происходит без выполнения команд SQL и связанных с этих накладных расходов на разбор и планирование, что уменьшает нагрузку на подписчика.

Типовые сценарии использования логической репликации:

– передача подписчикам инкрементальных изменений в данных, когда они происходят;

– срабатывание триггеров для отдельных изменений, когда их получает подписчик;

– объединение нескольких баз данных в одну (например, для целей анализа);

– репликация между разными основными версиями PostgreSQL;

– репликация между экземплярами PostgreSQL на разных платформах (например, с Linux на Windows).

6.6.3 PostgreSQL. Пример настройки физической репликации

Рассмотрим пример настройки физической репликации в PostgreSQL, состоящий из следующих шагов:

1) настроить архивацию WAL (для репликации с архивом WAL). Для этого в конфигурационном файле сервера postgresql.conf указать параметры:

```
archive_mode = ON;
archive_command = 'copy "%p" "PATH\\%f"' # Пример команды архивации для операционной системы Windows
archive_timeout = 60 # время существования не архивированных данных в секундах
```

2) на главном сервере создать нового пользователя, через которого будет осуществляться репликация, и дать ему соответствующие права;
3) создать копию главного сервера с помощью утилиты `pg_basebackup`:

```
pg_basebackup -p master_port -U replicator -D path_to_slave -Fp -Xs -P -R
```

4) в конфигурационном файле главного и ведомого серверов указать команду восстановления из архива WAL. В команде восстановления необходимо указать путь к архиву с WAL файлами:

```
restore_command = 'copy "PATH\\%f" "%p"' # Пример
команды для Windows
```

5) запустить серверы.

6.7 Репликация в MongoDB

В MongoDB репликация настраивается путем создания набора реплик. **Набор реплик** – это группа серверов с одним первичным узлом, который получает операции записи, и несколькими вторичными узлами, где хранятся копии данных первичного узла.

Если первичный узел дает сбой, вторичные узлы могут выбрать новый первичный узел из их числа. Если используется репликация и сервер выходит из строя, можно получить доступ к данным с других серверов в наборе реплик. Если данные на сервере повреждены или недоступны, можно сделать новую копию данных с одного из других членов набора реплик.

Репликация в MongoDB основана на двух механизмах: журнале операций (oplog) и тактовом сигнале (heartbeat). Журнал операций делает репликацию возможной, а с помощью тактовых сигналов ведется мониторинг состояния и активируется процедура обработки отказа.

Журнал операций представляет собой ограниченную коллекцию в базе `local` на каждом узле, в эту коллекцию записываются все изменения данных, необходимые для воспроизведения операции.

Механизм тактовых сигналов позволяет обеспечить отработку отказа и выбор нового первичного узла. По умолчанию каждый

член набора реплик посылает тактовые сигналы всем остальным членам раз в две секунды. Это позволяет в целом отслеживать состояние системы. Если первичный узел перестает отвечать, то система автоматически выбирает новый первичный узел из вторичных с самым свежим состоянием.

6.7.1 MongoDB. Пример настройки репликации

Приведем пример настройки кластера, запущенного на наборе из трех реплик. На первом шаге необходимо создать каталоги для серверов и выполнить инициализацию набора реплик. Пример команды инициализации серверов из командной строки имеет следующий вид:

```
$> mongod --replSet mdbRep --dbpath c:\rs1 --port 27017
$> mongod --replSet mdbRep --dbpath c:\rs2 --port 27018
$> mongod --replSet mdbRep --dbpath c:\rs3 --port 27019
```

После запуска серверов должно быть запущено три отдельных процесса `mongod`.

Далее необходимо объединить три сервера в один набор реплик. Для этого необходимо создать конфигурацию, в которой перечислены все члены, и отправить ее одному из созданных процессов `mongod`, который передаст эту конфигурации другим членам. Инициализация набора реплик выполняется командой `rs.initiate()`:

```
rsconf = {
    _id: "mdbRep",
    members: [
        { _id: 0, host: "localhost:27017" },
        { _id: 1, host: "localhost:27018" },
```

```
            { _id: 2, host: "localhost:27019" }
        ]
    }
    rs.initiate(rsconf)
```

Чтобы разрешить считывание из вторичных узлов, нужно сообщить об этом к подключенному вторичному узлу, с помощью команды `secondaryOk`:

```
rs.secondaryOk()
```

Проверить отказоустойчивость можно, остановив один из реплицируемых процессов. Драйвер сам начнет брать данных из рабочего процесса.

7 СЕКЦИОНИРОВАНИЕ. СЕГМЕНТИРОВАНИЕ

7.1 Основные понятия

Репликация – это набор технологий копирования и распространения данных и объектов баз данных между базами данных и последующей синхронизации баз данных для поддержания их согласованности. При репликации на разных серверах хранятся полные копии баз данных, которые синхронизируются для поддержания их согласованности.

Если же проблема заключается не в количестве одновременных запросов, а в размере базы данных и скорости выполнения одного запроса, то нужен другой подход. Данные можно разделить либо на разные файловые группы в пределах одного сервера, либо между несколькими серверами, которые будут опрашиваться параллельно, а объединение частичных результатов в окончательный будет производиться за пределами базы данных [12].

Выделяют два подхода к разделению данных:

– **секционирование** (partitioning) – это разделение хранимых объектов баз данных (таких как таблицы, индексы, материализованные представления) на отдельные части с раздельными параметрами физического хранения. Используется в целях повышения управляемости, производительности и доступности для больших баз данных;

– **сегментирование** (sharding) – подход, предполагающий разделение баз данных, отдельных её объектов или индексов поисковых систем на независимые сегменты, каждый из которых управляется отдельным экземпляром сервера базы данных, размещаемым, как правило, на отдельном вычислительном узле.

В отличие от секционирования, предполагающего раздельное хранение частей объектов базы данных под управлением единого

экземпляра СУБД, сегментирование позволяет задействовать технику распределённых вычислений, но при этом более сложно в реализации, так как требует обеспечения координации множества экземпляров таким образом, чтобы взаимодействие со всей совокупностью сегментов велось как с единой базой данных.

7.2 Секционирование

Секционирование данных – это разбиение одной большой логической таблицы на несколько меньших физических секций. Секционирование позволяет распределять части табличных данных (секции / разделы) по файловой системе на основе набора определяемых пользователем правил.

Секционирование реализовано во многих реляционных СУБД, включая PostgreSQL, MySQL, Microsoft SQL Server и другие.

На рисунке 17 приведен пример секционирования, в котором строки одной таблицы хранятся в разных секциях в зависимости от настроенного правила секционирования.

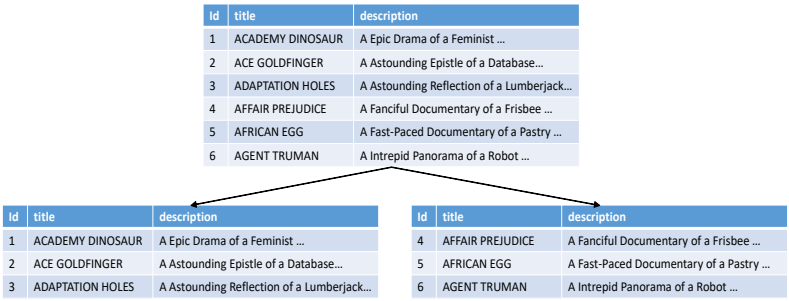

Рисунок 17 – Пример секционирования

Можно отметить следующие особенности применения секционирования таблиц:

– секционирование позволяет хранить в одной таблице больше данных, чем может храниться на одном диске или разделе файловой системы;

– в определённых ситуациях секционирование значительно увеличивает быстродействие, особенно когда большой процент часто запрашиваемых строк таблицы относится к одной или нескольким секциям. Секционирование заменяет верхние уровни деревьев индексов, что увеличивает вероятность нахождения наиболее востребованных частей индексов в памяти;

– массовую загрузку и удаление данных можно осуществлять, добавляя и удаляя секции, если такой вариант использования был предусмотрен при проектировании секций. Удаление отдельной секции выполняется гораздо быстрее, чем аналогичная массовая операция. И наоборот, процесс добавления новых данных в некоторых случаях можно значительно облегчить, добавив один или несколько новых разделов для хранения именно этих данных;

– редко используемые данные можно перенести на более дешёвые и медленные носители.

Секционирование удобно только для очень больших таблиц, так как возникают накладные расходы на чтение данных из разных файлов.

В различных СУБД возможности реализации несколько различаются. Можно выделить следующие типы секционирования:

– по интервалам (range partitioning);
– по списку значений (list partitioning);
– по хешу (hash partitioning);
– по ключам (key partitioning).

Данные типы секционирования поддерживаются в MySQL. Декларативное секционирование в PostgreSQL поддерживает три типа: по интервал, по списку и по ключу.

В некоторых СУБД (включая MySQL и PostgreSQL) поддерживается вложенное секционирование, при котором сами секции также разбиваются на секции.

7.2.1 Секционирование в MySQL

7.2.1.1 Секционирование по интервалам

При секционировании по интервалам строки таблицы ставятся в соответствие разделам на основе интервала, в который попадает значение выражения секционирования (partitioning expression) строки. При этом:

– интервалы значений должны быть смежными, но не должны перекрывать друг друга;

– оператор `VALUES LESS THAN` будет использоваться для определения таких диапазонов в порядке от наименьшего;

– строки, для которых выражение секционирования возвращает значение NULL, попадут в первый раздел;

– секционирования по интервалам может выполняться по нескольким столбцам таблицы.

В следующем примере продемонстрированно создание таблицы с четырьмя разделами, которые определяются с помощью выражения секционирования «`YEAR(created)`»:

```
CREATE TABLE userslogs (
    username VARCHAR(20) NOT NULL,
    logdata BLOB NOT NULL,
    created DATETIME NOT NULL,
    PRIMARY KEY (username, created)
)
PARTITION BY RANGE ( YEAR(created) ) (
    PARTITION p1 VALUES LESS THAN (2014),
    PARTITION p2 VALUES LESS THAN (2015),
    PARTITION p3 VALUES LESS THAN (2016),
    PARTITION p4 VALUES LESS THAN MAXVALUE
);
```

В примере `MAXVALUE` представляет собой целочисленное значение, которое всегда больше, чем максимально возможное целочисленное значение.

7.2.1.2 Секционирование по списку значений

Секционирование по списку значений похоже на секционирование по интервалам, за исключением того, что раздел выбирается на основе соответствия значения выражения секционирования одному из наборов дискретных значений:

– для секционирования по списку значений нет аналога выражения MAXVALUE, которое позволяет покрыть все возможные значения выражения секционирования. По этой причине необходимо учесть все возможные значения выражения секционирования. При попытке добавить значение, которое не покрывается выражением секционирования, возникнет ошибка и запись не добавится;

– значения NULL обрабатываются как любые другие значения;

– секционирование данного типа может выполняться по нескольким столбцам таблицы.

В следующем запросе создается таблица serverlogs, секционирование которой выполнено по двум разделам server_east и server_west:

```
CREATE TABLE serverlogs (
    serverid INT NOT NULL,
    logdata BLOB NOT NULL,
    created DATETIME NOT NULL
)
PARTITION BY LIST ( serverid ) (
    PARTITION server_east VALUES IN (1, 43, 65,
73),
    PARTITION server_west VALUES IN (5, 642, 196,
22)
);
```

При попытке добавить в эту таблицу запись с идентификатором, например, 2, возникнет ошибка, т.к. этот идентификатор не перечислен ни в одном из списков с возможными значениями при определении секций.

7.2.1.3 Секционирование по хешу

Разбиение по хешу используется, прежде всего, для обеспечения равномерного распределения данных между заданным числом секций. При секционировании по диапазону или списку необходимо явно указать, в какой секции должна храниться строка с указанным значением выражения секционирования. С секционированием по хешу это решение принимается СУБД, и необходимо только указать значение столбца или выражение на основе значения столбца, которое нужно хешировать, а также количество секций, на которые должна быть разделена секционированная таблица.

Выражение секционирования должно возвращать не константное неслучайное целочисленное значение. При этом данное выражение вычисляется каждый раз, когда строка вставляется или обновляется. Это означает, что очень сложные выражения могут привести к проблемам с производительностью, особенно при выполнении операций, которые затрагивают большое количество строк одновременно.

В следующем примере выполняется секционирование по хешу:

```
CREATE TABLE serverlogs_hash (
    serverid INT NOT NULL,
    logdata BLOB NOT NULL,
    created DATETIME NOT NULL
)
PARTITION BY HASH ( YEAR(created) )
PARTITIONS 10;
```

В данном случае не указывается явно, какие значения или диапазон значений относятся к той или иной секции таблицы, вместо

этого в примере указывается выражение на основе столбца, для которого будет вычисляться значение хеша. Кроме того, в этом типе секционирования явно указано количество секций – 10.

7.2.1.4 Секционирование по ключу

Секционирование по ключу похоже на секционирование по хешу, за исключением того, что во втором случае используется определяемое пользователем выражение, а в первом случае – секционирование выполняется по указанному ключевому полю:

```
CREATE TABLE serverlogs_key (
    serverid INT NOT NULL,
    logdata BLOB NOT NULL,
    created DATETIME NOT NULL,
    UNIQUE KEY (serverid)
)
PARTITION BY KEY ()
PARTITIONS 10;
```

Здесь выражение «`KEY()`» используется без явного указания столбца таблицы или выражения по которому будет осуществляться секционирование. MySQL будет автоматически использовать первичный ключ или уникальный ключ в качестве выражения секционирования. Если уникальные ключи недоступны, инструкция завершится ошибкой.

7.2.1.5 Отсечение секций

Отсечение или устранение секций (partition pruning) – это приём оптимизации запросов, который ускоряет работу с секционированными таблицами. Оптимизация основана на относительно простой концепции, которую можно описать как «не сканировать разделы, в которых не может быть строк, удовлетворяющих условию запроса».

Без устранения секций показанный запрос должен будет просканировать все секции таблицы. Когда устранение секций включено, планировщик рассматривает определение каждой секции и может заключить, что какую-либо секцию сканировать не нужно, так как в ней не может быть строк, удовлетворяющих предложению `WHERE` в запросе. Когда планировщик может сделать такой вывод, он исключает секцию из плана запроса.

Оптимизатор может выполнять отсечение всякий раз, когда условие `WHERE` может быть сведено к одному из следующих двух случаев:

- `partition_column = constant`
- `partition_column IN (constant1, constant2, ..., constantN)`

Функции над `partition_column` (например, `YEAR(partition_column)`) в запросе не используются.

Операторы `SELECT`, `DELETE` и `UPDATE` поддерживают отсечение секций. Однако, выбор выражений секционирования, поддерживающих отсечение секции, ограничен.

7.2.1.6 Особенности секционирования

Можно отметить следующие особенности секционирования в MySQL:

- внешние ключи: секционированные таблицы не поддерживают внешние ключи. Таким образом, нельзя добавить внешний ключ в секционированную таблицу. И наоборот, если у таблицы есть внешний ключ, ее нельзя секционировать. Кроме того, в несекционированной таблице не может быть внешнего ключа, указывающего на столбец секционированной таблицы;
- индексы: секционирование применяется ко всем данным и индексам в таблице; нельзя секционировать только данные, а не индексы, или наоборот. Также нельзя секционировать

только часть таблицы. Секционированные таблицы не поддерживают полнотекстовые индексы;

– связь с первичными и уникальными ключами таблицы: все столбцы, используемые в выражении секционирования таблицы, должны быть частью каждого уникального ключа данной таблицы. Сюда также относится первичный ключ таблицы, поскольку он по определению является уникальным ключом.

7.2.2 Секционирование в PostgreSQL

PostgreSQL поддерживает два типа секционирования:

– декларативное секционирование;

– секционирование с использованием наследования.

Декларативное секционирование – это более новый тип секционирования. При декларативном секционировании декларируется, что некоторая таблица разделяется на секции. Сама секционированная таблица является «виртуальной» и как таковая не хранится. Хранилище используется её секциями, которые являются обычными таблицами, связанными с секционированной. В каждой секции хранится подмножество данных таблицы, определяемое её границами секции. Все строки, вставляемые в секционированную таблицу, перенаправляются в соответствующие секции в зависимости от значений столбцов ключа разбиения. Если при изменении значений ключа разбиения в строке она перестаёт удовлетворять ограничениям исходной секции, эта строка перемещается в другую секцию.

Сами секции могут представлять собой секционированные таблицы, таким образом реализуется вложенное секционирование. Хотя все секции должны иметь те же столбцы, что и секционированная родительская таблица, в каждой секции независимо от других могут быть определены свои индексы, ограничения и значения по умолчанию.

Преобразовать обычную таблицу в секционированную и наоборот нельзя. Однако в секционированную таблицу можно добавить в качестве секции существующую обычную или секционированную таблицу, а также можно удалить секцию из секционированной таблицы и превратить её в отдельную таблицу; это может ускорить многие процессы обслуживания.

Хотя встроенное декларативное секционирование полезно во многих часто возникающих ситуациях, бывают обстоятельства, требующие более гибкого подхода. В этом случае секционирование можно реализовать, применив механизм наследования таблиц, что даст ряд возможностей, неподдерживаемых при декларативном секционировании, например:

– при декларативном секционировании все секции должны иметь в точности тот же набор столбцов, что и секционируемая таблица, тогда как обычное наследование таблиц допускает наличие в дочерних таблицах дополнительных столбцов, отсутствующих в родительской таблице;

– механизм наследования таблиц поддерживает множественное наследование;

– декларативное секционирование поддерживает только разбиение по спискам, по диапазонам и по хешу, тогда как с наследованием таблиц данные можно разделять по любому критерию, выбранному пользователем.

При декларативном секционировании обработка записей решается СУБД. При секционировании с использованием наследования для добавления записей в нужные секции необходимо вручную создавать триггеры, которые в зависимости от прописанного в коде триггера условия будут добавлять запись в нужную таблицу-секцию.

7.2.2.1 Декларативное секционирование по интервалам

Рассмотрим пример использования декларативного секционирования по интервалам в PostgreSQL. В отличие от MySQL, здесь необходимо отдельно создавать основную таблицу и таблицы-секции.

На первом шаге создадим таблицу `userslogs` как секционированную таблицу с предложением `PARTITION BY`, указав метод разбиения (в примере – `RANGE`) и список столбцов, которые будут образовывать ключ разбиения.

```
CREATE TABLE userslogs (
    username VARCHAR(20) NOT NULL,
    logdata TEXT NOT NULL,
    created DATE NOT NULL
)
PARTITION BY RANGE (created);
```

На втором шаге необходимо создать секции таблицы. В определении каждой секции должны задаваться границы, соответствующие методу и ключу разбиения родительской таблицы. Указание границ, при котором множество значений новой секции пересекается со множеством значений в одной или нескольких существующих секциях, будет ошибочным. Верхние границы не включаются в диапазон значений.

```
CREATE TABLE userslogs_y2013 PARTITION OF userslogs
   FOR VALUES FROM (MINVALUE) TO ('2014-01-01');
CREATE TABLE userslogs_y2014 PARTITION OF userslogs
   FOR VALUES FROM ('2014-01-01') TO ('2015-01-
01');
CREATE TABLE userslogs_y2015 PARTITION OF userslogs
   FOR VALUES FROM ('2015-01-01') TO ('2016-01-
01');
CREATE TABLE userslogs_y2016 PARTITION OF userslogs
   FOR VALUES FROM ('2016-01-01') TO (MAXVALUE);
```

Секции, создаваемые таким образом, во всех отношениях являются обычными таблицами PostgreSQL. В частности, для каждой секции можно независимо задать табличное пространство и параметры хранения.

Если необходимо реализовать вложенное секционирование, можно дополнительно указать предложение `PARTITION BY` в командах, создающих отдельные секции.

После создания секций можно создать индекс по ключевому столбцу (или столбцам), а также любые другие индексы, которые могут понадобиться. При этом автоматически будет создан соответствующий индекс в каждой секции и все секции, которые будут создаваться или присоединять позднее, тоже будут содержать такой индекс. Индексы или ограничения уникальности, созданные в секционированной таблице, являются «виртуальными», как и сама секционированная таблица: фактически данные находятся в дочерних индексах отдельных таблиц-секций.

Наконец, следует убедиться, что параметр конфигурации `enable_partition_pruning` не выключен в `postgresql.conf`. Иначе запросы не будут оптимизироваться должным образом.

Как и MySQL, PostgreSQL поддерживает команды для изменения и удаления секций, а также оптимизацию отсечение секций для выборки данных только из подходящих секций.

7.3 Сегментирование

Сегментирование (шардинг) – подход, предполагающий разделение баз данных, отдельных её объектов или индексов поисковых систем на независимые сегменты, каждый из которых управляется отдельным экземпляром сервера базы данных, размещаемым, как правило, на отдельном вычислительном узле. Таким образом,

если при секционировании разными сегментами таблицы управлял один сервер, то в сегментировании сегменты данных управляются разными серверами.

Секционирование в основном используется в реляционных СУБД, сегментирование – в NoSQL решениях.

Размещая подмножество данных на отдельных серверах, становится возможным хранить больше данных и обрабатывать бóльшую нагрузку, без необходимости наличия больших или более мощных серверов – достаточно просто большего количества менее мощных компьютеров. Сегментирование можно использовать и для других целей, включая повышение отказоустойчивости и создание георасределенных систем. В первом случае сегментирование позволяет изолировать отказы отдельных хостов или наборов реплик. Без сегментирования потеря отдельного хоста приводит к потере доступа ко всему набору данных, которые он содержит. В случае сегментирования отказ, например, одного сегмента из пяти оставляет доступными 80% данных коллекции. Распределение сегментов кластера по различным регионам в свою очередь может помочь:

– улучшить доступность для региональных пользователей;
– соблюсти местные законы, например, требование о хранении данных на территории определенной страны или региона.

Вручную сегментирование можно осуществить практически с помощью любой СУБД. При таком подходе приложение поддерживает соединения с несколькими различными серверами баз данных, каждый из которых является полностью независимым. Приложение управляет хранением различных данных на разных серверах и выполняет запросы к соответствующему серверу, чтобы получить данные. Такой вариант может хорошо работать, но его становится трудно поддерживать при добавлении или удалении узлов из кластера или при изменении распределения данных по сегментам. И в

целом при таком подходе все усилия по работе с данными, расположенными на разных серверах, ложатся на плечи разработчиков приложений.

Существуют несколько СУБД, которые поддерживают автоматическое сегментирование – в основном это NoSQL решения MongoDB, Apache Cassandra, Elasticsearch и другие.

Автоматическое сегментирование в реляционных СУБД поддерживается хуже и в основном требует использование либо дополнительного ПО, либо обладает существенными недостатками.

В частность, большинство гибридных решений для PostgreSQL основаны на технологии FDW (foreign data wrapper). FDW – это функциональность сервера PostgreSQL, которая позволяет ему рассматривать данные на удаленном сервере как часть большого набора и интерпретировать этот разделенный на несколько серверов набор данных как единое целое. У этой технологии есть свои проблемы, но самая большая из них – доступ к данным через FDW чрезвычайно медленный.

7.3.1 Сегментирование в MongoDB

Сегментирование позволяет создавать кластер из множества серверов и распределять коллекцию по ним, поместив подмножество данных в каждый сегмент.

Одна из целей сегментирования – сделать так, чтобы для клиентского приложения кластер из нескольких сегментов выглядел как единый сервер. Чтобы скрыть эти детали от приложения, перед сегментами запускается один или несколько процессов маршрутизации под названием `mongos`. Процессы `mongos` хранят так называемое «оглавление таблиц», которое сообщает им, в каком сегменте какие данные содержатся. Приложения могут подключаться к этому маршрутизатору и отправлять запросы в обычном режиме, как показано на рисунке 18.

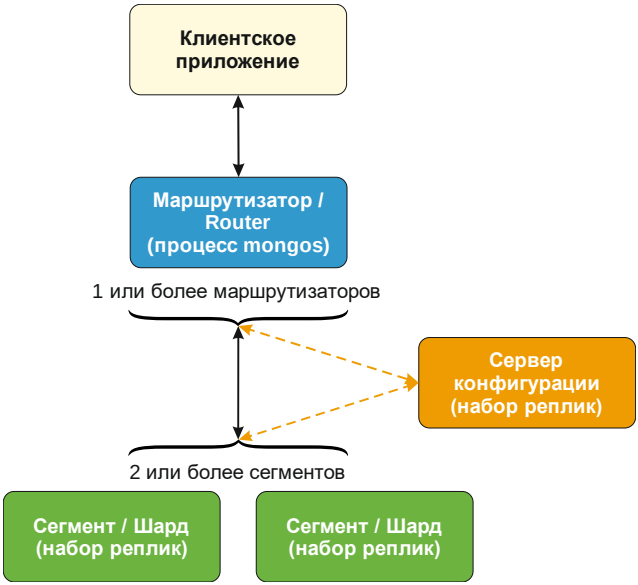

Рисунок 18 – Схема сегментирования

«Маршрутизатор», зная, какие данные на каком сегменте находятся, может пересылать запросы соответствующему сегменту или сегментам. При получении ответов маршрутизатор собирает их и, если необходимо, объединяет и отправляет обратно в приложение.

Таким образом, с точки зрения приложения, оно подключено к автономному серверу `mongod`, а вся реализация распределенного хранения данных от него скрыта.

При этом каждый сегмент может быть также реплицирован по нескольким серверам, т.е. каждый сегмент может храниться не на одном автономном сервере, а в наборе реплик.

Таким образом, сегментированный кластер MongoDB состоит из следующих компонентов:

– непосредственно сегмент, который содержит подмножество данных коллекции. Каждый сегмент можно развернуть как набор реплик;

– процесс `mongos`, который действует как маршрутизатор запросов, предоставляя интерфейс для взаимодействия клиентского приложения с сегментированным кластером;

– серверы конфигурации, которые хранят метаданные и параметры конфигурации для кластера.

Горизонтальное масштабирование подразумевает распределение набора данных и нагрузки по нескольким узлам СУБД. Для этого распределения данных по сегментам используется так называемый **ключ сегментирования**. Сущности, связанные одинаковым значением ключа сегментирования, группируются в набор данных по заданному ключу. Этот набор данных хранится в пределах одного физического сегмента, что существенно облегчает обработку данных. Таким образом, если известен ключ сегментирования некоторого объекта, то всегда можно ответить на вопросы: «где следует сохранить данные?» и «где найти запрошенные данные?».

MongoDB использует ключ сегментирования для распределения документов коллекции по сегментам. Ключ состоит одного или нескольких полей документа:

– начиная с версии MongoDB 4.4, в документах в сегментированных коллекциях могут отсутствовать поля ключа сегментирования. Отсутствующие поля ключа обрабатываются как имеющие значения `NULL` при распределении документов по сегментам, но не при маршрутизации запросов;

– ключ выбирается при сегментировании коллекции;

– начиная с версии MongoDB 5.0, можно выполнить повторное сегментирование коллекции, изменив ключ сегментирования;

– значение ключа сегментирования документа определяет его распределение по сегментам;

– начиная с версии MongoDB 4.2, можно обновить значение ключа сегментирования документа, если поле ключа сегментирования не является неизменяемым полем `_id`.

Чтобы сегментировать заполненную коллекцию, она должна иметь индекс, начинающийся с ключа сегментирования. При сегментировании пустой коллекции MongoDB создает поддерживающий индекс, если в коллекции еще нет соответствующего индекса для указанного ключа сегмента.

Выбор ключа сегментирования влияет на производительность, эффективность и масштабируемость сегментированного кластера. Кластер с наилучшим аппаратным обеспечением и инфраструктурой может оказаться узким местом из-за выбранного ключа сегментирования.

MongoDB поддерживает две **стратегии сегментирования**:

– ranged – разделение данных на непрерывные диапазоны;
– hashed – разделение данных на основе хеш-функции.

Еще одно понятие, используемое в MongoDB для сегментирования данных – **чанки** (фрагменты данных, chunks). MongoDB разделяет сегментированные данные на чанки, являющиеся единицами, которые MongoDB использует для перемещения данных.

Каждый процесс `mongo` всегда должен знать, где найти документ, учитывая его ключ сегментирования. Теоретически MongoDB может отслеживать, в каком сегменте находится каждый документ, но в случае с коллекциями с миллионами или миллиардами документов это становится затратно. Поэтому MongoDB группирует документы в чанки по некоторому диапазону ключа сегментирования. Чанки всегда хранятся в одном сегменте, что позволяет MongoDB хранить небольшую таблицу связей чанк-сегмент.

Чтобы добиться равномерного распределения фрагментов по всем сегментам в кластере, в фоновом режиме запускается балансировщик для переноса чанков между сегментами.

7.3.1.1 Стратегии сегментирования

Сегментирование с использованием хеш-функции основывается на вычислении хеша значения поля, входящего в ключ сегментирования. Каждому чанку назначается диапазон хеш-значений ключа сегментирования.

MongoDB автоматически вычисляет хеши при анализе запросов с использованием хешированных индексов. Приложениям не нужно вычислять хеши.

Хотя диапазон ключей сегментирования может быть «близким», их хешированные значения вряд ли будут находиться в одном и том же чанке. Сегментирование на основе хешированных значений способствует более равномерному распределению данных, особенно в наборах данных, где ключ сегментирования изменяется монотонно. Однако хешированное распределение означает, что запросы по диапазону значений с меньшей вероятностью будут нацелены на один сегмент, что приводит к большему количеству широковещательных операций в масштабе кластера (рисунок 19).

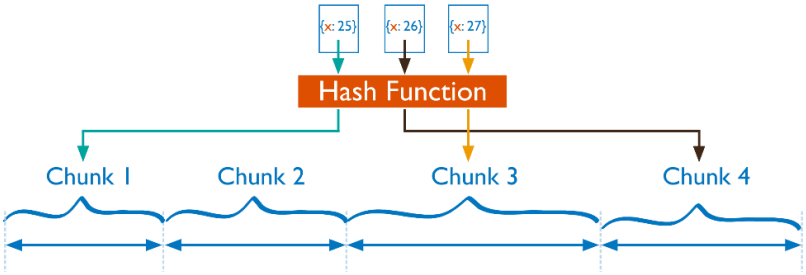

Рисунок 19 – Схема стратегии сегментирования на основе хеш-функции

Сегментирование по диапазонам включает разделение данных на диапазоны на основе значений ключа сегментирования, а не на основе хеша от этих значений. Каждому чанку назначается диапазон на основе значений ключа сегментирования (рисунок 20).

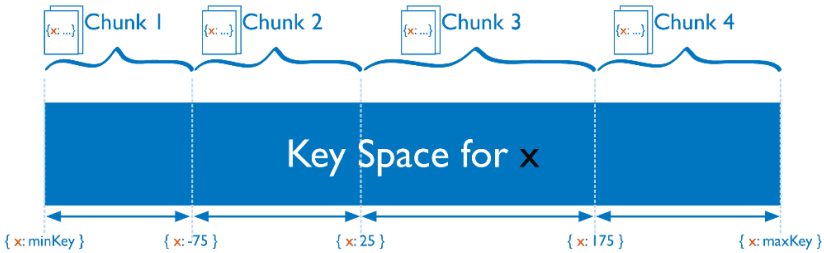

Рисунок 20 – Схема стратегии сегментирования по диапазонам

Диапазон ключей сегментирования, значения которых «близки», с большей вероятностью будет находиться в одном и том же чанке. Это позволяет выполнять операции, содержащие условия поиска по ключу сегментирования, к конкретному сегменту или сегментам, которые содержат необходимые данные.

Эффективность сегментирования по диапазонам зависит от выбранного ключа сегмента. Плохо продуманные ключи сегментов могут привести к неравномерному распределению данных, что может свести на нет некоторые преимущества сегментирования или вызвать снижение производительности.

7.3.1.2 Архитектура кластера в рабочей среде

В рабочей среде разработчики MongoDB предлагают использовать архитектуру кластера, обеспечивающего избыточность данных и высокую доступность системы. Предлагается развертывание серверов конфигурации в виде набора из трех реплик, развертывание сегментов также как набора из трех реплик, и развертывание одного или нескольких маршрутизаторов `mongos` (рисунок 21).

Для тестовой конфигурации можно использовать один-два сегмента и один процесс `mongos` для маршрутизации.

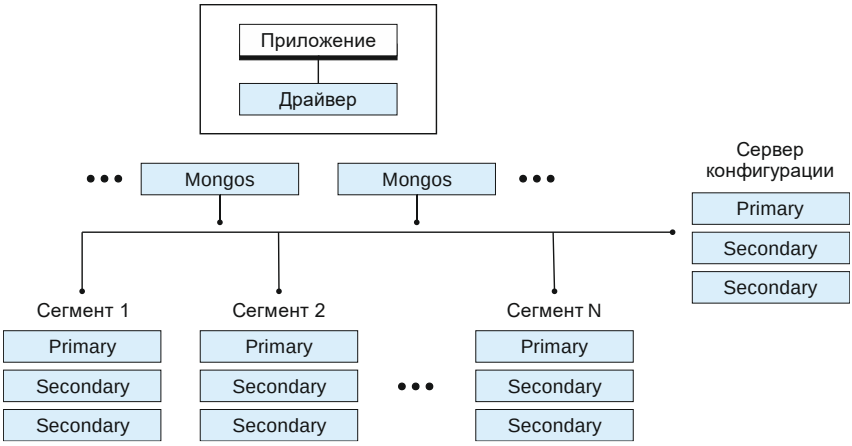

Рисунок 21 – Архитектура кластера в рабочей среде

7.3.1.3 Настройка сегментирования

Произведем настройку тестовой конфигурации кластера, состоящего из процесса-маршрутизатора mongos, конфигурационного сервера, запущенного как набор из трех реплик, и двух сегментов, также запущенных как набор из трех реплик для каждого сегмента. В этом примере не рассматриваются шаги, необходимые для настройки механизма аутентификации или ролевого контроля доступа. Но в рабочем окружении необходимо использовать как минимум аутентификацию на основе сертификатов x.509.

Шаг 1. Создание сервера конфигурации. В примере для инициализации серверов будет использоваться файл конфигурации (вместо передачи параметров через командную строку). Файл конфигурации для каждой реплики содержит путь для хранения данных сервера, порт и `ip` либо имя сервера, роль в сегментированном кластере – для конфигурационного сервера указывается роль `configsvr`, и имя набора реплик:

```
storage:
     dbPath: E:\mongo\cfg1\
```

```
net:
     port: 27018
     bindIp: 127.0.0.1  #<имя хоста | ip-адрес(а)>
sharding:
     clusterRole: configsvr
replication:
     replSetName:  cfg_replica_set   #<replica set
name>
```

Для запуска mongo с указанным файлом конфигурации используется следующая команда:

```
$> mongod --config <path-to-config-file>
```

В данном примере будет использоваться конфигурационный сервер, содержащий три реплики, поэтому было запущена три экземпляра mongo с тремя файлами конфигурации, различающими путями к данным и портами:

```
$> mongod --config E:\mongo\cfg1.cfg
$> mongod --config E:\mongo\cfg2.cfg
$> mongod --config E:\mongo\cfg3.cfg
```

Шаг 2. Настройка набора реплик для конфигурационного сервера. Для настройки необходимо подключиться к одному из экземпляров сервера, входящего в состав группы сервера конфигурации, с использованием утилиты mongosh:

```
$> mongosh --host <hostname> --port <port>
```

В примере выполняется подключение к первому серверу, запущенному на порту 27018:

```
$> mongosh --host 127.0.0.1 --port 27018
```

Далее выполняется инициализация набора реплик с использованием метода rs.initiate. В качестве параметра метода необходимо передать документ, содержащий имя набора реплик, задать

значение configsvr: true для набора реплик сервера конфигурации, в поле members задать массив документов с описанием каждого сервера-реплики:

```
rs.initiate({
     _id: "cfg_replica_set",
     configsvr: true,
     members: [
          { _id: 0, host: "localhost:27018" },
          { _id: 1, host: "localhost:27019" },
          { _id: 2, host: "localhost:27020" }
     ]
})
```

Шаг 3. Создание каталогов для сервера сегмента и инициализация серверов. На этом шаге создаются члены набора реплик для каждого сегмента кластера. В примере создаются два сегмента, каждый из которых реплицирован на три сервера-реплики. Файл конфигурации для реплик – членов сегмента кластера, отличается ролью сервера – в этом случае указывается тип shardsvr:

```
storage:
     dbPath: E:\mongo\shard1_rs1\
net:
     port: 27021
     bindIp: 127.0.0.1  #<имя хоста | ip-адрес(а)>
sharding:
     clusterRole: shardsvr
replication:
     replSetName:  shard1_replica_set    #<replica
name>
```

Запуск серверов выполняется с использованием следующей команды:

```
$> mongod --config <path-to-config-file>
```

Всего было запущено шесть серверов:

```
$> mongod --config E:\mongo\shard1_rs1.cfg
$> mongod --config E:\mongo\shard1_rs2.cfg
...
$> mongod --config E:\mongo\shard2_rs3.cfg
```

Три реплики для первого сегмента объединена в набор реплик с именем `shard1_replica_set`, для второго сегмента – в набор реплик `shard2_replica_set`.

Шаг 4. Настройка набора реплик для сегментов. Для настройки репликации необходимо подключиться к одному из серверов, входящих в состав реплики, и выполнить инициализацию набора реплик. Например, для первого сегмента будут выполнены следующие команды:

```
$> mongosh --host 127.0.0.1 --port 27021
rs.initiate({
    _id: "shard1_replica_set",
    members: [
        { _id: 0, host: "localhost:27021" },
        { _id: 1, host: "localhost:27022" },
        { _id: 2, host: "localhost:27023" }
    ]
})
```

Те же команды повторяются для инициализации второго сегмента кластера – выполняется подключение к одному из серверов, входящих в набор реплик второго сегмента, и выполняется инициализация набора реплик.

Шаг 6. Создание процесса-маршрутизатора `mongos`. Заключительным шагом необходимо запустить процесс-маршрутизатор `mongos`. Для его запуска используется файл конфигурации следующего вида.

```
net:
     port: 27027
     bindIp: 127.0.0.1 # <имя хоста | ip-адрес(а)>
sharding:
configDB:                                    cfg_rep-
lica_set/127.0.0.1:27018,127.0.0.1:27019
```

Здесь необходимо указать имя набора реплик сервера конфигурации и по крайней мере одного из члена набора реплик сервера конфигурации.

После этого осуществляется подключение к серверу с использованием утилиты `mongosh` и добавляются сегменты в кластер с использованием функции `addShard` с указанием имени реплики и серверов, входящих в ее состав:

```
$> mongod --config E:\mongo\mongos.cfg
$> mongosh --host 127.0.0.1 --port 27027
sh.addShard("shard1_replica_set/localhost:27021,
            localhost:27022,localhost:27023")
sh.addShard("shard2_replica_set/localhost:27024,
            localhost:27025,localhost:27026")
```

После этого выполняется сегментирование коллекции. Если коллекции не существует – она будет создана. Для сегментирования коллекции используется функция `sh.shardCollection`, в параметрах которой указывается база данных, имя коллекции, ключ сегментирования и стратегия сегментирования (с использованием хеш-функции или диапазона значений):

```
sh.shardCollection("<database>.<collection>",
                   { <shard key field> : type } )
```

В примере создается сегментированная коллекция `listings` из БД `airbnb` с использование хеш-функций в качестве стратегии сегментирования:

```
sh.shardCollection("airbnb.listings",
{_id:"hashed"})
```

Для проверки статуса можно выполнить команду `sh.status`, которая отобразит информацию о настройках конфигурации, сегментах, активных экземплярах процесса-маршрутизатора, состоянии балансировщика нагрузки, а также сегментированных базах данных.

8 АУДИТ

8.1 Задачи аудита

Угроза безопасности данных может исходить как от неавторизованных, так и от авторизованных пользователей базы данных. В первом случае – это злоумышленники извне, пытающиеся получить доступ к данным, во втором – несанкционированные действия работающих или бывших сотрудников. В этой связи отслеживание информации о том, кто, когда и каким образом работал с сервером базы данных очень важно, т.к. позволяет обнаружить и предотвратить инцидент, либо расследовать его. Одним из способов сбора и накопления такой информации является аудит баз данных.

Аудит событий безопасности БД представляет собой процесс получения и анализа данных о происходящих в системе событиях и степени их соответствия требованиям к защите данных.

Аудит может отслеживать события, происходящие на различных уровнях. События записываются в журналы событий или файлы аудита.

Когда аудит активен, каждая контролируемая операция с базой данных порождает аудиторский след с информацией о том, какой объект базы данных был задействован, а также кто и когда выполнял эту операцию. Аудит должен давать ответы на такие вопросы как: "кто получал доступ к данным на чтение или запись?", "когда данные были изменены?", а также "каким было значение данных до изменения?".

Комплексный аудиторский след, накопленный в течение определенного отрезка времени, позволяет проводить углубленный анализ данных аудита и строить модели типичного поведения, которые могут быть использованы для автоматического выявления нетипич-

ного поведения, основываясь на статистическом анализе или на методах машинного обучения. Кроме обнаружения вторжений и выявления несанкционированных действий наличие аудита, по понятным причинам, может являться сдерживающим фактором для потенциальных внутренних злоумышленников. Также, в некоторых ситуациях, когда речь идет о конфиденциальных данных, может потребоваться сформировать отчет о пользователях, имевших доступ к определенным данным – для решения этой задачи также подходит аудит базы данных.

8.2 Журнал аудита

В идеале сбор информации о состоянии системы безопасности БД должен осуществляться непрерывно, для этого очень многие СУБД автоматически ведут журнал аудита (контрольный журнал).

Журнал аудита обеспечивает индивидуальную ответственность пользователя за все его действия, для этого все события аудита однозначно связываются с учетной записью пользователя в СУБД. В журнале содержится:

– описание стандартного набора событий (авторизации пользователя, доступа к тем или иным данным и операций с ними; создания, модификации и удаления объектов БД; выполнение нештатных SQL-команд и т.д.);

– настраиваемый перечень атрибутов в отдельной записи журнала аудита (даты и времени события, идентификатор пользователя, имя и сетевой адрес компьютера, описание события, связанные с событием объекты, признак успешного или неудачного завершения события).

Для удобства анализа данных журнал обязан позволять фильтровать и сортировать свои записи. Более того, журнал аудита сам по себе должен быть защищен от несанкционированного доступа.

8.3 Проблемы аудита

Обычно средства аудита позволяет проводить аудит на разных уровнях, на уровне сервера, на уровне БД, на уровне объектов БД, на уровне пользователей и т.д.

Одной из проблем является снижение производительности: с одной стороны, аудиторский след должен быть достаточно подробным, с другой стороны, сбор такого большого количества информации может отрицательно влиять на производительность системы.

Другой проблемой является объем данных: чем более подробный аудиторский след требуется, тем больший объем памяти необходим для его хранения.

В связи с этим, желательно иметь возможности гибкой настройки параметров аудита, чтобы минимизировать проблемы с производительностью и хранением. Например, на аудит можно поставить только определенную группы таблиц, в которой хранятся конфиденциальные данные.

8.4 Методы аудита

Существует несколько подходов, которые в той или иной степени позволяют решать задачи аудита:

- трассировка;
- анализ журналов транзакций;
- использование темпоральных данных;
- аудиторские следы в данных;
- мониторинг сетевого трафика сервера БД;
- мониторинг сервера БД.

8.4.1 Трассировка

Этот подход основан на технологии трассировки, встроенной непосредственно в движок СУБД. Суть этой технологии заключается в том, что при активной функции аудита СУБД протоколирует

все события, связанные с контролируемыми объектами. Хотя каждая СУБД предоставляет различные возможности настройки аудита, к числу протоколируемых событий, как правило, относятся:

– аутентификация пользователей (как успешная, так и неудачная);
– предоставление прав доступа к БД;
– создание, изменение и удаление объектов БД;
– вставка, редактирование и удаление данных;
– нарушения ссылочной целостности при модификации данных;
– исполнение хранимых процедур;
– изменение настроек сервера и прикладного ПО;
– любые отказы в обслуживании пользователя;
– попытки осуществить потенциально опасные операции без наличия соответствующих прав;
– различного рода исключительные ситуации и ошибки в работе ПО.

Недостатком этого подхода является высокий риск деградации производительности в нагруженных системах. Также, поскольку данные трассировки, как правило, сохраняются в проприетарном формате, то средства анализа этих данных ограничены предоставляемыми разработчиками СУБД, что может не всегда удовлетворять поставленным задачам.

8.4.2 Анализ журнала транзакций

Каждая современная СУБД использует журналы транзакций для регистрации всех изменений в БД для целей восстановления. Этот подход основан на использовании программных средств интерпретации и анализа журналов транзакций, чтобы определить, какие данные были изменены, когда и какими пользователями. Основным недостатком этого метода является то, что в журналах

транзакции фиксируются только операции модификации данных, а операции чтения или, например, попытки аутентификации на сервере – не фиксируются. Также существуют способы отключения ведения журналов транзакций, либо варианты ведения с неполным протоколированием. В этом случае информация об изменениях данных будет потеряна полностью или частично. Попутно возникает необходимость резервирования журналов для возможности анализа до того момента, как они будут усечены средствами СУБД. Достоинством данного подхода можно считать отсутствие влияния на производительность системы, так как сканирование журналов хотя и достаточно затратный процесс, но выполняется отдельным программным обеспечением и, возможно, на отдельном сервере.

8.4.3 Использование темпоральных данных

Многие современные СУБД поддерживают так называемые темпоральные таблицы. В темпоральных таблицах все записи имеют две временных метки, определяющих начало и конец периода актуальности данных. Операции модификации данных для этих таблиц происходят иначе, чем для обычных. Например, при выполнении операции `UPDATE` происходит не замена значения атрибутов в существующей записи, а корректировка конца периода текущей записи и добавление новой записи с новыми значениями атрибутов и новым периодом актуальности. Фактически таблица представляет собой исторический набор данных и на любую дату может быть получен актуальный срез данных. В некотором смысле такую возможность можно использовать для отслеживания изменения данных.

Недостатком этого метода является то, что отслеживаются только изменения данных и их время, но не то, кем они были сделаны. Также этот метод не может быть использован для отслеживания операций чтения и других потенциально важных событий.

Достоинством данного подхода можно считать незначительное влияния на производительность системы, так как он не требует дополнительного протоколирования, а анализ изменений в данном случае хотя и производится средствами СУБД, но достаточно легковесен.

8.4.4 Аудиторские следы в данных

Этот подход заключается в добавлении необходимых «столбцов аудита» в таблицы, например, таких как `LAST_MODIFIED_DATE` и `LAST_MODIFIED_USER`, которые заполняются на уровне клиентских приложений или на уровне триггеров БД. Но это достаточно проблемное решение, так как у администратора, например, есть возможность отключить триггеры или изменить данные напрямую, а не через клиентское приложение, либо изменить данные в «столбцах аудита».

Как и остальные методы отслеживания изменений, этот метод не может быть использован для отслеживания операций чтения и других потенциально важных событий. Кроме того, хорошей практикой является, когда аудиторский след хранится отдельно от данных, а в данном случае при удалении строки данных будет потерян и аудиторский след. Влияние данного метода на производительность системы незначительно, но приводит к увеличению объема хранимых данных.

8.4.5 Мониторинг сетевого трафика сервера БД

Этот подход основан на прослушивании сетевого трафика СУБД. Все SQL-инструкции, которые передаются по сети от клиентов к серверу и относятся к контролируемым объектам, протоколируются. Недостатком данного подхода является то, что отслеживаться могут только те инструкции, которые переданы по сети, т.е. все события, выполненные непосредственно на сервере, отследить

невозможно. Кроме того, существуют такие реализации клиент-серверного взаимодействия, при которых большая часть операций централизована, и важные транзакции не осуществляются по сети. Мониторинг сети не потребляет ресурсов СУБД и, соответственно, не оказывает влияния на производительность системы. Также возможна достаточно гибкая настройка аудита на уровне определения интересующих инструкций.

8.4.6 Мониторинг сервера БД

Этот подход схож с предыдущим, но перехват и анализ SQL-инструкций выполняется не на уровне сети, а непосредственно на уровне сервера. Этот подход позволяет отслеживать все SQL-инструкции, исполняемые ядром СУБД. Потенциальная опасность такого подхода заключается в том, что в этом случае процесс аудита взаимодействует непосредственно с ядром СУБД, и какие-либо ошибки могут приводить к критическим сбоям в работе СУБД. Такой подход также не оказывает влияния на производительность системы и допускает гибкую настройку аудита на уровне определения интересующих инструкций.

8.5 Общие рекомендации

Стратегия организации аудита сервера баз данных зависит от хранимых данных, происходящих с ними процессов, а также от организационных нормативов и требований в данной предметной области. В любом случае, аудиторский след должен охватывать все конфиденциальные данные, а также ряд событий, которые необходимо отслеживать:

– попытки неудачной аутентификации. Данные события сигнализируют о возможных попытках подбора параметров аутентификации с целью получения доступа к серверу БД;

– изменения в системе безопасности. Создание и удаление пользователей и ролей, изменение членства в ролях, а также отзыв и предоставление привилегий – всё это события, которым стоит уделять внимание. Любая активность в системе безопасности должна тщательно проверяться, так как может сигнализировать, например, о планирующейся или успешной атаке;

– выполнения резервного копирования. Создание резервной копии, пожалуй, самый простой способ кражи данных. Аудит таких событий позволит отследить несанкционированное выполнение резервных копий и предотвратить или расследовать утечку данных;

– действия привилегированных пользователей. Привилегированные пользователи (администраторы сервера, администраторы баз данных) в силу специфики своей работы, как правило, имеют полный доступ ко всем корпоративным данным. Аудит их действий позволит быть уверенным, в том, что они не получают доступ к производственным данным и не вносят изменения в них без производственной необходимости;

– события подсистемы аудита. Отслеживание событий, связанных с администрированием подсистемы аудита, позволит быть уверенным в том, что эта подсистема функционирует должным образом, что аудит не был деактивирован и набор протоколируемых событий не был изменен.

8.6 Возможности аудита в MySQL

MySQL и его ответвления Percona Server for MySQL и MariaDB, т.е. дистрибутивы с открытым исходным кодом, являющиеся полностью совместимой заменой Oracle MySQL, предоставляют различные способы осуществления аудита.

8.6.1 Плагины аудита

Плагин аудита MySQL Enterprise Audit Plugin в настоящий момент предлагается только в составе коммерческой версии MySQL Enterprise Edition. Он предоставляет обширные функции аудита: механизмы фильтрации событий, ведение журнала событий в формате JSON, компрессию и шифрование данных, динамическое изменение конфигурации без перезагрузки БД, вывод информации в формате аудита в соответствии с различными стандартами.

Для версии MySQL от Percona есть бесплатный плагин аудита Percona Audit Log Plugin который имеет открытый исходный код. Плагин позволяет вести журнал событий в различных форматах, а также отправлять события на syslog-сервер. Syslog – это стандартизированный сетевой протокол отправки и регистрации сообщений о происходящих в системе событиях. В плагине можно гибко настроить ведение аудита для определенных групп команд и для определенных пользователей или, наоборот, исключить из аудита определенные группы команд и пользователей. Также плагин поддерживает ротацию журналов событий. Плагин поставляется в составе Percona Server for MySQL и не подразумевает использование с другими версиями MySQL.

Для версии MySQL от компании MariaDB существует бесплатный плагин аудита MariaDB Audit Plugin, который также имеет открытый исходный код. Плагин позволяет вести журнал событий в текстовом формате, также в плагине есть возможность ротации журналов событий и отправки событий на syslog-сервер. В плагине поддерживаются фильтры по типам команд, фильтры по пользователям (можно указать для каких пользователей вести аудит или наоборот, каких пользователей исключить). Плагин поставляется в составе MariaDB, но допускает использование и с Percona Server for MySQL и Oracle MySQL (версии 5.7).

8.6.2 Анализ серверных журналов

Как частичную реализацию задачи аудита можно рассматривать чтение и анализ серверных журналов MySQL, таких журналов несколько, их особенности рассмотрены ниже. К глобальным недостаткам этого метода можно отнести то, что для интерпретации и анализа данных необходимо использовать стороннее программное обеспечение:

– журнал ошибок (Error Log). Содержит информацию о том, когда был запущен или остановлен экземпляр сервера MySQL, а также о любых критических ошибках в процессе его работы. В задачах аудита этот журнал можно использовать только для обнаружения неудачных попыток аутентификации;

– журнал медленных запросов (Slow Query Log). Содержит данные о SQL-запросах, на выполнение которых ушло времени больше, чем определено переменной `long_query_time`. В этот журнал записываются все медленные запросы, включая `SELECT`, а также информация о пользователе и времени исполнения. Гипотетически в этот журнал можно отправить вообще все запросы, установив `long_query_time = 0`, но производительность сервера в этом случае сильно снизится, а журнал будет весьма объемным;

– двоичный журнал (Binary Log). Содержит только те запросы, которые изменяли данные, поэтому не может быть использован для аудита запросов выборки данных. Тем не менее, этот журнал может быть использован для обнаружения времени модификации данных. Ведение этого журнала не создаст дополнительной нагрузки на сервер, поскольку он, как правило, всегда ведется для задач резервного копирования или репликации;

– журнал запросов (General Query Log). Содержит информацию о подключении (включая неудачном) и отключении клиентов, а также все SQL-запросы (включая синтаксически правильные, но не выполненные вследствие ошибок), полученные от клиентов. Этот журнал предоставляет наиболее полные данные с точки зрения аудита, но его ведение может оказывать значительное влияние на производительность сервера.

8.6.3. Анализ производительности

Для отслеживания работы сервера MySQL и настройки его производительности одним из самых полезных инструментов является журнал медленных запросов – запросов, время выполнения которых превышает заданное количество секунд. Частое выполнение таких запросов может серьезно снизить производительность сервера БД.

В MySQL Server 8.0 логирование медленных запросов включено по умолчанию. Для настройки работы журнала необходимо отредактировать файл конфигурации сервера: переменная `slow-query-log` определяет состояние журнала медленных запросов (0 – выключен, 1– включен); переменная `long_query_time` определяет минимальное время выполнения запроса в секундах, при котором данный запрос фиксируется в журнале (по умолчанию `long_query_time` равно 10 секундам).

Если журнал медленных запросов включен и в качестве места назначения вывода выбран файл, каждому оператору, записываемому в журнал, предшествует строка, начинающаяся с символа # и содержащая следующие поля (со всеми полями в одной строке):

- `Query_time` – время выполнения запроса в секундах;
- `Lock_time` – время получения блокировки в секундах;
- `Rows_sent` – количество строк, отправленных клиенту;

– `Rows_examined` – количество строк, просмотренных на сервере для проверки выполнения условия запроса.

Анализ данных, представленных в файле журнала медленных запросов, можно выполнять как вручную, так и с использованием входящей в состав MySQL Server утилиты `mysqldumpslow`.

8.7 Возможности аудита в PostgreSQL

8.7.1 Протоколирование

В PostgreSQL представлено стандартное средство протоколирования работы сервера. Настройка протоколирования событий управляется множеством параметров, которые позволяют определить метод и формат протоколирования, расположение файлов журналов событий, параметры ротации журналов, типы событий сервера и SQL-выражений, которые необходимо протоколировать и т.д. Настройка протоколирования осуществляется через конфигурационный файл сервера.

Параметры настройки протоколирования в PostgreSQL, определяющие, куда протоколировать события аудита, включают:

– параметр `log_destination`. PostgreSQL поддерживает несколько методов протоколирования сообщений сервера: stderr, csvlog, jsonlog и syslog. Опция jsonlog поддерживается, начиная с 15й версии PostgreSQL. В Windows также поддерживается eventlog. В качестве значения `log_destination` указывается один или несколько методов протоколирования, разделённых запятыми. По умолчанию используется stderr. Параметр можно задать только в конфигурационных файлах или в командной строке при запуске сервера;

– параметр `logging_collector` включает сборщик сообщений. Это фоновый процесс, который собирает отправленные в stderr сообщения и перенаправляет их в журнальные файлы;

– параметр log_directory определяет каталог, в котором создаются журнальные файлы (при включённом параметре logging_collector);

– параметр log_filename задает имена журнальных файлов (при включённом параметре logging_collector).

Также можно задать максимальное время жизни отдельного журнального файла, по истечении которого создаётся новый файл, максимальный размер отдельного журнального файла и другие опции.

Следующий блок параметров определяет то, какие сообщения записывать в журнал сервера:

– параметр log_min_messages управляет минимальным уровнем сообщений, записываемых в журнал сервера. Допустимые значения от DEBUG до PANIC. Чем дальше в этом списке уровень сообщения, тем меньше сообщений будет записано в журнал сервера. По умолчанию используется WARNING;

– параметр log_min_error_statement управляет тем, какие SQL-операторы, завершившиеся ошибкой, записываются в журнал сервера;

– параметр log_min_duration_statement записывает в журнал продолжительность выполнения всех команд, время работы которых не меньше указанного. С помощью этого параметра можно выявить неоптимизированные запросы в приложениях.

Кроме того, можно управлять тем, что именно будет записываться в журнал аудита. Типы протоколируемых SQL-выражений определяются параметром log_statement который может принимать следующие значения:

– none – это режим по умолчанию, при котором SQL-команды не протоколируются;

– `ddl` – протоколируются все команды определения данных;

– `mod` – протоколируются все команды `ddl`, а также команды модификации данных;

– `all` – протоколируются все SQL-команды, включая оператор `SELECT`.

Выбирая, что будет записываться в протокол, важно учитывать возможные риски безопасности. Журнал сервера может содержать конфиденциальную информацию и должен быть защищён, где бы он ни хранился и куда бы ни передавался. Например, операторы `DDL` могут содержать незашифрованные пароли или другие данные аутентификации. В операторах, выводимых на уровне `ERROR`, может отображаться исходный код SQL для приложений, а также могут содержаться фрагменты строк данных. Поэтому следует позаботиться о том, чтобы к журналам сервера получали доступ только лица с соответствующими полномочиями.

Также в виде строки форматирования можно задать так называемый префикс записи журнала, который выводится в начале каждой строки. Используя различные спецификаторы вывода, в префикс можно включить название текущей БД, имя пользователя или приложения, время события и т.д.

Протоколирование всегда работает на уровне сервера, т.е. нельзя поставить на аудит определенную базу данных или объект базы данных, равно как нельзя поставит на аудит действия конкретных пользователей, можно лишь отобрать интересующие данные в журнале. Для просмотра и анализа журнала событий необходимо программное обеспечение сторонних производителей, в составе СУБД такие средства не предусмотрены. Избыточное протоколирование, необходимое для записи выражений `SELECT`, значительно увеличивает нагрузку на сервер и формирует очень объемные файлы журналов.

8.7.2 Расширение PGAudit

Для PostgreSQL существует бесплатное расширение PGAudit, которое обеспечивает детальный аудит событий сессий и/или объектов с помощью стандартной подсистемы протоколирования PostgreSQL, а также позволяет создавать журналы аудита, соответствующие правительственным и финансовым требованиям, а также стандартам ISO.

Аудит событий сессии подразумевает протоколирование всех SQL-выражений, выполненных ядром СУБД в рамках клиентской сессии. Настройка протоколируемых выражений выполняется с помощью перечисления классов команд (`READ`, `WRITE`, `ROLE`, `DDL` и т.д.) в конфигурационных файлах. Причем классы команд могут включаться/отключаться на уровне сервера, базы данных и пользователя.

Аудит событий объектов протоколирует выражения, которые затрагивают конкретную таблицу. Поддерживаются только команды `SELECT`, `INSERT`, `UPDATE` и `DELETE`, команда `TRUNCATE` не протоколируется. Аудит событий объектов основан на системе ролей. Можно определить несколько ролей аудита, что позволит нескольким группам отвечать за различные аспекты ведения журналов аудита.

8.8 Возможности аудита в MongoDB

MongoDB Enterprise включает возможность аудита экземпляров `mongod` и `mongos`. `Mongod` управляет отдельным экземпляром сервера, в то время как `mongos` предоставляет интерфейс для подключения к кластеру. Средства аудита позволяет администраторам и пользователям отслеживать активность системы со многими пользователями и приложениями.

Средство аудита может записывать события аудита в консоль, системный журнал, файл JSON или файл BSON. Чтобы включить аудит в MongoDB Enterprise, необходимо задать назначение вывода событий аудита с помощью параметра `--auditDestination`.

После включения система аудита может записывать следующие операции:

– операции со схемой (`DDL`-операции);

– операции с набором реплик и сегментированным кластером;

– события аутентификации и авторизации;

– операции модификации данных (требуется установка параметра `auditAuthorizationSuccess`).

С помощью системы аудита можно настроить фильтры для ограничения записываемых событий.

8.8.1 Гарантия аудита

Система аудита записывает каждое событие аудита (если не была настроена фильтрация) в буфер событий аудита в памяти. MongoDB периодически записывает этот буфер на диск. Для событий, полученных в рамках любого отдельного соединения, события имеют общий порядок: если одно событие записывается на диск, система гарантирует, что на диск записаны все предыдущие события для этого соединения.

Если запись о событии аудита соответствует операции, влияющей на долговременное состояние базы данных, например, изменение данных, событие аудита всегда записывается на диск перед записью в журнал транзакций для этой операции. То есть перед добавлением операции в журнал транзакций записывают все события аудита в рамках соединения, которое инициировало операцию, вплоть до записи об операции включительно.

Для обеспечения гарантии записей о событиях аудита требуется, чтобы СУБД работала с включенным ведением журнала транзакций.

Кроме того, если сервер не может выполнить запись в журнал аудита, сервер прекратит работу.

8.8.2 Настройка аудита

Для включения аудита в коммерческой версии необходимо либо указать параметры при запуске экземпляра сервера из командной строки, либо указать соответствующие настройки в конфигурационных файлах.

Для включения записи событий аудита в консоль или системный журнал, необходимо указать значения параметра `destination` секции `auditLog` конфигурационного файла как `console` или `syslog` соответственно:

```
storage:
     dbPath: data/db
auditLog:
     destination: console  # syslog
```

Для включения записи событий аудита в файлы формата JSON или BSON необходимо указать тип вывода `file`, формат вывода – JSON или BSON, и путь к файлу с логом.

```
storage:
     dbPath: data/db
auditLog:
     destination: file
     format: JSON  # BSON
     path: data/db/auditLog.json
```

8.8.3 Настройка фильтрации

MongoDB Enterprise поддерживает аудит различных операций. Если функция аудита включена, по умолчанию она записывает все

подлежащие аудиту операции. Но можно настроить фильтры, чтобы ограничить записываемые события. Фильтры можно настроить при запуске или во время выполнения экземпляра сервера.

Фильтр для нескольких типов операций. В примере ниже выполняется аудит только действий `createCollection` и `dropCollection` с использованием фильтра:

```
{atype:                                          {$in:
["createCollection","dropCollection"]}}
```

Чтобы указать фильтр аудита в конфигурационном файле, используется следующий формат:

```
storage:
     dbPath: data/db
auditLog:
     destination: file
     format: BSON
     path: data/db/auditLog.json
     filter:  '{atype:  {$in:  ["createCollection",
               "dropCollection"]}}'
```

Фильтрация операций аутентификации в одной базе данных. Поле `<field>` может включать любое поле в сообщении аудита. Для операций аутентификации (т.е. `atype: "authenticate"`) сообщения аудита включают поле `db` в документе `param`. В примере ниже выполняется аудит только операций проверки подлинности, которые выполняются в базе данных `test`:

```
{ atype: "authenticate", "param.db": "test" }
```

Чтобы отфильтровать все операции аутентификации в базах данных, можно опустить опцию `"param.db": "test"` и использовать фильтр `{ atype: "authenticate" }`.

Фильтрация операций создания и удаления коллекций для одной базы данных. Для операций создания и удаления коллекций сообщения аудита включают поле `ns` пространства имен в документе `param`. В следующем примере выполняется аудит операций, которые выполняются в базе данных `test`, отфильтрованной с использованием регулярного выражения:

```
{atype: {$in: ["createCollection", dropCollec-
tion"]}, "param.ns": /^test\\./ }}
```

Фильтр по роли авторизации. В следующем примере выполняется фильтрация событий аудита для операций пользователей с ролью `readWrite` в базе данных `test`, включая пользователей с ролями, унаследованными от `readWrite`:

```
{ roles: { role: "readWrite", db: "test" } }
```

Фильтрация операций чтения и записи. Чтобы фиксировать операции чтения и записи в аудите, необходимо также разрешить системе аудита регистрировать события авторизации с помощью параметра `auditAuthorizationSuccess`. Этот фильтр проверяет несколько операций чтения и записи:

```
{
   atype: "authCheck",
   "param.command": {$in: ["find", "insert", "de-
lete", "update", "findandmodify" ] }
}
```

Операции, подлежащие аудиту, включают в себя: `find()`, `insertOne()`,`insertMany()`,`remove()`,`updateOne()`,`updateMany()`, `findAndModify()`. Можно также указать конкретную коллекцию, события изменения данных в которой необходимо записывать в журнал аудита, с использованием опции `"param.ns"`.

8.8.4 Сообщения аудита

Функция аудита событий может записывать события в формате JSON. Выходные данные сообщений настраиваются.

Сообщения аудита имеют следующий синтаксис:

```
{
  atype: <string>,
  ts : { $date: <timestamp> },
  uuid : { $binary: <string>, $type: <string> },
  local: { ip: <string>, port: <int> || isSystemUser: <boolean> || unix: <string> },
  remote: { ip: <string>, port: <int> || isSystemUser: <boolean> || unix: <string> },
  users : [ { user: <string>, db: <string> }, ... ],
  roles: [ { role: <string>, db: <string> }, ... ],
  param: <document>,
  result: <int>
}
```

В сообщение:

– `atype` – тип события аудита, такой как `authenticate`, `createCollection`, `dropCollection`, `authCheck` и другие, которые рассматривались в примерах настройки фильтрации записей аудита;

– `ts` – документ, который содержит метку времени;

– `uuid` – идентификатор события аудита;

– `param` – документ, определяющий детали для каждого конкретного события;

– `result` – код ошибки.

Рассмотренная подсистема аудита доступна в коммерческой версии MongoDB, но можно использовать версию Percona Server for MongoDB, который имеет встроенную подсистему аудита.

9 МОНИТОРИНГ

9.1 Мониторинг инфраструктуры

Под мониторингом IT-инфраструктуры в общем смысле будем понимать автоматизированную систему контроля состояния элементов, включающую получение сведений о рабочих характеристиках системы путем регулярного измерения параметров функционирования различных компонентов инфраструктуры с целью оперативного выявления и устранения сбоев в работе системы.

Мониторинг нужен для решения следующих задач:

– исправления и изменения в связи с неисправностями. Необходимо знать, когда что-то повреждено или скоро будет повреждено, чтобы вовремя это исправить, не допуская нарушения целевых показателей качества обслуживания;

– анализ производительности и поведения. Важно понимать, каковы задержки в приложениях и где они возникают, а также видеть их изменения со временем, включая пиковые значения и всплески. Эти данные играют решающую роль в понимании того, как влияет использование новых функций и оптимизация;

– планирование мощностей. Возможность соотносить поведение пользователя и эффективность приложений с реальными ресурсами – процессором, сетью, хранилищем данных, пропускной способностью и памятью – крайне важна для предотвращения нехватки ресурсов в критический момент;

– отладка и послеаварийный анализ. Чем интенсивнее изменения, тем чаще что-нибудь оказывается повреждено. Хороший оперативный контроль дает возможность быстро находить точки сбоя и точки оптимизации, позволяющие снизить риск в будущем;

– бизнес-анализ. Понимание того, как используется функционал системы, может сыграть ведущую роль в выявлении проблем. Но не менее важно видеть, как пользователи используют функции системы и как это влияет на соотношение стоимости разработки и эффективности;

– причинно-следственные связи. Благодаря тому, что события в инфраструктуре и приложениях регистрируются и журналируются в стеке оперативного контроля, можно быстро соотнести изменения рабочей нагрузки, поведения и доступности. Примерами таких событий являются развертывание приложений, изменение инфраструктуры и изменение схемы баз данных.

9.1.1 Уровни системы мониторинга

Всю систему мониторинга можно поделить на несколько уровней.

Первый уровень – **уровень приложения**. Мониторинг показывает, когда и как пользователи пользуются приложением, и помогает выявить его слабые места. На этом уровне можно выделить следующие типы мониторинга:

– мониторинг бизнес-логики приложения. Проверяется сам факт того, что приложение работает, и оцениваем бизнес-показатели, например, количество просмотров веб-страниц, количество заказов, частоту регистраций, сумму покупок и так далее;

– мониторинг health-метрик сервисов. Проверяется жизнеспособность сервисов и всех ресурсов, требуемых для их стабильной работы;

– интеграционный мониторинг. Подразумевает проверку синхронной и асинхронной коммуникации между критическими для бизнеса системами.

Второй уровень – это **мониторинг приложения как инженерного продукта**. Этот вид мониторинга отражает ключевые моменты, от которых напрямую зависит доступность и производительность приложения. Виды мониторинга включают в себя:

– сбор и наблюдение журналов приложения;

– мониторинг производительности приложения (Application Performance Monitoring, APM) – мониторинг приложения с целью анализа ошибок в коде и того, как они влияют на производительность. Проверяется состояние физического оборудования и виртуальной машины, контейнера и самого приложения, вспомогательной инфраструктуры, кеша;

– трассировка – отслеживание всех вызовов для контроля выполнения в распределенных системах.

Третий уровень – **уровень инфраструктуры**. На уровне инфраструктуры подразумевается проверка всех систем, серверов и служб:

– мониторинг уровня оркестрации – проверка, насколько работоспособны контейнеры, кластеры и системы управления;

– мониторинг системного ПО – проверка, насколько хорошо функционирует операционная система и ее компоненты;

– мониторинг уровня аппаратного обеспечения – диагностика аппаратной части: процессора, материнской платы, жестких дисков и оптических накопителей.

Отдельный уровень настройки системы мониторинга – настройка оповещений для системы мониторинга, включающая в себя следующие меры:

– организация единой системы рассылки оповещения. Оповещения должны быть понятными (например, откуда они пришли). Необходим контроль доставки оповещений;

– организация системы дежурств. Оповещения не должны приходить всем, т.к. в этом случае на них либо будут реагировать все, или, что более вероятно, не будет реагировать никто.

Необходимо определить зоны ответственности с указанием кто, когда и за что отвечает;

– организация «базы знаний» обработки инцидентов. По каждому серьезному инциденту или ошибке в приложении фиксируются действия, которые решают проблемы.

9.2 Мониторинг баз данных

Мониторинг баз данных – это комплексная задача, которая включает в себя отслеживание показателей производительности баз данных на различных уровнях: SQL (т.е. мониторинг оптимальности запросов), экземпляр БД, инфраструктура, пользователи.

Выделяют следующие типичные задачи мониторинга:

– мониторинг доступности. Основная задача БД как сервиса – это предоставление данных клиентам, следовательно, в первую очередь необходимо мониторить доступность базы, а также ее некоторые качественные и количественные характеристики;

– мониторинг подключенных пользователей и их активности. К базе могут подключаться как авторизованные клиенты, так и вредоносные пользователи, деятельность которых необходимо отслеживать;

– мониторинг работы с данными. Необходимо мониторить, с какими данными, т.е. таблицами и в меньше степени индексами, работает пользователь, чтобы оценить нагрузку на СУБД;

– мониторинг запросов. Основная нагрузка на базу приходится на запросы, поэтому важно оценивать запросы и их производительность. Используя эту информацию, можно переписать запрос или добавить необходимые индексы;

– мониторинг фоновых процессов. Фоновые процессы позволяют поддерживать производительность базы данных, что

также требует некоторых ресурсов. Необходимо отслеживать, что фоновые процессы не влияют на производительность клиентских запросов;

– мониторинг системных метрик. Если база данных развернута в облачном хранилище, мониторинг системных метрик отдельного хоста отходит на второй план, но задача мониторинга системных метрик базы данных все еще остается актуальной.

9.3 Иерархия мониторинга

Существует большое количество показателей базы данных, системы, хранилища и различных приложений прикладного уровня, которые можно контролировать. Схематично, можно выделить несколько уровней иерархии системы мониторинга.

1. Проверка доступности базы данных.
2. Мониторинг общих показателей задержек/ошибок и сквозной мониторинг работоспособности.
3. Мониторинг прикладного уровня для измерения задержек/ошибок для каждого вызова базы данных.
4. Сбор как можно большего количества показателей, относящихся к уровням системы, хранилища, базы данных и приложений, независимо от того, будут они полезны или нет. Для большинства операционных систем, сервисов и БД это обеспечивается подключаемыми модулями.
5. Специальные проверки для отдельных известных проблем. Например, проверки поведения при отключении определенного процента узлов базы данных или слишком высокого процента глобальных блокировок.

9.4 Мониторинг хранилища данных

Отчасти от типа хранилища данных зависит ответ на вопрос, что может потребоваться отслеживать в базах данных? Но все параметры мониторинга можно разбить на 4 группы:

- мониторинг на уровне соединения с хранилищем данных;
- контроль процессов внутри базы данных;
- мониторинг объектов базы данных;
- мониторинг запросов к базе.

9.4.1 Мониторинг соединения с хранилищем данных

Первый уровень – это уровень соединения с хранилищем данных. На этом уровне может потребоваться мониторинг времени, которое требуется для подключения к внутреннему хранилищу данных в рамках транзакции. Также необходимо осуществлять контроль максимального и фактического количества соединений с базой данных. В соответствии с параметрами конфигурации база данных сможет принимать только определенное количество соединений, устанавливая искусственную верхнюю границу для минимизации риска перегрузки сервера. Контроль этого максимума, как и фактического количества соединений, имеет важное значение, поскольку по умолчанию он может быть установлен произвольно низким.

Оценка уровня загрузки бывает наиболее полезен в сочетании с коэффициентом использования. Если в системе есть задержки и перегрузка без полного использования, это может означать, что где-то есть узкое место, которое вызывает коллизии. Загруженность может быть измерена в таких типичных точках, как:

- журнал TCP-соединений;
- превышение времени ожидания соединения;
- ожидание потоков в пулах соединений;

– подкачка страниц памяти;
– блокировки процессов в базе данных.

Для распознавания перегрузок решающее значение имеют длина очереди и время ожидания. Каждое обнаруженное ожидание соединения или процесса – это индикатор потенциального узкого места.

Кроме того, необходимо отслеживать ошибки и использовать сведения о них для выявления и устранения неисправностей и/или проблем конфигурации. Ошибки можно зафиксировать различными способами, в т.ч используя журналы базы данных, которые предоставляют коды ошибок, возникающих при сбоях на уровне базы данных, журналы приложений и прокси-серверов, а также журналы хоста.

9.4.2 Мониторинг процессов внутри базы данных

Следующий уровень – это мониторинг процессов внутри базы данных. На этом уровне оцениваются следующие показатели:

– показатели пропускной способности и задержки;
– состояние репликации;
– коллизии параллельного доступа и блокировки.

Показатели пропускной способности и задержки позволяют оценить, сколько и какие именно операции выполняются в хранилищах данных. Эти сведения позволяют составить полное представление о работе базы. В качестве примеров таких операций можно выделить следующие: операции чтения, операции записи (такие как вставка, модификация и удаление записей), операции подтверждения и отката изменений в транзакции, использование DDL-операторов для создания объектов базы данных и другие административные задачи.

Следующий процесс, состояние которого необходимо отслеживать – это репликация. При репликации возможны три ситуации,

которые могут вызвать проблемы, если их вовремя не отслеживать и не предотвращать.

– задержка репликации. Иногда применение изменений к другим узлам может замедляться, в том числе из-за загруженности сети;

– прерванная репликация. Процессы, реализующие репликацию данных, могут прерваться из-за любых возникших ошибок. Решать эту проблему нужно быстро, чему способствует мониторинг, с последующим устранением причины ошибок и возобновлением репликации;

– несогласованность данных, что делает репликацию бесполезной и потенциально опасной.

Следующий показатель, который необходимо использовать – это коллизии параллельного доступа и блокировки. В реляционных базах данных активно используются блокировки для поддержки одновременного доступа из нескольких сеансов. Блокировки позволяют выполнять операции чтения и модификации, гарантируя при этом, что другие процессы не смогут в это время изменить данные. Использование блокировок может приводить к задержкам, поскольку образуется очередь процессов, ожидающих снятия блокировки. В некоторых случаях возможны превышения тайм-аутов процессов из-за взаимных блокировок, для которых ситуации не могут быть разрешены иначе, чем откатом к предыдущему состоянию.

9.4.3 Мониторинг объектов базы данных

Для мониторинга также важно понимать, что представляет собой база данных и как в ней хранятся данные. В простейшем случае это понимание того, сколько памяти занимают каждый объект базы данных и связанные с ним ключи и индексы. Как и в случае с хранилищем на основе файловой системы, понимание скорости роста и

времени, за которое будет достигнута верхняя граница, здесь столь же важно, если не важнее, чем знание текущей загрузки хранилища.

Полезно не только понимать, как хранятся данные и как увеличивается хранилище, но и контролировать характер распределения критически важных данных. Например, знать верхние и нижние границы, средние значения и область значений для элементов данных полезно для понимания производительности операций индексации и поиска. Это особенно важно для целочисленных типов и символьных данных с малым разнообразием значений, т.к. позволяет оптимально выбирать типы данных и индексации.

Если набор данных сегментирован с использованием диапазонов значений ключа или списков, то понимание того, как происходит распределение между сегментами, поможет обеспечить максимальную производительность для каждого узла. Мониторинг позволит сформулировать рекомендации о необходимости изменения балансировки или пересмотра моделей сегментирования данных.

9.4.4 Мониторинг запросов к базе данных

Мониторинг запросов к базе данных может включать в себя потребление ресурсов процессора и ввода-вывода, количество прочитанных и записанных строк, детализированное время выполнения и время ожидания, а также счетчик выполненных операций. Решающее значение для оптимизации имеет понимание способов оптимизации, применяемых индексов и статистики по объединению, сортировке и агрегированию.

9.5 Мониторинга в PostgreSQL.

В MySQL и MongoDB подсистемы мониторинга доступны только в коммерческих версиях, однако можно использовать сто-

ронние модули, которые позволяют собирать разнообразные метрики о параметрах работы СУБД и визуализировать их в системах мониторинга типа Zabbix или Prometheus.

В PostgreSQL существует подсистема накопительной статистики, которая позволяет собирать статистику о работе всех компонентов СУБД и предоставляет интерфейс в виде набора хранимых процедур и представлений для просмотра статистики. В настоящее время подсистема позволяет отслеживать обращения к таблицам и индексам как на уровне блоков на диске, так и на уровне отдельных строк, собирать информацию о выполняемых для каждой таблицы действиях очистки и анализа, выполнять подсчёт вызовов пользовательских функций и времени, затраченного на выполнение каждой из них.

Кроме того, PostgreSQL может предоставить динамическую информацию о том, что происходит в системе в текущий момент времени, в частности, сообщить, какие именно команды выполняются другими серверными процессами и какие другие соединения существуют в системе. Эта возможность не зависит от системы накопительной статистики.

Поскольку сбор статистики несколько увеличивает накладные расходы при выполнении запроса, есть возможность настроить СУБД так, чтобы выполнять или не выполнять сбор статистической информации. Это контролируется конфигурационными параметрами, которые обычно устанавливаются в файле `postgresql.conf`.

На рисунке 22 приведена диаграмма основных компонентов СУБД и соответствующих процедур и представлений [13]. Благодаря системе накопительной статистики можно получить информацию о подключении клиентов, выполнении запросов, использовании индексов и таблиц, мониторить формирование WAL-логов транзакций, состояние репликации и другие рабочие процессы PostgreSQL.

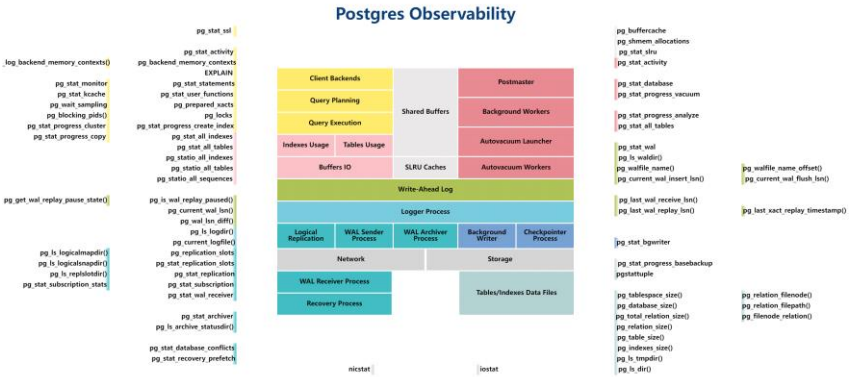

Рисунок 22 – Система накопительной статистики

Для просмотра текущего состояния системы предназначены несколько предопределённых представлений, которые перечислены в таблице 9.1.

Таблица 9.1. Представления для определения текущего состояния системы

Представление	**Описание**
`pg_stat_activity`	Одна строка для каждого серверного процесса с информацией о текущей активности процесса, включая его состояние и текущий запрос.
`pg_stat_replication`	По одной строке для каждого процесса-передатчика WAL со статистикой по репликации на ведомом сервере, к которому подключён этот процесс.
`pg_stat_wal_receiver`	Одна строка со статистикой приёмника WAL, полученной с сервера, на котором работает приёмник.
`pg_stat_subscription`	Одна строка для подписки, сообщающая о рабочих процессах подписки.

Окончание табл. 9.1

Представление	Описание
pg_stat_gssapi	Одна строка для каждого подключения, в которой показывается информация об использовании аутентификации и шифровании GSSAPI для данного подключения.
pg_stat_progress_analyze	По одной строке с текущим состоянием для каждого обслуживающего процесса, в котором работает ANALYZE.
pg_stat_progress_create_index	По одной строке с текущим состоянием для каждого обслуживающего процесса, в котором выполняется CREATE INDEX или REINDEX.

Также есть возможности получение информации о процессах, выполняющих резервное копирование, операцию COPY и другие операции.

В дополнение к рассмотренным выше представлениям есть несколько других представлений, перечисленных в таблице 9.2, позволяющих просмотреть собранную статистику. Кроме того, на базе существующих функций накопительной статистики можно создать собственные представления.

Также существуют представления для просмотра статистики только по пользовательским таблицам, только по системных таблицам и так далее.

Таблица 9.2. Спецификации для просмотра собранной статистики

Представление	Описание
pg_stat_archiver	Одна строка со статистикой работы процесса архивации WAL.
pg_stat_database	Одна строка для каждой базы данных со статистикой на уровне базы.

Окончание табл. 9.2

Представление	Описание
pg_stat_all_tables	По одной строке на каждую таблицу в текущей базе данных со статистикой по обращениям к этой таблице.
pg_stat_all_indexes	По одной строке для каждого индекса в текущей базе данных со статистикой по обращениям к этому индексу.
pg_statio_all_tables	По одной строке для каждой таблицы в текущей базе данных со статистикой по операциям ввода/вывода с этой таблицей.
pg_statio_all_indexes	По одной строке для каждого индекса в текущей базе данных со статистикой по операциям ввода/вывода для этого индекса.

9.5.1 Примеры просмотра статистики

Чтобы получить динамическую информацию о текущих подключениях, можно выполнить запрос к представлению pg_stat_activity. Результат запроса будет содержать информацию по каждому процессу, включая идентификатор БД, к которой подключен этот серверный процесс, имя базы данных, имя пользователя, подключенного к этому серверному процессу, название приложения, подключённого к этому серверному процессу, общее текущее состояние серверного процесса, текст последнего запроса и другую информацию. Часть информации, полученной при выполнении следующего запроса, представлена на рисунке 23:

```
SELECT * FROM pg_stat_activity;
```

	datid	datname	usename	application_name	state	query
	5	postgres	postgres	pgAdmin 4 - DB:postgres	idle	…
	16908	pagila	postgres	pgAdmin 4 - DB:pagila	idle	…
	16908	pagila	postgres	pgAdmin 4 - CONN:729358	active	SELECT * FROM pg_stat_activity;

Рисунок 23 – Просмотр информации о текущих подключениях

В следующем примере выполняется просмотр статистики использования пользовательских таблиц. В этом случае используется представление, выдающее накопленную статистику. Результат выборки аналогичен представлению `pg_stat_all_tables`, за исключением того, что отображаются только пользовательские таблицы. Результат содержит идентификатор таблицы, имя схемы, в которой расположена эта таблица, имя таблицы, количество последовательных чтений, произведённых в этой таблице, количество вставленных строк, количество изменённых строк, количество удалённых строк и другую информацию. Часть информации о статистике использования пользовательских таблиц представлена на рисунке 24:

```
SELECT * FROM pg_stat_user_tables;
```

	relid	schemaname	relname	seq_scan	n_tup_ins	n_tup_upd	n_tup_upd
	17004	public	country	1	100	0	0
	16935	public	customer	6	599	0	0
	16947	public	actor	34	205	19	0

Рисунок 24 – Просмотр статистики использования пользовательских таблиц

Используя эту информацию, можно оценить нагрузку на таблицы базы данных. Следующий вопросом является изучение запросов, вызывающих наибольшую нагрузку на сервер. Необходимость обнаружения долгих запросов связана с тем, что оптимизация запросов, как правило, дает больший прирост производительности, чем оптимизация конфигурации PostgreSQL, операционной системы, или даже оптимизация аппаратного обеспечения.

Модуль `pg_stat_statements` предоставляет возможность отслеживать статистику выполнения сервером всех операторов SQL.

Этот модуль нужно загружать, добавив `pg_stat_statements` в `shared_preload_libraries` в

файле `postgresql.conf`, так как ему требуется дополнительная разделяемая память. Для загрузки или выгрузки модуля необходим перезапуск сервера.

Когда модуль `pg_stat_statements` загружается, он отслеживает статистику по всем базам данных на сервере. Для получения и обработки этой статистики этот модуль предоставляет представление `pg_stat_statements` и вспомогательные функции `pg_stat_statements_reset` и `pg_stat_statements`.

Используя информацию из этого модуля, можно получать информацию по наиболее часто выполняемым запросам, мониторить самые долгие запросы и т.д. Также можно оценивать и мониторить самые тяжелые запросы в плане использования ресурсов, те, которые читают с диска, работают с памятью или, наоборот, создают какую-то нагрузку по записи на диск. Кроме того, можно отслеживать те запросы, которые возвращают большое количество строк. В добавок, можно мониторить запросы, которые используют временные файлы или временные таблицы.

9.6. Системы мониторинга

Рассмотренные выше метрики редко собираются вручную, обычно они собираются автоматически с использованием инструментов мониторинга IT-инфраструктуры, которые позволяют полностью отслеживать ее состояние: собирать данные, анализировать и визуализировать их, оповещать о сбоях и т.д.

9.6.1 Zabbix

Одной из наиболее популярных систем мониторинга является Zabbix – свободно-распространяемая система мониторинга статусов разлиных сервисов компьютерной сети, серверов и сетевого оборудования.

У Zabbix есть собственный веб-интерфейс с гибкими настройками. Можно настроить информационные панели на основе виджетов, отображать графики, карты сети, слайд-шоу и детализированные отчеты. По умолчанию интерфейс Zabbix предоставляет несколько предопределенных тем, но можно создавать свои собственные дашборды.

9.6.1.1 Компоненты системы

С точки зрения архитектуры Zabbix состоит из нескольких программных компонентов: сервер, база данных, веб-интерфейс, прокси и агенты.

Zabbix-сервер – это ядро системы, которое дистанционно контролирует сетевые сервисы и является хранилищем, в котором содержатся все конфигурационные, статистические и оперативные данные. Он является тем субъектом в программном обеспечении Zabbix, который оповещает администраторов о проблемах с контролируемым оборудованием. Для хранения данных используется MySQL, PostgreSQL, SQLite или Oracle Database.

Веб-интерфейс – часть Zabbix-сервера, которая, как правило (но не обязательно), запускается на том же физическом узле, что и Zabbix-сервер.

Zabbix-агент – программа контроля локальных ресурсов и приложений на сетевых системах (таких как накопители, оперативная память, статистика процессора и т.д.), эти системы должны работать с запущенным Zabbix-агентом.

Zabbix-прокси собирает данные о производительности и доступности от имени Zabbix-сервера. Все собранные данные заносятся в буфер на локальном уровне и передаются Zabbix-серверу, к которому принадлежит прокси-сервер. Zabbix-прокси целесообразно применять для дистанционного контроля филиалов и других точек, в том числе сетей, не имеющих местных администраторов. Zabbix-прокси может быть также использован для распределения

нагрузки одного Zabbix-сервера. В этом случае прокси только собирает данные, тем самым на сервер ложится меньшая нагрузка на процессор и на устройства ввода/вывода.

9.6.1.2 Архитектура системы

Архитектуру системы можно представить в виде следующей диаграммы, представленной на рисунке 25.

Можно выделить следующие программные компоненты: сервер, база данных, веб-интерфейс, прокси и агенты. Кроме того, есть возможность взаимодействия с утилитами командной строки. Начиная с версии 5.2 в Zabbix появилась интеграция с хранилищем секретов Hashicorp Vault, в котором можно хранить любую чувствительную информацию от логина/пароля в БД до значений макросов. Это позволит централизованно управлять авторизационными данными, вести аудит и не хранить данные в файловой системе или в БД Zabbix. Все информационные потоки могут использовать зашифрованное HTTPS соединение. Для взаимодействия с базой данных в БД могут быть созданы пользователи и роли с атрибутами, требующими шифрованное подключение.

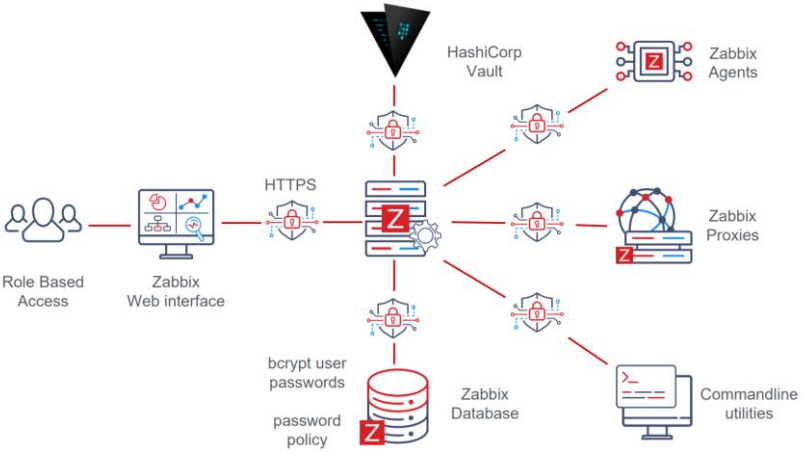

Рисунок 25 – Архитектура системы мониторинга Zabbix

9.6.1.3 Возможности Zabbix

Для хранения данных используется внешняя БД MySQL, PostgreSQL, SQLite или Oracle Database. Соответственно, для выборки определенных данных необходимо обращаться напрямую к базе с использованием языка SQL.

В Zabbix встроена функция оповещения, что позволяет управлять событиями различными способами: отправка сообщений, выполнение удаленных команд и т.д. Также можно настраивать сообщения в зависимости от роли получателя, выбирая, какую информацию включать, например, дату, время, имя хоста, значение элементов, значения триггеров, профиль хоста и т.д.

9.6.1.4 Преимущества и недостатки Zabbix

Zabbix можно считать системой мониторинга общего назначения, на которую можно замкнуть всю инфраструктуру. Например, настроить мониторинг ssl сертификатов и время делегирования домена. Также в качестве достоинств можно отметить следующие:

– единая точка доступа для всех пользователей;

– разграничение доступа к данным и к конфигурации;

– поддержка всех основных платформ (Linux, MacOS, Windows);

– в качестве базы данных для хранения метрик можно использовать MySQL, PostgreSQL, SQLite или Oracle;

– широкие возможности мониторинга.

К недостаткам можно отнести следующие:

– сложная настройка шаблонов для сбора метрик из различных приложений;

– сложная и долгая настройка масштабирования;

– хранение всех данных мониторинга в базе данных (что требует дополнительных вычислительных мощностей);

– мониторинг происходит через агента, который работает постоянно и устанавливается на каждом сервере.

9.6.2 Prometheus

Следующая система мониторинга – Prometheus. Изначально Prometheus предназначался для мониторинга контейнерных сред, но со временем возможности системы расширились до приложений, серверов, баз данных и виртуальных машин.

Prometheus предоставляет web-интерфейс с отображением графиков практически по любому сочетанию параметров, но сами графики не очень удобные, поэтому Prometheus часто используют вместе с Grafana. Grafana – это полноценный инструмент визуализации, который включает встроенную поддержку Prometheus.

Grafana предоставляет список панельных модулей, позволяющих создавать визуализации с такими параметрами, как карты мира, тепловые карты, гистограммы, круговые и линейные диаграммы.

9.6.2.1 Компоненты системы

Сервер Prometheus (Prometheus Server) – центральный компонент системы мониторинга, чья основная задача – хранить и мониторить определенные объекты. Объектом может стать что угодно: Linux-сервер, сервер Apache, один из процессов, сервер базы данных или любой другой компонент системы, который нужно контролировать. В терминах Prometheus объекты мониторинга называются целевыми объектами (Prometheus targets). Через конечную точку HTTP сервер Prometheus проверяет статус приложения.

Каждый элемент целевого объекта, который необходимо мониторить (статус центрального процессора, память или любой другой элемент), называется метрикой. Таким образом, Prometheus собирает через HTTP метрики целевых объектов, хранит их локально или удаленно и отображает.

Prometheus собирает метрики на основе методов `push` и `pull`. В первом методе отслеживаемое приложение отвечает за отправку метрик в систему мониторинга с помощью Push шлюза (Pushgateway). В методе `pull` приложение подготавливает метрики, а сервер Prometheus считывает целевые объекты с интервалом, который определяет сбор метрик, и сохраняет их в базе данных временных рядов (Time-Series database, TSDB). Целевые объекты и временной интервал считывания метрик задаются в конфигурационном файле `prometheus.yml`.

Компонент управления оповещениями AlertManager служит для запуска оповещений через Email, Slack или другие клиентские уведомления.

Для визуализации данных можно использовать как пользовательский веб-интерфейс Prometheus (web UI), так и сторонние клиенты (наиболее популярный – Grafana).

Для мониторинга сторонних систем (таких как сервер Linux, сервер MySQL и т.д.) можно использовать экспортеры. Экспортер – часть программного обеспечения, которое получает существующие метрики от сторонней системы и экспортирует их в формат, понятный серверу Prometheus.

В качестве примеров таких экспортеров можно привести:

– `node_exporter` – выдача метрик файловой системы;

– `postgres_exporter` – выдача метрик базы данных PostgreSQL;

– `mongodb_exporter` – выдача метрик базы данных MongoDB.

Описанные компоненты как часть архитектуры Prometheus показаны на рисунке 26.

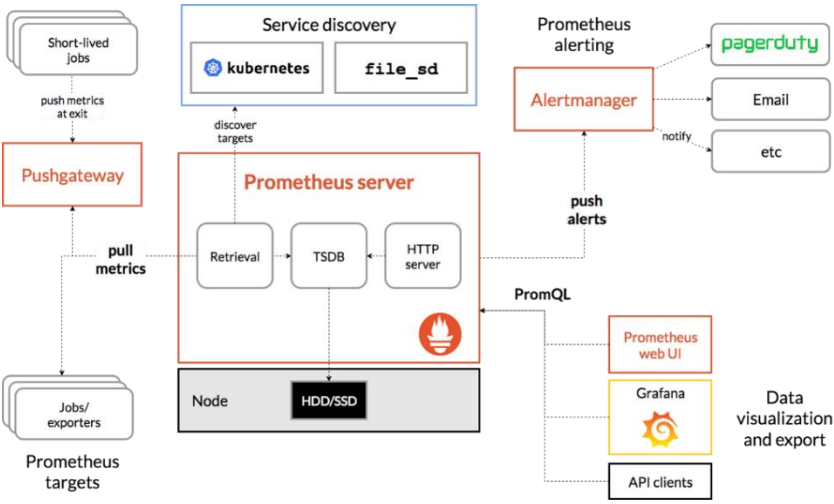

Рисунок 26 – Архитектура системы мониторинга Prometheus

9.6.2.2 Возможности Prometheus

Prometheus хранит данные в виде БД временных рядов (TSDB). Как следствие, Prometheus не подходит для хранения текста, журналов или журналов событий. Если Prometheus используется в связке с Grafana, можно использовать систему агрегации логов Loki Grafana. Хотя для хранения логов лучше использовать специализированные продукты.

Prometheus предоставляет собственный язык для запросов, который называется PromQL, что делает эту систему более гибкой по сравнению с Zabbix. Язык запросов может применять функции и операторы к запросам метрик, фильтровать, группировать по меткам и использовать регулярные выражения для улучшения сопоставления и фильтрации. Результат выражения можно отобразить в виде графика, просмотреть в виде табличных данных в браузере или использовать во внешних системах через HTTP API.

Для настройки оповещений в Prometheus необходимо установить Alertmanager. Это связано с тем, что оповещение с помощью

Prometheus разделено на две части. Правила оповещения пересылают оповещения в систему Alertmanager, которая решает задачи дедупликации оповещений, группировки и отправки получателям. Уведомления можно отправлять по электронной почте, через системы уведомлений по телефону и через чаты.

9.6.2.3 Преимущества и недостатки Prometheus

Таким образом, Prometheus – более специализированная система мониторинга, в качестве достоинств которой можно выделить:

– хорошо подходит для динамических систем, например, Kubernetes. Позволяет автоматически находить необходимые целевые объекты для мониторинга;

– формат данных Prometheus поддерживает большое количество приложений. Они сразу выдают все метрики для Prometheus, остается только указать их адреса в настройках;

– свой язык запросов (PromQL, Prometheus query language), с помощью которого можно удобно и быстро построить запрос на выборку данных;

– отличается высокой производительностью за счет использования базы данных временных рядов: позволяет собирать данные с тысячи серверов с интервалом в десять секунд;

– предусмотрена возможность масштабирования;

– простота настройки.

Недостатками являются:

– необходимость дополнительной настройки внешней базы данных, если необходимо хранить исторические данные месяцы и годы, т.к. изначально Prometheus был рассчитан на краткосрочное хранение метрик и работы с ними в базе данных временных рядов;

– Prometheus хранит у себя только значения временных рядов и не подходит для текста, логов, журналов событий;

– отсутствие возможности аутентификации и авторизации пользователей. Для аутентификации предлагается использовать ограничения на firewall или настроить аутентификацию типа `basic_auth` при проксировании через nginx;

– в Prometheus необходимо использовать сторонние приложения для визуализации данных, хотя интеграция с Grafana очень простая и дает более удобную визуализацию, чем встроенная в Zabbix.

В результате, Zabbix более подходит для низкоуровневого мониторинга устройств, операционных систем или сетей с долговременным хранением данных о работе устройств, Prometheus более специализирован на сборе и хранении метрик, особенно для сбора метрик приложений.

10 ШИФРОВАНИЕ

10.1 Основные определения

В предыдущих разделах уже было рассмотрено достаточное количество уязвимостей, характерных для СУБД, и способов предотвращения их эксплуатации злоумышленниками. Однако даже при грамотном администрировании БД, при своевременном обновлении компонентов СУБД и применении прочих действий по предотвращению разглашения конфиденциальных данных существуют сценарии, при которых оно может быть осуществлено. Например, с привлечением сотрудника организации, имеющего доступ к файлам БД, кражей физического носителя данных и т.д. Для предотвращения инцидентов существует единственный способ – сделать данные нечитаемыми для тех, кому они не предназначены, с использованием криптографических методов защиты данных.

Криптографическая защита данных – это одно из самых сильных средств противодействия несанкционированному просмотру данных. В основе подавляющего большинства криптографических систем данных защиты выступает шифрование.

Шифрование – это процесс преобразования открытых данных с использованием специального алгоритма, после чего эти данные не могут быть восстановлены к исходному виду без ключа дешифрования. **Шифрование базы данных** – использование технологии шифрования для преобразования информации, хранящейся в базе данных, в шифротекст, что делает её прочтение невозможным для лиц, не обладающих ключами шифрования.

Кроме того, необходимо шифровать обмен данными между БД и любыми клиентами или приложениями, которым требуется доступ к базе данных – так называемое **шифрование подвижных данных**.

Схематично процесс шифрования и дешифрования данных представлен на рисунке 27.

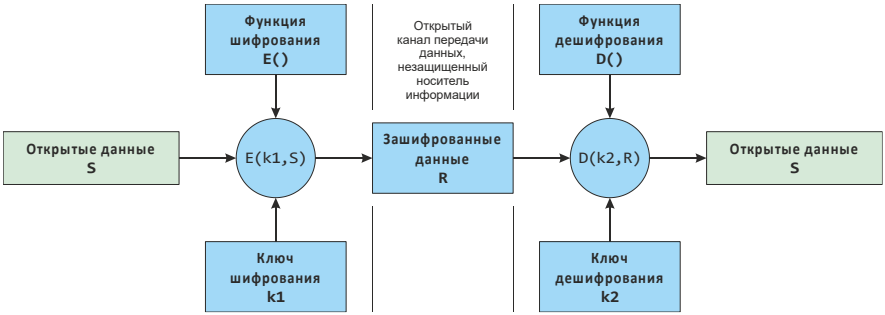

Рисунок 27 – Процесс шифрования

Исходные открытые данные S подвергаются криптографическому преобразованию, для этого они обрабатываются шифрующей функцией $E(k1, S)$, в качестве параметра которой выступает назначаемый пользователем ключ шифрования $k1$. В результате получаются зашифрованные данные R, которые для стороннего наблюдателя выглядят как последовательность случайных символов. Восстановление открытых данных S из зашифрованных данных R возможно только в том случае, если известны функция дешифрования D и ключ дешифрования $k2$.

Различают два вида криптосистем: на основе симметричного и асимметричного шифрования. В симметричной криптосистеме для шифрования и дешифрования данных применяется один и тот же ключ. В таких системах для исключения несанкционированного доступа к информации ключ следует держать в секрете. В асимметричной криптосистеме применяется два ключа. Шифрование данных осуществляется несекретным ключом (для дешифрования данных он не подходит), а дешифрование проводится с помощью второго секретного ключа.

Защищенная СУБД должна уметь шифровать хранящиеся в ней данные (включая служебную информацию), исходный код запросов, хранимых процедур и триггеров, данные, передаваемые к другим компьютерам по незащищенным каналам.

10.2 Виды шифрования БД

Поскольку информация так или иначе хранится в базе данных, то по большому счету не важно в каком виде ее хранить, в оригинальном или в преобразованном. Шифрование баз данных может быть осуществлено с помощью различных подходов:

- шифрование на уровне хранилища;
- шифрование на уровне базы данных;
- шифрование на уровне приложения.

10.2.1 Шифрование на уровне хранилища

Шифрование баз данных на уровне хранилища также называют «прозрачным» (Transparent Database Encryption, TDE). Прозрачное шифрование данных используется для шифрования всей базы данных, что, следовательно, подразумевает шифрование «неактивных» данных. Под «неактивными» данными понимаются данные, которые в настоящее время не редактируются или не перемещаются по сети. Данные шифруются перед записью на диск и дешифруются во время чтения в память.

Основным преимуществом прозрачного шифрования является то, что шифрование и дешифрование выполняются «прозрачно» для приложений, то есть не требуется их модификация. Также при использовании такого вида шифрования кроме непосредственно файлов базы данных также шифруются её резервные копии. Недостатком можно считать то, что данный вид шифрования не обеспечивает сохранность информации при передаче по каналам

связи или во время использования. Соответственно требуются дополнительные меры для шифрования данных на транспортном уровне.

К этому же виду шифрования также можно отнести шифрование на уровне файловой системы, например, с использованием системы шифрования данных EFS (Encrypting File System), реализованной в ОС Windows. Такой механизм шифрования также является «прозрачным» для пользователей и приложений, однако осуществляется не средствами СУБД, а средствами ОС. В отличие от прозрачного шифрования данных, автоматического шифрования резервных копий не происходит.

10.2.2 Шифрование на уровне базы данных

Шифрование на уровне базы данных используется для шифрования значений отдельных столбцов (Column-Level Encryption). Данные шифруются и дешифруются непосредственно в момент их использования. В базу данных помещаются уже зашифрованные значения, сама же база данных записывается на диск без дальнейшего шифрования.

Основным преимуществом данного вида шифрования является возможность шифрования отдельных столбцов, что повышает гибкость шифрования в целом. Для каждого зашифрованного столбца можно использовать уникальный ключ шифрования, что повышает надежность защиты конфиденциальных данных. Кроме того, данные не расшифровываются, пока не придет время их использовать, то есть данные, загруженные из хранилища в память, зашифрованы.

Среди недостатков шифрования на уровне базы данных можно отметить необходимость изменения схемы данных, так как все зашифрованные данные хранятся в двоичном виде. Кроме того, снижается общая производительность СУБД. Во-первых, из-за дополнительной обработки при шифровании и расшифровке данных. Во-

вторых, большего времени требует просмотр таблицы, поскольку индексы по зашифрованным столбцам не могут быть использованы.

10.2.3 Шифрование на уровне приложения

В шифровании на уровне приложений процесс шифрования осуществляется приложением, которое создаёт или изменяет данные, то есть шифрование данных происходит перед их записью в базу. Такой подход является весьма гибким, так как позволяет настроить процесс шифрования для каждого пользователя на основе информации о его правах в приложении.

Одним из главных преимуществ шифрования, встроенного в приложение, является то, что нет необходимости использовать дополнительное решение для защиты данных при передаче по каналам связи, так как они отправляются уже зашифрованными. Ещё один плюс такого метода – кража конфиденциальной информации становится сложнее, так как злоумышленник должен иметь доступ не только к данным, но и к приложению, чтобы расшифровать данные.

Основным недостатком такого подхода является то, что для реализации шифрования на уровне приложений необходимо внесение изменений как в приложение, так и в базу данных. Также могут возникнуть проблемы с производительностью БД, у которой пропадает возможность индексирования и поиска по зашифрованным данным. Ещё одним минусом является сложность управления ключами в системе с таким шифрованием, так как несколько приложений могут использовать БД, и, следовательно, ключи хранятся во многих местах.

10.3 Риски шифрования данных

К рискам, возникающим при шифровании данных, можно отнести следующие:

– ненадлежащее управление ключами. Если управление ключами шифрования не происходит в «изолированной системе», возникает риск того, что неблагонадежные сотрудники могут иметь возможность расшифровать конфиденциальные данные с помощью ключей, к которым они получат доступ;

– утрата ключа. Утеря ключей порождает риск потери данных, так как расшифровка без ключей практически невозможна.

10.4 Шифрование подвижных данных

За исключением шифрования на уровне приложения, остальные варианты обеспечивают безопасность только «неподвижных» данных (data at rest) – тех данных, которые не передаются по сети. Однако в процессе передачи данных между клиентом и сервером также существует возможность перехвата нешифрованных данных злоумышленником. Такая уязвимость потенциально может эксплуатироваться следующими типами атак:

– подслушивание (Eavesdropping). Атака на сетевом уровне, которая сфокусирована на перехвате пакетов, передаваемых по сети, и чтении их содержимого в поисках интересующей информации;

– посредник (Man-in-the-Middle). Атака на сетевом уровне, в ходе которой злоумышленник встраивается в канал связи между клиентом и сервером и тайно ретранслирует трафик, имея возможность анализировать и изменять содержимое пакетов.

Использование протоколов шифрования «подвижных» данных (data at transit) позволяет предотвратить описанные угрозы. Реализуется такой подход либо посредством установления шифрованного соединения между клиентом и сервером, при этом СУБД и

клиент должны поддерживать шифрованные соединения, либо с помощью организации защищенного туннеля между клиентом и сервером. Второй подход применим для любых СУБД, т.к. реализуется сторонними программными средствами.

10.4.1 Использование шифрованного соединения

Суть использования шифрованного соединения заключается в шифровании и дешифровании сетевого траффика на концах устанавливаемого соединения, соответственно во время передачи по каналу связи данные находятся в зашифрованном виде. Существует ряд криптографические протоколов, обеспечивающих защищенную передачу данных в компьютерной сети. В настоящее время отраслевым стандартом для защиты соединений TCP/IP является принцип шифрованного соединения на основе протокола SSL/TLS.

SSL (Secure Sockets Layer – уровень защищённых сокетов) представляет собой криптографический протокол для безопасной связи. С версии 3.0 SSL заменили на TLS (Transport Layer Security – безопасность транспортного уровня). Хотя формально это два различных протокола, отличающиеся в некоторых деталях реализации, ввиду того что TLS является развитием SSL, в настоящее время под SSL чаще всего подразумевают TLS. Цель протокола – обеспечить защищенную передачу данных. При этом для аутентификации используются асимметричные алгоритмы шифрования (пара открытый-закрытый ключ), а для сохранения конфиденциальности, с целью повышения скорости обработки данных – симметричные (секретный ключ).

Процесс установления соединения между клиентом и сервером называется рукопожатием (Handshake):

1) при запуске клиентского приложения оно запрашивает информацию о сертификате у сервера, который в свою очередь высылает копию SSL-сертификата с открытым ключом;

2) клиент проверяет подлинность сертификата, дату действия сертификата и в некоторых случаях наличие корневого сертификата, выданного авторизованным центром сертификации;
3) если подлинность сертификата подтверждена, то клиент генерирует предварительный секрет (Pre-master Secret) сессии на основе открытого ключа, полученного от сервера, используя максимально высокий уровень шифрования, который поддерживают обе стороны;
4) сервер расшифровывает предварительный секрет с помощью своего закрытого ключа и использует его для вычисления общего секрета (Master Secret), используя определенный вид шифрования (два самых распространённых из них – RSA и Диффи-Хеллман).

После завершения процесса рукопожатия стороны используют симметричный ключ, действительный только для данной сессии (его также называют сессионный ключ) для шифрования пакетов. После завершения сессии ключ уничтожается, а при следующем установлении соединения описанный процесс запускается заново.

10.4.2 Использование защищённых туннелей

Следующий подход – использование защищённых туннелей – позволяет обеспечить должный уровень защиты сетевого трафика для клиентов, не поддерживающих SSL. Суть этого подхода заключается в организации защищенного туннеля между определенными портами клиента и сервера. Со стороны клиента это может быть произвольный свободный порт (например, 63333), со стороны сервера – порт, который слушает сервер БД (гипотетически, это также любой свободный порт, но для различных СУБД есть значения по умолчанию: 3306 для MySQL, 5432 для PostgreSQL).

Для организации тунеллирования могут использоваться различные службы, например, ssh, stunnel, ipseq и т.д. Различаются эти протоколы как-правило тем, на каком уровне TCP/IP они работают и, соответственно какие возможности предоставляют.

Для клиента подключение к удаленному серверу БД будет выглядеть как локальное подключение к серверу БД. Хотя в таком варианте не производится никакого шифрования передаваемых данных, но ввиду того, что открытые данные передаются по защищенному туннелю, обеспечивается защита трафика. Незащищенными участками маршрута трафика будут участки от сервера БД до сервера службы тунеллирования и от клиента службы тунеллирования до клиента БД, но с этим не связано никаких дополнительных рисков, так как эти службы работают на одном компьютере.

10.5 Шифрование в MySQL

В MySQL поддерживается шифрование на уровне полей данных и прозрачное шифрование данных всей базы.

10.5.1 Шифрование на уровне данных

Для шифрования / расшифровывания данных на уровне полей в MySQL поддерживается следующий ряд функций.

Функции шифрования:

– `AES_ENCRYPT()` – использует алгоритм AES;

– `ASYMMETRIC_ENCRYPT()` – использует асимметричный ключ для шифрования данных.

Функции дешифрования:

– `AES_DECRYPT()` – использует алгоритм AES;

– `ASYMMETRIC_DECRYPT()` – использует асимметричный ключ для расшифровки данных.

Сервисные функции:

– `CREATE_ASYMMETRIC_PRIV_KEY()` – генерация закрытого ключа;

– `CREATE_ASYMMETRIC_PUB_KEY()` – генерация открытого ключа;

– `MD5()` – генерация MD-5 хеша;

– `SHA1()`, `SHA2()` – генерация SHA хешей.

Рассмотрим пример симметричного шифрования/дешифрования паролей пользователей. Данные о пользователях будут храниться в таблице `users`, которая содержит имя и пароль пользователя, причем пароль планируется хранить в зашифрованном виде, поэтому используется тип данных `BLOB` для хранения бинарных данных:

```
CREATE TABLE users (
    username VARCHAR(50) NOT NULL,
    password BLOB NOT NULL
);
```

Для шифрования строк используется функция `AES_ENCRYPT`, которая принимает строку и ключ шифрования. В данном случае, первым параметром является пароль пользователя, вторым – ключ шифрования:

```
INSERT INTO users (username, password)
VALUES
('admin', aes_encrypt('33NC!DEgtFc8g3!2', 'Strong
password')),
('joe', aes_encrypt('123', 'Strong password')),
('jane', aes_encrypt('qwerty', 'Strong pass-
word'));
```

Запрос на чтение данных из таблицы `users` напрямую вернёт зашифрованные пароли. Простое приведение типов для зашифро-

ванных данных вернет NULL, но можно посмотреть бинарные данные в виде HEX значений:

```
SELECT username, CAST(password AS CHAR), HEX(pass-
word)
FROM users;
```

Для доступа к текстовым паролям данные надо дешифровать, используя функцию AES_DECRYPT:

```
SELECT username, CAST(aes_decrypt(password,
'Strong password') AS CHAR) AS text_password
FROM users
WHERE username = 'jane';
```

При указании неверного ключа будет возвращено значение NULL.

10.5.2 Прозрачное шифрование данных

В MySQL также поддерживается возможность прозрачного шифрования всей базы данных. Реализовано AES-256 шифрование для табличных пространств InnoDB. Используется двухуровневая схема ключей шифрования, состоящая из главного ключа (мастер-ключа) шифрования и ключей табличного пространства:

– когда табличное пространство зашифровано, ключ табличного пространства шифруется мастер-ключом и хранится в заголовке табличного пространства;

– когда необходимо получить доступ к зашифрованным данным табличного пространства, InnoDB использует мастер-ключ шифрования для расшифровки ключа табличного пространства;

– ключ табличного пространства не меняется, а главный ключ шифрования может быть изменен при необходимости (например, при подозрении, что мастер-ключ скомпрометирован) – это называют «ротацией мастер-ключа».

Ротация мастер-ключа выполняется как атомарная операция на уровне экземпляра сервера. Каждый раз, когда мастер-ключ меняется, все ключи табличных пространств в экземпляре MySQL заново зашифровываются и сохраняются обратно в заголовки соответствующих табличных пространств. Шифрование с помощью нового мастер-ключа должно быть успешно завершено для всех ключей табличных пространств, если этот процесс прерывается из-за сбоя сервера, InnoDB откатывает операцию при перезагрузке сервера. При ротации мастер-ключа изменяется только главный ключ шифрования и заново зашифровываются ключи табличных пространств. Данные, хранимые в соответствующих табличных пространствах, не расшифровываются и повторно не зашифровываются, так как непосредственно ключ табличного пространства не меняется.

10.5.3 Защита соединений в MySQL

По умолчанию MySQL сервер всегда включает поддержку SSL-подключений, но при этом не требуется обязательного SSL-подключения для клиентов. Клиенты могут выбирать соединение с SSL или без него, при подключении это управляется параметром `ssl-mode`. Если необходимо указать серверу, что все подключения должны выполняться с использованием SSL, то для этого используется системная переменная `require_secure_transport`, которая по умолчанию имеет значение `OFF`.

Для генерации ключей и сертификатов используется утилита `mysql_ssl_rsa_setup`, которая поставляется вместе с MySQL. Сгенерированные сертификаты и ключи сохраняются в виде pem-файлов в директории данных MySQL. При каждом запуске сервер анализирует наличие ключей в папке данных и, если они найдены, то автоматически включает поддержку SSL-подключений.

10.6 Шифрование в PostgreSQL

10.6.1 Шифрование на уровне полей данных

Криптографические функции для PostgreSQL предоставляет модуль `pgcrypto`. Для работы `pgcrypto` требуется OpenSSL – модуль не будет установлен, если PostgreSQL был установлен без поддержки OpenSSL.

Модуль `pgcrypto` позволяет хранить в зашифрованном виде избранные поля. Это применимо, когда ценность представляют только некоторые данные. Используется та же схема, что и в MySQL: чтобы прочитать эти поля, клиент передаёт дешифрующий ключ, сервер расшифровывает данные и выдаёт их клиенту. Данный модуль поддерживает как функции шифрования на основе PGP, так и низкоуровневые функции шифрования.

PGP (Pretty Good Privacy) – библиотека функций, позволяющая выполнять операции шифрования и цифровой подписи сообщений, файлов и другой информации, представленной в электронном виде, в том числе обеспечить прозрачное шифрование данных на запоминающих устройствах, например, на жёстком диске.

Шифрование PGP осуществляется последовательно хешированием, сжатием данных, шифрованием с симметричным ключом, и, наконец, шифрованием с открытым ключом, причём каждый этап может осуществляться одним из нескольких поддерживаемых алгоритмов.

Симметричное шифрование производится с использованием одного из симметричных алгоритмов на сеансовом ключе. Сеансовый ключ генерируется с использованием криптостойкого генератора псевдослучайных чисел. Сеансовый ключ зашифровывается открытым ключом получателя с использованием алгоритмов RSA или Elgamal.

В модуле pgcrypto доступен следующий ряд функций для шифрования, дешифрования и хеширования.

Функции шифрования:

– pgp_sym_encrypt – шифрует данные симметричным ключом PGP;

– pgp_pub_encrypt – зашифровывает данные открытым ключом PGP.

Функции дешифрования:

– pgp_sym_decrypt – расшифровывает сообщение, зашифрованное симметричным ключом PGP;

– pgp_pub_decrypt – расшифровывает сообщение, зашифрованное открытым ключом. Принимает в качестве параметра закрытый ключ, соответствующий открытому ключу, применяющемуся при шифровании.

Сервисные функции:

– digest – вычисляет двоичный хеш данных, используя заданный алгоритм (md5, sha1, sha224, sha256, sha384 и sha512);

– crypt – выполняет хеширование паролей, используя заданный алгоритм (bf, md5, xdes, des). Для сохранения нового пароля необходимо вызвать gen_salt(), чтобы сгенерировать новое значение соли;

– gen_salt – вычисляет значение соли для функции crypt();

Функции crypt() и gen_salt() разработаны специально для хеширования паролей. Функция crypt() выполняет хеширование, а gen_salt() подготавливает параметры алгоритма для неё. Они используют случайное значение, называемое солью, чтобы у пользователей с одинаковыми паролями зашифрованные пароли оказывались разными.

Рассмотрим пример шифрования обычных записей симметричным ключом шифрования с использованием функции pgp_sym_encrypt:

```
CREATE TABLE users_data (
    username VARCHAR(50) PRIMARY KEY,
    data bytea );

INSERT INTO users_data (username, data)
VALUES
('admin', pgp_sym_encrypt('admin info', 'strong
key')),
('joe', pgp_sym_encrypt('joe info', 'strong key')),
('jane', pgp_sym_encrypt('jane info', 'strong
key'));
```

Для дешифрования данных, зашифрованных симметричных ключом, используется функция pgp_sym_decrypt:

```
SELECT username, pgp_sym_decrypt(data, 'strong
key')
FROM users_data WHERE username = 'jane';
```

При попытке расшифровки неправильным ключом в PostgreSQL, в отличие от MySQL, запрос завершится с ошибкой, а не возвратом пустых данных.

Второй пример – задача шифрования паролей. Для этого используется функция crypt, принимающая пароль и соль, полученную методом gen_salt:

```
CREATE TABLE users (
    username VARCHAR(50) PRIMARY KEY,
    password TEXT
);

INSERT INTO users (username, password)
```

```
VALUES
('admin',
crypt('33NC!DEgtFc8g3!2',gen_salt('md5'))),
('joe', crypt('qwerty', gen_salt('md5'))),
('jane', crypt('qwerty', gen_salt('md5')));
```

В примере запрос на чтение данных из таблицы users напрямую вернёт зашифрованные пароли, причем одинаковый пароль «qwerty» для пользователей joe и jane будет иметь различный шифротекст.

Для проверки подлинности можно выполнить следующий запрос:

```
SELECT username
FROM users
WHERE username = 'joe'
      AND password = crypt('qwerty', password);
```

10.6.2 Защита соединений в PostgreSQL

В PostgreSQL встроена поддержка SSL для шифрования трафика между клиентом и сервером. Для использования этой возможности необходимо, чтобы и на сервере, и на клиенте была установлена криптографическая библиотека OpenSSL и включена поддержка SSL в конфигурационном файле `postgresql.conf`.

Запущенный сервер будет принимать как обычные, так и SSL-подключения по одному порту TCP. По умолчанию клиент выбирает режим подключения сам, но для подключений, у которых в конфигурационном файле `pg_hba.conf` указан тип соединения `hostssl`, всегда будут устанавливаться SSL-подключения. Для генерации ключей и сертификатов используется библиотека OpenSSL.

10.7 Шифрование в MongoDB

Шифрование данных доступно в MongoDB Enterprise и не поддерживается в свободно распространяемой версии MongoDB Community Edition. Однако можно использовать версию Percona for MongoDB.

Процесс шифрования данных включает в себя следующие этапы:

- генерация мастер-ключа;
- генерация ключей для каждой базы данных;
- шифрование данных с помощью ключей базы данных;
- шифрование ключей базы данных с помощью мастер-ключа.

При использовании шифрования данных все файлы данных шифруются в файловой системе. Данные не шифруются только в памяти и во время передачи.

Ключи базы данных являются внутренними для сервера и записываются на диск только в зашифрованном формате. Мастер-ключ на диск не записывается.

Безопасное управление ключами шифрования является важнейшим требованием для шифрования хранилища. Только мастер-ключ является внешним по отношению к серверу (т. е. хранится отдельно от данных и ключей базы данных) и требует внешнего управления, остальные ключи могут храниться вместе с экземпляром MongoDB. Для управления мастер-ключом MongoDB поддерживает два варианта управления ключами:

- интеграция со сторонним устройством управления ключами через протокол совместного управлением ключами (Key Management Interoperability Protocol, KMIP);
- локальное управление ключами через файл ключа.

Как и СУБД MySQL и PostgreSQL, MongoDB поддерживает TLS/SSL для шифрования всего сетевого трафика MongoDB. TLS/SSL гарантирует, что сетевой трафик MongoDB доступен для чтения только предполагаемому клиенту. MongoDB использует библиотеки TLS/SSL операционных систем:

- Windows Secure Channel (Schannel);
- Linux/BSD OpenSSL;
- macOS Secure Transport.

11 SQL-ИНЪЕКЦИИ

11.1 Основные понятия

Внедрение SQL-кода (англ. SQL injection / SQLi) – один из распространённых способов взлома сайтов и программ, работающих с базами данных, основанный на внедрении в запрос произвольного SQL-кода [14].

Суть данных атак заключается во внедрении в пользовательский ввод произвольных SQL-команд, позволяющих изменить логику исполняемого SQL-запроса. Такая уязвимость возможна только в тех приложениях, где текст исполняемого запроса представляет собой конкатенацию SQL-операторов и данных пользовательского ввода. В основном SQL-инъекциями актуальны для web-приложений, однако настольные приложения также могут содержать уязвимости такого рода, если параметры запросов считываются из настроечных файлов или задаются пользователями.

В результате успешной атаки злоумышленник может выполнить произвольный запрос к базе данных с правами приложения, например, прочитать содержимое любых таблиц, удалить, изменить или добавить данные, а в ряде случаев получить возможность чтения и записи локальных файлов и выполнения произвольных команд на атакуемом сервере.

Рассмотрим простейший пример SQL-инъекции. Пусть есть таблица `users`, содержащая идентификатор пользователя, имя пользователя и его пароль. Аутентификация пользователей выполняется проверкой результата следующего запроса:

```
SELECT *
FROM users
WHERE username='username' AND password='password';
```

где имя пользователя 'username' и пароль 'password' вводятся пользователем. Так, например, такой запрос возможен в web-приложениях, когда пользователю необходимо заполнить поля логина и пароля.

Представим, что для прохождения процедуры аутентификации выполняется HTTP GET запрос следующего вида:

```
/auth?username=<user>&password=<pass>
```

Данный запрос принимает два параметра: username и password. Строковые значения этих параметров вводятся на стороне клиентского приложения. Так, например, для веб-приложения проверка аутентификации может иметь следующий вид:

```
$sql = "SELECT * FROM users WHERE
username='$username' AND password='$password'";
response = mysql_query($sql);
```

Здесь значения $username и $password – это значения, введенные пользователем в полях ввода веб-приложения для логина и пароля соответственно, которые пришли на сервер в качестве параметров запроса. Т.е. в данном случае была выполнена конкатенация SQL-запроса и значений присланных полей из HTTP-запроса.

Если пользователь введет корректные данные – имя и пароль – то будет сформирован корректный запрос авторизации, например HTTP запрос:

```
/auth?username=A.C.%20Иванов&password=1246buak
```

будет преобразован в SQL-запрос:

```
SELECT * FROM users
WHERE username='A.C. Иванов' AND pass-
word='1246buak';
```

Однако, пользователи не всегда вводят корректные данные. Если задать в поле ввода логина значение «admin» а в поле ввода

пароля – значение «' OR 'a' = 'a», то результирующий запрос примет следующий вид.

```
SELECT * FROM users
WHERE username='admin' AND password='' OR 'a'='a';
```

Присутствие тождественно истинного выражения 'a' = 'a', присоединенного через логическое «или», приведет к тому, что условие в предложении WHERE будет всегда истинно, запрос вернет непустой набор данных и, соответственно, аутентификация пользователя admin выполнится успешно.

Такого рода проблема заключается в отсутствии фильтрации пользовательского ввода.

11.2 Основные приёмы внедрения SQL кода

Можно выделить следующие основные приемы внедрения SQL-кода:

- внедрение заведомо истинного условия;
- экранирование (комментирование) хвоста запроса;
- генерация ошибок.

Заведомо истинное выражения можно внедрять как в числовой параметр, так и в строковый. Главный принцип основан на внедрении истинного выражения после условия WHERE. Пользователь может задать любое значения для проверки условия фильтрации и затем встроить заведомо истинное выражение.

Подход на основе экранирования части запроса позволяет исключить из предложения фильтрации одно или несколько условий. Например, на каком-то сайте пользователю необходимо ввести логин и пароль. Допустим, злоумышленник узнал имя пользователя, но не узнал его пароль, тогда в поле ввода логина можно ввести логин и закрыть его кавычкой, а затем закомментировать оставшуюся часть запроса, поставив символ комментирования --:

```
SELECT * FROM users
WHERE username = 'A.C. Иванов'
      -- AND password = 'password';
```

Соответствующая проверка пароля будет экранирована комментарием и выполняться не будет. Таким образом можно пройти аутентификацию даже без знания пароля. Данное действие возможно также благодаря истинному значению после условия `WHERE`.

Еще одним распространенным подходом является генерация ошибочных запросов с целью получения полезной информации о виде исходного запроса, структуре данных и даже о файловой системе. Все эти данные можно получить с помощью сообщений об ошибках. Например, выполнив следующий запрос, можно получить сообщение об ошибке, из которого можно узнать название используемой БД.

```
SELECT 1 AND ExtractValue(1,concat(0x5C,(select database())));
```

Результат запроса может иметь следующий вид:

```
Error Code: 1105. XPATH syntax error: '\coronavirus_statistics'
```

11.3 Типы SQL-инъекций

Существует большое разнообразие вариантов SQL инъекций, которые применимы в различных условиях для достижения различных целей, в зависимости от методологии их проведения можно выделить два типа SQL-инъекций:

– классические SQL-инъекции – атаки в условиях, когда приложение получает от сервера ответ, содержащий результат запроса, или сообщение об ошибке, содержащее интересующие данные;

– слепые SQL-инъекции – атаки в условиях, когда ответ сервера не содержит никакой релевантной информации, но может быть интерпретирован как логическое значение.

В зависимости от условий применяются различные методики получения данных: в первом случае это формирование запросов, непосредственно выводящих интересующие данные, во втором – получение от сервера ответов да/нет на интересующие запросы.

11.3.1 Классические SQL-инъекции

Классические SQL-инъекции наиболее просты в эксплуатации, поскольку ответ сервера содержит результат запроса, на который производится воздействие. В зависимости от того, содержатся ли данные в ответе сервера в явном виде либо косвенно (в составе сообщения об ошибке), атаки этого вида можно разделить на три подкласса:

– SQL-инъекции на основе объединения;
– SQL-инъекции на основе последовательных запросов;
– SQL-инъекции на основе возвращаемых ошибок.

11.3.1.1 SQL-инъекции на основе объединения (Union-based)

SQL-инъекции на основе объединения нацелены на получение дополнительных данных из определенной таблицы, используя возможность объединения результатов нескольких SQL-запросов. Проведение атаки с использованием этой техники предполагает применение оператора объединения `UNION`.

Принцип работы можно продемонстрировать на следующем примере. Допустим, есть запрос, возвращающий строковое значение названия книги, поиск которой осуществляется по её идентификатору (`id`). Для получения информации о книге используется следующий GET-запрос, принимающий идентификатор книги:

```
SELECT book.name FROM book WHERE book.id = $id;
```

HTTP GET-запрос получения данных имеет следующий вид:

```
/books?id=1
```

Если передать в качестве значения переменной `id` идентификатор существующей книги, то будет выведена информацию по этой книге. Если же введется несуществующий идентификатор, например, -1, то запрос примет вид:

```
SELECT book.name FROM book WHERE book.id = -1;
```

В таком случае запрос возвращает `NULL`.

Соответственно, чтобы выбрать что-либо «секретное» из БД, достаточно применить операцию `UNION SELECT` и значение `NULL` объединится с новой выборкой, в результате чего именно результат второго оператора выборки и будет возвращен. Например, можно попытаться узнать пароль пользователя `admin`, выполнив следующий HTTP-запрос:

```
/books?id=-1 UNION SELECT users.password FROM users
WHERE username = 'admin';
```

Соответствующий SQL-запрос будет иметь вид:

```
SELECT book.name FROM book WHERE book.id = -1
UNION
SELECT users.password FROM users
WHERE username = 'admin';
```

Для корректного запроса с оператором `UNION` должны быть выполнены два ключевых условия:

– отдельные запросы должны возвращать одинаковое количество столбцов;

– типы данных в каждом столбце должны быть совместимы между отдельными запросами.

Для проведения атаки такого типа обычно необходимо получить ответ на два вопроса:

– сколько столбцов возвращает исходный запроса;

– какие столбцы исходного запроса имеют подходящий тип данных для вывода результатов внедренного запроса.

Первый вопрос решается последовательными попытками сортировки по номеру колонки до тех пор, пока запрос выполняется без ошибки, например, вставка:

```
1 ORDER BY 5 --
```

Итоговый запрос:

```
SELECT * FROM book WHERE book.id = 1 ORDER BY 5 --
```

Как только указывается число (номер столбца), по которому следует сортировать, которое больше количества столбцов, то возникает ошибка, что данного столбца в таблице не существует. Тем самым можно узнать количество столбцов в таблице.

Для определения подходящего типа столбца (обычно это строковый столбец) для вывода данных при известном числе столбцов, также используют переборный подход, до тех пор, пока запрос `UNION SELECT` не выполнится без ошибки.

При использовании `UNION`, если типы объединяемых столбцов не совпадают, то выводится соответствующее сообщение об ошибке. Если типы совпадают, ошибки не возникает. Например, при выполнении запроса можно получить следующую ошибку:

```
SELECT * FROM book WHERE book.id = 1
UNION
SELECT 'a', NULL, NULL, NULL, NULL --
 ERROR: неверный синтаксис для типа integer: "a"
```

Данные примеры приведены для СУБД PostgreSQL, так как MySQL достаточно свободно приводит разные типа данных для вывода.

11.3.1.2 SQL-инъекции на основе последовательных запросов (Stacked queries-based)

SQL-инъекции на основе последовательных запросов возможны, когда сервер поддерживает последовательные запросы. Метод основан на том, что в уязвимый параметр добавляется символ «`;`» и непосредственно за ним указывается произвольный внедряемый SQL-запрос. Такой тип инъекций, как правило, используется для внедрения SQL-команд, отличных от `SELECT`, например, для манипуляции данными. В зависимости от сервера приложения, СУБД и прав доступа такие атаки потенциально могут дать возможности чтения/записи из файловой системы, а также выполнения команд в операционной системе.

Так, например, при исполнении запроса на выборку данных можно изменить пароль всем пользователям.

```
SELECT * FROM users WHERE username=-1;
UPDATE users SET password = 'YouShallNotPass!';
```

11.3.1.3 SQL-инъекция на основе ошибок (Error-based)

SQL-инъекции на основе ошибок применимы только в условиях, когда в приложении предусмотрен (или скорее не отключен) детальный вывод ошибок. Фактически в этом случае необходимо сформировать синтаксически некорректное выражение, при попытке исполнения которого сервер сгенерирует сообщение об ошибке, содержащее интересующие данные. Такой тип атаки позволяет получить данные даже используя запросы, которые по определению их не возвращают.

Генерация ошибок применялась при рассмотрении SQL-инъекций на основе объединения для получения количества столбцов и их типов. Например, запросом `SELECT a()` можно сгенерировать ошибку, в которой указана текущая БД – `coronavirus_statistics`.

```
SELECT a();
Error Code: 1305. FUNCTION coronavirus_statistics.a
does not exist
```

Следующие два запроса вызывают ошибку синтаксического анализа XML.

```
SELECT extractvalue(rand(),concat(0x3a,(
        SELECT version())))
Error Code: 1105. XPATH syntax error: ':8.0.26'
SELECT extractvalue(rand(),concat(0x3a,(
        SELECT username FROM users LIMIT 0,1)))
Error Code: 1105. XPATH syntax error: ':A.C.
Иванов'
```

В первом варианте можно получить версию сервера MySQL, а во втором – некоторые данные из таблицы, в данном случае – имя пользователя в таблице `users`.

Функция `ExtractValue` в MySQL выполняет запрос `XPath` к строке, представляющей данные XML. `XPath` – это язык запросов к элементам XML-документа. Функция `ExtractValue` принимает следующие входные данные: первым параметром указывается XML-документ, вторым параметром – `XPATH` запрос. Если запрос `XPath` синтаксически неверен, появляется сообщение об ошибке.

Таким образом, в некоторых случаях в качестве параметра можно передать `SELECT` запрос, содержащий вызов функции `XPATH`, и через сообщение об ошибке получить интересующую нас информацию.

11.3.2 Слепые SQL-инъекции

Слепые SQL-инъекции требуют определенных ухищрений для успешного проведения, поскольку ответ сервера не содержит результат того запроса, на который производится воздействие. В зависимости от того, меняется ли ответ сервера, возвращает ли запрос

какие-то данные или нет, отключен вывод ошибок или нет атаки этого вида можно разделить на три подкласса:

- SQL-инъекции на основе содержимого;
- SQL-инъекции на основе возвращаемых ошибок;
- SQL-инъекции на основе времени исполнения запроса.

11.3.2.1 SQL-инъекция на основе содержимого (Content-based)

SQL-инъекция на основе содержимого применима в условиях, когда ответ сервера содержит некоторые данные, и в зависимости от того, вернул запрос какой-то результат или нет, изменяется ответ сервера. Фактически решение задачи сводится к формированию запросов, которые возвращают непустой набор данных, когда проверяемая гипотеза истинна, и пустой набор данных, когда гипотеза – ложна.

На основе этого можно получать интересующие данные, например, попытаться узнать пароль пользователя `admin`, угадывая по одной букве. Так, например, сравнивая первый символ пароля с буквой `m`, можно понять, что пароль начинается с символа, ascii код которого меньше, чем код буквы `m`:

```
SELECT * FROM book WHERE book.id = 1 AND
EXISTS(SELECT * FROM users
       WHERE username = 'admin'
       AND SUBSTRING(password, 1, 1) < 'm')
```

Дальше можно провести обратную проверку, т.е. сравнить первый символ пароля с кодом символа `a`.

```
SELECT * FROM book WHERE book.id = 1 AND
EXISTS(SELECT * FROM users
       WHERE username = 'admin'
       AND SUBSTRING(password, 1, 1) > 'a')
```

В примере предполагалось, что первый символ пароля больше символа a, но запрос вернул пустую таблицу. Следовательно, предположение неверно. Значит стоит рассматривать символы кодом еще меньше. Например, заглавные буквы или цифры.

В результате, в зависимости от того, получен корректный ответ или нет, двоичный поиск можно продолжить, перебирая символы в прямом или обратном направлении.

После того, как первый символ будет угадан, можно переходить ко второму, далее к третьему и т.д. Хоть этот способ занимает довольно много времени, результат все же будет получен.

11.3.2.2 SQL-инъекция на основе ошибок (Error-based)

SQL-инъекция на основе ошибок применима в условиях, когда ответ сервера не меняется в зависимости от того, вернул запрос какой-то результат или нет, но предусмотрено информирование об ошибках. В такой ситуации единственным выходом будет инициировать SQL-ошибки в зависимости от проверяемого условия. Это подразумевает формирование запросов таким образом, чтобы он вызывал ошибку базы данных, если проверяемая гипотеза истинна, и отрабатывал без ошибок, когда гипотеза ложна.

Например, запрос должен возвращать данные о книге. Добавим некоторое условие в виде оператора CASE:

```
1 AND ((SELECT CASE WHEN 1=2 THEN 1/0 ELSE 1 END)
> 0);
```

Т.е. запрос вернет единицу, деленную на 0, если выполняется условие 1=2, иначе 1. В таком виде данное добавочное условие будет всегда возвращать TRUE, ошибки не возникнет:

```
SELECT * FROM book WHERE book.id = 1 AND
   ((SELECT CASE WHEN 1=2 THEN 1/0 ELSE 1 END) >
0);
```

Но если поменять условие равенства 1=2 на 1=1, то будет выведено сообщение о делении на 0. Поэтому, если заменить тождество на проверяемое условие, можно добиться желаемого поведения.

```
SELECT * FROM book WHERE book.id = 1 AND
    ((SELECT CASE WHEN 1=1 THEN 1/0 ELSE 1 END) >
0);
ERROR: деление на ноль
```

По аналогии с приведенным примером в разделе 11.3.2.1, можно получить пароль администратора, сравнивая код первого символа с разными буквами. Если предположение будет верное, то вернется сообщение об ошибке, иначе запрос вернет данные из таблицы. Далее по каждому символу можно искать пароль администратора:

```
SELECT * FROM book
WHERE book.id = 1 AND
  ((SELECT CASE WHEN EXISTS(SELECT * FROM us-
ers
        WHERE username = 'admin' AND
        SUBSTRING(password, 1, 1) < 'm')
    THEN 1/0 ELSE 1 END) > 0);
```

Также стоит отметить, что если настройки вывода ошибок позволяют выводить детали ошибки, то в тексте ошибки, также, как и в случае с классическими SQL-инъекциями, можно увидеть весьма полезные данные, что делает дальнейшую атаку уже не совсем «слепой».

11.3.2.3 SQL-инъекция на основе времени исполнения запроса (Time-based)

SQL-инъекции на основе времени исполнения запроса применимы в условиях, когда ответ сервера не зависит от результатов запроса и вывод ошибок также отключен. Поскольку сервер не дает

никакой ответной реакции на модификацию запроса, необходимо как-то заставить его сделать это. Для решения этой задачи применяется подход, заключающийся в отправке базе данных SQL-запросов, которые вынуждают базу данных ждать определенное количество времени, прежде чем ответить. Время ответа и будет интерпретироваться как логическое значение. Фактически задача сводится к формированию запроса таким образом, чтобы он вызывал задержку исполнения, если проверяемая гипотеза истинна, и отрабатывал без задержки, когда гипотеза – ложна.

Например, после ввода идентификатора книги, данные о которой возвращает запрос, можно внедрить условие через оператор `IF`. Если выражение истинно, то ответ на запрос будет дан с задержкой:

```
SELECT * FROM book WHERE book.id = 1;
SELECT IF(1=1, SLEEP(15), 0);
```

Следовательно, если вернуться к задаче поиска пароля администратора, то в проверяемое условие необходимо подставить предположение о нахождении сначала первого символа, потом второго, третьего и так далее:

```
SELECT * FROM book WHERE book.id = 1;
SELECT IF(EXISTS(SELECT * FROM users
WHERE username = 'admin'
AND SUBSTRING(password, 1, 1) < 'm'),
SLEEP(15), 0);
```

11.4 База данных INFORMATION_SCHEMA

`INFORMATION_SCHEMA` – это база данных, хранящаяся в каждом экземпляре MySQL, в которой содержится информация обо всех других базах данных, которые содержит сервер MySQL. Дан-

ная база данных содержит несколько представлений только для чтения. Используя базу данных INFORMATION_SCHEMA и хранящиеся в ней представления schemata, tables, columns, злоумышленник может определить структуру всех баз данных, хранящихся на сервере.

В следующем примере возвращается список всех БД на сервере:

```
SELECT schema_name FROM information_schema.schemata;
```

Для объединения возвращаемых данных в одну строку можно воспользоваться функцией GROUP_CONCAT:

```
SELECT GROUP_CONCAT(schema_name SEPARATOR ', ')
FROM information_schema.schemata;
```

После просмотра списка всех БД, можно выбрать конкретную БД и получить список всех ее таблиц. В следующем примере выводятся все имеющиеся таблицы в базе данных coronavirus_statistics:

```
SELECT table_schema, table_name
FROM information_schema.tables
WHERE table_schema = 'coronavirus_statistics';
```

После выбора таблицы можно получить названия всех ее столбцов:

```
SELECT column_name
FROM information_schema.columns
WHERE table_schema = 'coronavirus_statistics'
AND table_name ='graphs';
```

Таким образом, обращаясь к этой БД в SQL-инъекциях, злоумышленник может определить структуру всех баз данных, хранящихся на сервере.

11.5 Противодействие SQL-инъекциям

К методам противодействия SQL-инъекциям относятся:

– фильтрация пользовательского ввода;
– использование подготовленных выражений;
– настройка прав доступа к учетным записям;
– выдача клиентскому приложению «неинформативных» сообщений об ошибках.

11.5.1 Настройка прав доступа к учетным записям

Необходимо уделять внимание правам доступа учетных записей, которые используются клиентским приложением для подключения к базе данных – им должны предоставляться только минимально-необходимые права доступа. Это позволит ограничить действия, которые можно выполнить на сервере, и, соответственно, минимизировать ущерб в случае успешной атаки.

11.5.2 Выдача клиентскому приложению «неинформативных» сообщений об ошибках

Сообщения об ошибках являются дополнительным источником информации для злоумышленников. Несомненно, они нужны для отладки системы на стадии тестирования, но после запуска системы в промышленную эксплуатацию следует пересылать клиентскому приложению «неинформативные» сообщения об ошибках без каких-либо технических деталей. Все подробности о происходящих ошибках необходимо протоколировать в журналы на сервере для дальнейшего анализа.

11.5.3 Фильтрация пользовательского ввода

Самым очевидным решением противодействия SQL-инъекциям является фильтрация пользовательского ввода перед

формированием исполняемых запросов. Это можно сделать как на уровне сервера приложения, так и с использованием программно-аппаратного брандмауэра веб-приложений.

Например, для числовых значений можно использовать преобразование ввода пользователя к целевому типу данных, для строковых данных выполнять фильтрацию спецсимволов и ключевых слов или фильтрацию по регулярным выражениям и т.д.

Такой подход подразумевает либо использование готовых классов фильтрации данных, либо их самостоятельную разработку. Недостатком этого подхода можно назвать то, что существуют определенные методы, позволяющие обойти такого рода фильтры – так называемая обфускация SQL-инъекций.

Обфускация – это приведение исходного текста или исполняемого кода программы к виду, сохраняющему её функциональность, но затрудняющему анализ, понимание алгоритмов работы и модификацию при декомпиляции. В контексте SQL-инъекций этот термин предполагает маскировку спецсимволов и ключевых слов в тексте пользовательского ввода.

Для обходов различных известных фильтров существуют различные приёмы, например:

– замена логических операций `AND` и `OR` их символьными аналогами `&&` и `||`;

– использование кодов символов вместо их прямого написания;

– двойное кодирование символов, использование строковых функций; добавление SQL комментариев в текст пользовательского ввода;

– использование переменного регистра;

– вложенное дублирование ключевых слов.

При этом различные подходы к обфускации могут работать на одних СУБД и не работать на других, например, замена логических операций их символьными аналогами возможна только для MySQL.

Примеры обфускации SQL-инъекций представлены в таблице 11.1.

Таблица 11.1. Примеры обфускации

Отфильтрованная инъекция	**Пропущенная инъекция**
Обход фильтрации ключевого слова `OR`	
`1 or 1 = 1`	`1 || 1 = 1` (сработает в MySQL)
Обход фильтрации ключевого слова `OR` и символа `'`	
`1 or substr(user,1,1) = 'a'`	`1 || substr(user,1,1) = 0x61`
Обход фильтрации по регулярным выражениям в PHPIDS 0.6 (блокирует запросы, содержащие = , (или ', за которыми следует любая строка или целое число)	
`1 UNION SELECT 1, table_name FROM information_schema.tables WHERE table_name = 'users'`	`1 UNION SELECT 1, table_name FROM information_schema.tables WHERE table_name LIKE 0x7573657273`
Обход фильтрации ключевых слов с использованием SQL-комментариев	
`1 union select password from...`	`1 un/**/ion se/**/lect pass/**/word fr/**/om...`
Обход замены ключевых слов пустой строкой	
`1 union select...`	`1 UNunionION SEselectLECT...`

Как показано в таблице, обойти фильтрацию ключевого слова `OR` возможно заменой символьным аналогом. Обход фильтрации ключевого слова `OR` и символа верхней кавычки возможен заменой слова `OR` на символьный аналог, а строкового параметра – напрямую его кодом.

Обход фильтрации по регулярным выражениям доступен в системе PHPIDS 0.6, которая блокирует запросы, содержащие знак равенства, скобку или верхнюю кавычку, за которыми следует любая строка или целое число. Чтобы обойти такую фильтрацию, знак равенства можно заменить на оператор `LIKE`, строковый параметр можно представить в виде кода.

Обход фильтрации ключевых слов с использованием SQL комментариев позволяет добавлять в запрос пустые комментарии, которые при компиляции уберутся из тела запроса. Такой метод может позволить обойти анализаторы запросов, которые выполняют поиск ключевых слов.

Также, обойти замену ключевых слов можно пустой строкой. В большинстве реализаций фильтрации пользовательского ввода предобработка запроса осуществляется за один проход, за который все найденные ключевые слова просто заменяются на пустую строку. Тем самым, если ключевое слово будет разделено своим же повтором, то при обработке вставленное ключевое слово будет убрано, и задуманное ключевое слово примет корректный вид.

В рамках защиты, подобного рода приёмы вынуждают расширять списки запрещенных ключевых слов и ужесточать правила фильтрации, что в свою очередь может создавать проблемы для работы легальных пользователей и при этом не гарантирует полной безопасности. Как следствие, подход к фильтрации данных может быть сложным и ненадежным.

11.6 Подготовленные выражения

Более надежный метод защиты от внедрения SQL кода, чем фильтрация пользовательского ввода – это использование параметризованных запросов (также известных как подготовленные выражения) вместо конкатенации строк внутри запроса.

Подготовленные выражения – это функциональность SQL-баз данных и языков программирования, предназначенная для разделения данных запроса и собственно операторов выполняемого SQL-запроса.

При использовании подготовленных выражений есть ряд преимуществ:

– сокращение объема кода – один и тот же подготовленный запрос можно использовать многократно для разных данных;

– уменьшение времени выполнения – для подготовленных выражений нет необходимости выполнять синтаксический разбор при каждом исполнении, а это довольно затратная операция. Синтаксический разбор выражения производится один раз, на этапе его подготовки;

– встроенная защита от SQL-инъекций – является прямым следствием того, что синтаксический разбор производится на этапе подготовки.

К недостаткам этого подхода можно отнести следующие:

– подготовленные выражения могут использоваться только для параметризации значений в запросах выборки и модификации данных;

– подготовленные выражения не могут использоваться для параметризации таких значений, как имена таблиц или столбцов, параметров сортировки в предложении `ORDER BY`;

– однократные параметризованные запросы работают, как правило медленнее, чем обычные.

Если в рамках конкретной задачи эти ограничения являются непреодолимыми, то необходимо использовать рассмотренный выше подход на основе фильтрации.

SQL-инъекция происходит непосредственно на этапе синтаксического разбора выражения, наличие лишних кавычек, символов комментирования и прочего заставляет синтаксический анализатор формировать результирующий запрос не таким образом, как задумали разработчики, а таким образом, как необходимо злоумышленнику. В случае использования подготовленных выражений синтаксический разбор уже сделан и любые символы, переданные в параметрах, интерпретируются как часть данных, а не как часть SQL-выражения.

В MySQL для создания подготовленных запросов используется выражение PREPARE, которое разбирает SQL запрос и присваивает ему имя. Текст запроса должен представлять только один запрос. В пределах данного запроса символы «?» могут использоваться в качестве маркеров параметров для указания места использования данных в запросе. Маркеры параметров могут использоваться только там, где должны находиться данные, а не для ключевых слов SQL, идентификаторов и т.д. После подготовки SQL запроса с помощью PREPARE, выражение выполняется с помощью команды EXECUTE. Если подготовленный запрос содержит маркеры параметров, необходимо использовать выражение USING, в котором перечисляются пользовательские переменные, содержащие связанные с параметрами значения. Значения параметров могут быть представлены только пользовательскими переменными, в выражении USING должно быть указано такое количество переменных, которое соответствует числу маркеров параметров «?» в SQL запросе. После завершения работы с подготовленным запросом выполняется команда DEALLOCATE PREPARE для освобождения ресурсов данного запроса:

```
PREPARE stmt_name FROM
    'SELECT * FROM users
    WHERE users.username =?
    AND users.password =?';
SET @n = 'admin';
SET @p = '157erfdai@njdiQWrtbh';
EXECUTE stmt_name USING @n, @p;
DEALLOCATE PREPARE stmt_name;
```

В PostgreSQL подготовленные запросы реализуются с небольшими отличиями. При создании подготовленного запроса обращение к параметрам происходит по положению, с помощью модификаторов $1, $2 и т.д. Внедряемые пользователем значения

передаются в качестве аргументов через команду EXECUTE. Подготовленные запросы можно также очистить вручную с помощью команды DEALLOCATE:

```
PREPARE st_user AS
   SELECT * FROM users
   WHERE username=$1 AND password=$2;
EXECUTE st_user('admin','157erfdai@njdiQWrtbh');
DEALLOCATE PREPARE st_user;
```

Если же в качестве параметра username передать строку «admin», а в качестве параметра password строку «'' OR 1 = 1», то логика исполнения запроса не изменится, будет выполнен исходный запрос, который вернет пустой набор данных.

12 ЦЕЛОСТНОСТЬ ДАННЫХ

12.1 Основные понятия

В контексте информационной безопасности термин «целостность данных» подразумевает, что данные не были изменены при выполнении какой-либо операции над ними, будь то передача, хранение или отображение. Когда речь идет о СУБД, то в понятие **целостности** вкладывается соответствие имеющейся в базе данных информации её внутренней логике, структуре и всем явно заданным ограничениям. Ограничения должны быть формально описаны, после чего СУБД должна обеспечивать их выполнение, контролируя те операции модификации данных, которые могут нарушить эти ограничения, и запрещая те операции, которые их действительно нарушают.

Целостность БД не гарантирует достоверности (истинности) содержащейся в ней информации, но обеспечивает по крайней мере правдоподобность этой информации, отвергая заведомо невозможные значения.

Таким образом, не следует путать целостность (непротиворечивость) БД с истинностью (достоверностью) хранимых данных. Достоверность есть соответствие фактов, хранящихся в базе данных, реальному миру. Очевидно, что для определения достоверности БД требуется обладание полными знаниями как о содержимом БД, так и о реальном мире. Для определения целостности БД требуется лишь обладание знаниями о содержимом БД и о заданных для неё правилах. Поэтому СУБД не может гарантировать наличие в базе данных только истинных данных; все, что она может сделать – это гарантировать отсутствие каких-либо данных, вызывающих нарушение ограничений целостности (то есть гарантировать то, что

она не содержит каких-либо данных, не совместимых с этими ограничениями). Из того, что данные являются достоверными, следует, что они непротиворечивы (но не обратное), а из того, что данные противоречивы, следует, что они недостоверны (но не обратное).

Таким образом целостность БД только частично покрывает понятие целостности в смысле информационной безопасности. В этой связи обеспечение целостности данных можно рассматривать как комплексную проблему, которая подразумевает выполнение ряда принципов:

– определение ограничений целостности на уровне модели данных;

– модификация данных только посредством корректных транзакций;

– работа с данными только авторизованных пользователей. Т.е. изменение данных может осуществляться только авторизованными пользователями, имеющими определенные привилегии;

– применение принципа минимальны привилегий. Минимальность привилегий подразумевает, что пользователи (в конечном счете, субъекты) должны быть наделены теми и только теми привилегиями, которые минимально необходимы для выполнения тех или иных действий;

– управление передачей привилегий. Необходимо для эффективной работы всей политики безопасности. Если схема назначения привилегий неадекватно отражает организационную структуру предметной области или не позволяет администраторам безопасности гибко манипулировать ею для обеспечения эффективности производственной деятельности, защита начинает мешать нормальной работе и провоцирует попытки ее обойти;

– разграничение функциональных обязанностей. Подразумевает организацию работы с данными таким образом, что в каждой из ключевых стадий, составляющих единый критически важный с точки зрения целостности процесс, необходимо участие различных пользователей;

– аудит событий. Аудит произошедших событий (включая возможность восстановления полной картины происшедшего) является превентивной мерой в отношении потенциальных нарушителей и позволяет восстановить данные в случае их повреждения.

В отношении базы данных разделяют следующие типы ограничений целостности:

– целостность сущностей направлена на обеспечение внутреннего единства отдельной сущности и подразумевает, что каждая строка таблицы обязана быть уникальной, т.е. таблица должна иметь первичный ключ, и столбец или столбцы, выбранные в качестве первичного ключа, должны быть уникальными и не содержать значений `NULL`;

– целостность ссылок подразумевает что значение внешнего ключа может быть только в одном из двух состояний: либо значение внешнего ключа содержит значение, являющееся первичным ключом главной таблицы, либо содержит значение NULL. Во втором случае подразумевается, что между объектами, представленными в БД, нет связи либо она неизвестна;

– целостность домена подразумевает, что все столбцы в реляционной БД должны быть определены на определенном домене, т.е. на допустимом потенциальном множестве значений определенного типа;

– пользовательские правила целостности. К этой категории относится набор правил, указанных пользователем, которые не принадлежат к категориям целостности сущности, домена и

ссылочной целостности. Сюда можно отнести ограничения значения по умолчанию (`DEFAULT`), уникальности (`UNIQUE`), запрет `NULL` значений (`NOT NULL`), специальные условия на значения (`CHECK`).

12.2 Транзакции

Для поддержания целостности данных в том числе используется и механизм транзакций. Транзакция – это выполняемая от имени определенного пользователя или процесса последовательность операторов манипулирования данными, переводящая базу данных из одного целостного состояния в другое целостное состояние.

Транзакциям присущи следующие свойства, известные как ACID:

- atomicity – атомарность;
- consistency – согласованность;
- isolation – изолированность;
- durability – долговечность.

12.2.1 Свойства транзакций

Атомарность гарантирует, что никакая транзакция не будет зафиксирована в системе частично. Транзакция выполняется как атомарная операция – либо выполняется вся транзакция целиком, либо она целиком не выполняется. Транзакция не может быть выполнена частично: если в процессе выполнения транзакции возникает какой-то сбой, то все выполненные до этого момента изменения данных должны быть отменены. Транзакция считается успешно выполненной, если в процессе исполнения не возникло какой-либо аварийной ситуации и все проверки ограничений целостности дали положительный результат.

Следующее требование – **согласованность**. Транзакция переводит базу данных из одного согласованного (целостного) состояния в другое согласованное (целостное) состояние. Внутри транзакции согласованность базы данных может нарушаться. Согласованность достигается вследствие атомарности, так как завершенная транзакция гарантирует согласованность данных, а транзакция, нарушающая ограничения целостности или не завершенная вследствие сбоев, не изменяет данные.

При **изолированности** транзакции должны выполняться независимо одна от другой. Транзакции недоступны промежуточные результаты других параллельно исполняемых транзакций, а также её промежуточные результаты недоступны другим транзакциям. Другими словами, изолированная транзакция `A` может получить доступ к объекту, обрабатываемому транзакцией `B`, только после завершения `B`, и наоборот. Таким образом при одновременной работе двух пользователей с одними и теми же данными они не замечают друг друга, как если бы они работали с этими данными строго по очереди.

Четвертое свойство – **долговечность**. Если транзакция выполнена, то результаты ее работы должны сохраниться в базе данных и не могут быть утеряны или отменены ни при каких обстоятельствах. В случае успешного выполнения транзакции все изменения гарантировано фиксируются в постоянной памяти, даже если в следующий момент произойдет сбой системы. В случае неуспешного завершения транзакции по любой причине гарантировано отсутствие каких-либо связанных с ней изменений во внешней памяти.

Следует по возможности использовать небольшие транзакции, т.е. включающие как можно меньше команд и изменяющие минимум данных. Соблюдение этого требования позволит наиболее эффективным образом обеспечить одновременную работу с данными нескольких пользователей.

12.2.2 Блокировки

Повышение эффективности работы при использовании небольших транзакций связано с тем, что при выполнении транзакции сервер накладывает на данные блокировки.

Блокировкой называется временное ограничение на выполнение некоторых операций обработки данных. Блокировка может быть наложена как на отдельную строку таблицы, так и на всю базу данных. Транзакции и блокировки тесно связаны друг с другом: транзакции накладывают блокировки на данные, чтобы обеспечить выполнение требований ACID. Без использования блокировок несколько транзакций могли бы изменять одни и те же данные.

Блокировка представляет собой метод управления параллельными процессами, при котором объект БД не может быть модифицирован без ведома транзакции, т.е. происходит блокирование доступа к объекту со стороны других транзакций, чем исключается непредсказуемое изменение объекта. Различают два вида блокировки:

– блокировка записи – транзакция блокирует строки в таблицах таким образом, что запрос другой транзакции на блокировку этих строк будет отклонен;

– блокировка чтения – транзакция блокирует строки в таблицах таким образом, что запрос другой транзакции на блокировку записи этих строк будет отклонен, а на блокировку чтения – принят.

В СУБД используют протокол доступа к данным, позволяющий избежать проблемы параллелизма. Его суть заключается в следующем:

– транзакция, результатом действия которой является извлечение строки таблицы, обязана наложить блокировку чтения на эту строку;

– транзакция, предназначенная для модификации строки данных, накладывает на нее блокировку записи;

– если запрашиваемая блокировка на строку отклоняется из-за уже имеющейся блокировки, то транзакция переводится в режим ожидания до тех пор, пока блокировка не будет снята;

– блокировка записи сохраняется вплоть до конца выполнения транзакции.

Решение проблемы параллельной обработки БД заключается в том, что строки таблиц блокируются, а последующие транзакции, модифицирующие эти строки, отклоняются и переводятся в режим ожидания. Если в системе управления базами данных не реализованы механизмы блокирования, то при одновременном чтении и изменении одних и тех же данных несколькими пользователями могут возникнуть проблемы одновременного доступа.

12.2.3 Проблемы совместного доступа к данным

При параллельном выполнении транзакций возможны следующие проблемы:

– потерянное обновление (англ. lost update) – при одновременном изменении одного блока данных разными транзакциями теряются все изменения, кроме последнего. Т.е. в этом случае одна транзакция переписывает изменения, осуществленные другой транзакцией, в результате одно из изменений будет утеряно;

– «грязное» чтение (англ. dirty read) – чтение данных, добавленных или изменённых транзакцией, которая впоследствии не подтвердится (откатится). В этом случае незафиксированные изменения, осуществленные одной транзакцией, читаются (или обновляются) другой. В случае перезаписи этих промежуточных значений или отката первой транзакции незафиксирован-

ные изменения могут быть отменены, а прочитавшая их транзакция с этого момента станет работать с неверными данными;

– неповторяющееся чтение (англ. non-repeatable read) – при повторном чтении в рамках одной транзакции ранее прочитанные данные оказываются изменёнными. Возникает тогда, когда транзакция считывает из базы значение, после чего вторая транзакция обновляет это значение. Если в этот момент времени первая транзакция продолжает выполняться, то имеющиеся в ее распоряжении данные становятся неактуальными;

– фантомное чтение (англ. phantom reads) – одна транзакция в ходе своего выполнения несколько раз выбирает множество строк по одним и тем же критериям. Другая транзакция в интервалах между этими выборками добавляет строки или изменяет столбцы некоторых строк, используемых в критериях выборки первой транзакции, и успешно заканчивается. В результате получится, что одни и те же выборки в первой транзакции дают разные множества строк.

12.2.4 Уровни изоляции транзакций

Для решения перечисленных выше проблем в стандарте SQL-92 определены четыре уровня изоляции транзакций. Уровень изоляции транзакции определяет, могут ли другие (конкурирующие) транзакции вносить изменения в данные, измененные текущей транзакцией, а также может ли текущая транзакция видеть изменения, произведенные конкурирующими транзакциями, и наоборот. Каждый последующий уровень поддерживает требования предыдущего и налагает дополнительные ограничения.

`READ UNCOMMITTED` (чтение незафиксированных или «грязных» данных) – наименее защищенный уровень изоляции, при котором транзакции способны читать незафиксированные изменения, сделанные другими транзакциями. При этом возможно считывание

не только логически несогласованных данных, но и данных, изменения которых ещё не зафиксированы.

`READ COMMITTED` (чтение фиксированных данных) – исключается «грязное» чтение, транзакция увидит только изменения, зафиксированные другими транзакциями. Тем не менее, в процессе работы одной транзакции другая может быть успешно завершена и сделанные ею изменения зафиксированы. В итоге первая транзакция будет работать с другим набором данных. Большинство промышленных СУБД, в частности, Microsoft SQL Server, PostgreSQL и Oracle по умолчанию используют именно этот уровень изоляции.

`REPEATABLE READ` (повторяющееся чтение) – накладывает блокировки на обрабатываемые транзакцией строки и не допускает их изменения другими транзакциями. В результате транзакция видит только те строки, которые были зафиксированы на момент ее запуска. Однако другие транзакции могут вставлять новые строки, соответствующие условиям поиска инструкций, содержащихся в текущей транзакции. При повторном запуске инструкции текущей транзакцией будут извлечены новые строки, что приведёт к фантомному чтению. Это уровень изоляции по умолчанию в MySQL.

`SERIALIZABLE` (сериализуемость) – самый надежный уровень изоляции, полностью исключающий взаимное влияние транзакций. Только на этом уровне параллельные транзакции не подвержены эффекту «фантомного чтения».

Разные уровни изоляции транзакций выдают разные правила того, какие действия из соседних транзакций будут видны. Более строгие уровни изоляции допускают меньшее влияние транзакций друг на друга.

На разных уровнях изоляции транзакции допускаются различные конфликты в данных, показанные на рисунке 28: грязное чтение, неповторяющееся чтение, фантомное чтение и аномалии сериализации. Потерянное обновление не допускается ни на одном уровне изоляции.

Уровень изоляции	Потерянное обновление	«Грязное» чтение	Неповторяющееся чтение	Фантомное чтение	Аномалии сериализации
Read uncommited	Исключено	Возможно	Возможно	Возможно	Возможно
Read commited	Исключено	Исключено	Возможно	Возможно	Возможно
Repeatable read	Исключено	Исключено	Исключено	Возможно	Возможно
Serializable	Исключено	Исключено	Исключено	Исключено	Исключено

Рисунок 28 – Перечь конфликтов для разных уровней изоляции транзакций

Уровни изоляции транзакций определяют, какие проблемы совместного доступа к данным допустимы. Если рассматривать уровень изоляции `READ UNCOMMITTED`, то допустимо грязное чтение, неповторяющееся чтение, фантомное чтение и аномалии сериализации. Потерянное обновление исключено.

Стандарт определяет минимальные требования, но СУБД может исключить какие-то проблемы совместного доступа, главное – обеспечить необходимый минимум.

В PostgreSQL на уровне `READ UNCOMMITTED` невозможно грязное чтение, а на уровне `REPEATABLE READ` невозможно фантомное чтение, которые допускаются стандартом.

Стоит отметить, что уровень изоляции `SERIALIZABLE` наиболее ресурсоемкий, в этом случае практически полностью останавливается параллельная обработка данных, заставляя остальные транзакции простаивать в очереди. Чем более строгие требование предъявляются к обеспечению изолированности транзакций, тем больше шансов, что какие-то транзакции между собой будут конфликтовать и транзакция будет отменена.

В целом, для уменьшения вероятности конфликтов между транзакциями транзакции должны быть как можно короче, длинных транзакций стоит избегать.

Реализация изоляции транзакций достигается одним из двух способов: использование блокировок и хранение нескольких версий данных.

12.2.5 «Мертвые» блокировки

«Мертвые», или тупиковые, блокировки характерны для многопользовательских систем. «Мертвая» блокировка возникает, когда две транзакции блокируют два блока данных и для завершения любой из них нужен доступ к данным, заблокированным ранее другой транзакцией. Для завершения каждой транзакции необходимо дождаться, пока блокированная другой транзакцией часть данных будет разблокирована. Но это невозможно, так как вторая транзакция ожидает разблокирования ресурсов, используемых первой.

Без применения специальных механизмов обнаружения и снятия «мертвых» блокировок нормальная работа транзакций будет нарушена. Для этих целей сервер снимает одну из блокировок, вызвавших конфликт, и откатывает инициализировавшую ее транзакцию.

Полностью избежать возникновения «мертвых» блокировок нельзя. Хотя сервер и имеет эффективные механизмы снятия таких блокировок, все же при написании приложений следует учитывать вероятность их возникновения и предпринимать все возможные действия для предупреждения этого. «Мертвые» блокировки могут существенно снизить производительность, поскольку системе требуется достаточно много времени для их обнаружения, отката транзакции и повторного ее выполнения.

Для минимизации возможности образования «мертвых» блокировок при разработке кода транзакции следует придерживаться следующих правил:

- выполнять действия по обработке данных в определенном порядке, чтобы не создавать условия для захвата одних и тех же данных;
- минимизировать длительность транзакций;
- применять как можно более низкий из допустимых уровней изоляции.

12.2.6 MVCC

Второй подход для обеспечения изолированности транзакций – это использование многоверсионности. Управление параллельным доступом посредством многоверсионности MVCC (англ. MultiVersion Concurrency Control) – один из механизмов СУБД для обеспечения параллельного доступа к базам данных, заключающийся в предоставлении каждому пользователю так называемого «снимка» или среза базы, обладающего тем свойством, что вносимые пользователем изменения невидимы другим пользователям до момента фиксации транзакции.

Это означает, что каждый SQL-оператор получает снимок данных на определённый момент времени вне зависимости от текущего состояния данных. Это защищает операторы от несогласованности данных, возможной, если другие конкурирующие транзакции внесут изменения в те же строки данных, и обеспечивает изоляцию транзакций для каждого сеанса баз данных. MVCC, отходя от методик блокирования, снижает уровень конфликтов блокировок и, таким образом, обеспечивает более высокую производительность в многопользовательской среде.

Основное преимущество использования модели MVCC по сравнению с блокированием заключается в том, что блокировки MVCC, полученные для чтения данных, не конфликтуют с блокировками, полученными для записи, и поэтому чтение никогда не мешает записи, а запись чтению.

При таком подходе читающая транзакция никогда не будет заблокирована другими транзакциями, читающими или изменяющими те же данные – каждая из них будет независимо работать со своей версией. Блокироваться будут только попытки изменить данные, которые уже изменены другой транзакцией, но еще не зафиксированы.

Во всех базовых СУБД используются свои реализации модели MVCC, которые отличаются механизмами формирования версионных меток данных (это могут быть идентификаторы транзакций или временные метки) и способом хранения многих версий данных (в базовой таблице, в журнале транзакций, в служебных таблицах), но реализуют описанную концепцию.

12.3 Защита данных встроенными средствами СУБД

Далее рассмотрим обеспечение защиты данных, в т.ч. обеспечение целостности, встроенными средствами СУБД: представлениями, хранимыми процедурами, функциями и триггерами.

Представления – это виртуальные таблицы, хранимые процедуры и триггеры – фрагменты кода на высокоуровневом языке программирования, являющем собой расширение языка SQL, хранящееся и выполняющиеся на сервере. Реализация всех правил «бизнес логики» с помощью указанных объектов БД предоставляет значительно более высокий уровень защиты данных, нежели их реализация в клиентских программах.

12.3.1 Представления

Представления – это временные (или виртуальные) таблицы, объекты баз данных, информация в которых не хранится постоянно, как в базовых таблицах, а формируется динамически при обращении к ним. По сути, представление – это именованный запрос, хранящийся в базе данных.

Представление не требует для своего хранения дисковой памяти, за исключением памяти, необходимой для хранения определения самого представления. Представление не может существовать само по себе, а определяется только в терминах одной или нескольких таблиц.

Применение представлений позволяет разработчику базы данных создать интерфейс приложения, не зависящий от реально существующих таблиц, а также позволяет предоставить каждому пользователю или группе пользователей ограниченный набор данных, скрывая поля и записи с конфиденциальной информацией. Представления позволяют ограничить доступ к данным на уровне записей, дополняя, таким образом, дискреционную модель разграничения доступа.

Разрешение на доступ к представлению должно быть явно предоставлено или отозвано, независимо от прав доступа к базовым таблицам представления, тем самым реализуется принцип минимальных привилегий. Данные в базовой таблице, не включенные в представление, скрыты от пользователей, которые имеют права доступа к представлению, но не к базовой таблице.

Определяя различные представления и выборочно давая на них права доступа, пользователь (или любая комбинация пользователей) может быть ограничен различными подмножествами данных. С помощью представлений можно ограничить доступ к данным и выдавать только часть:

– подмножество строк базовой таблицы (подмножество, зависящее от значений). Например, можно определить представление, которое содержит информацию о сотрудниках определенного отдела, и при этом скрыть информацию о сотрудниках из других отделов от некоторых пользователей;

– подмножество столбцов базовой таблицы (подмножество, не зависящее от значений). Например, можно определить представление, которое содержит все столбцы таблицы сотрудников, за исключением столбцов с информацией о паспорте или заработной платы сотрудника, так как эта информация является конфиденциальной;

– подмножество строк и столбцов базовой таблицы;

– подмножество столбцов, являющихся соединением более чем одной базовой таблицы. Например, можно определить представление, которое образовано на соединении таблиц «сотрудники» и «проекты», которое не предоставляет личных данных о сотрудниках и финансовую информацию о проектах;

– статистическая сводка данных по базовой таблице. Например, можно определить представление, содержащее только среднюю заработную плату сотрудников по каждому отделу;

– подмножество данных другого представления или совокупности некоторых представлений и базовых таблиц.

Можно выделить следующие преимущества использования представлений. Первое преимущество – это повышение защищенности данных. Права доступа к данным могут быть предоставлены исключительно через ограниченный набор представлений, содержащих только то подмножество данных, которое необходимо пользователю. Это позволяет назначить права на отдельные строки таблицы или получить на сами данные, а результат каких-то действий над ними. Подобный подход позволяет существенно ужесточить контроль доступа отдельных категорий пользователей к информации в базе данных.

Второе преимущество – обеспечение целостности данных. Если в операторе `CREATE VIEW` будет указана фраза `WITH CHECK OPTION`, то СУБД станет осуществлять контроль за тем, чтобы в исходные таблицы базы данных не была введена ни одна из строк, не удовлетворяющих предложению `WHERE` в определяющем запросе. Этот механизм гарантирует целостность данных в представлении.

Третье преимущество – разделение логики хранения данных и программного обеспечения. Можно менять структуру данных, не затрагивая программный код. При изменении схемы БД достаточно создать представления, аналогичные таблицам, к которым раньше

обращались приложения. Это удобно, когда нет возможности изменить программный код или к одной базе данных обращаются несколько приложений с различными требованиями к структуре данных.

Четвертое преимущество – ограничение доступа к данным. Можно предоставить пользователям упрощенную модель только необходимых ему данных.

Кроме того, использование представлений позволяет упростить структуру сложных запросов, использовать оптимизацию или предварительную компиляцию запросов.

Из недостатков можно отменить ограниченные возможности обновления. В большинстве случаев представления не позволяют вносить изменения в содержащиеся в них данные.

12.3.2 Хранимые процедуры и функции

Хранимые процедуры и функции – это объекты базы данных, представляющие собой набор SQL-инструкций, который компилируется и хранится как самостоятельный исполняемый код в системном каталоге базы данных. При этом одна процедура может быть использована в любом количестве клиентских приложений, что позволяет существенно сэкономить трудозатраты на создание прикладного программного обеспечения и эффективно применять стратегию повторного использования кода.

С точки зрения безопасности использование хранимых процедур позволяет определить интерфейс взаимодействия с данными строго определенным образом. Возможно разрешить доступ к данным только через процедуры и функции, не давая доступ непосредственно к таблицам или представлениям, с которыми эти процедуры взаимодействуют.

Например, можно предоставить пользователям доступ к процедуре, которая обновляет таблицу, но не давать им доступ к самой

таблице. Когда пользователь вызывает процедуру, она выполняется с привилегиями владельца процедуры. Пользователи, обладающие только привилегиями на выполнение процедуры (но не привилегиями на запросы обновление или удаление непосредственно из таблиц), могут вызвать процедуру, но не могут манипулировать данными таблицы каким-либо другим способом.

Фактически все прямые операции выборки и модификации данных могут быть запрещены и единственным вариантом их выполнения могут быть соответствующие процедуры или функции. Так, например, если пользователю доступна только хранимая процедура удаления одной записи по идентификатору, то у него нет возможности выполнить запрос, чтобы удалить все записи в таблице одним оператором.

Кроме того, если операции с данными требуют некоторой дополнительной логики, эти проверки также проще реализовать в коде хранимой процедуры, чем другими способами. Например, если продажи на сумму более 5000 рублей могут выполнять только продавцы с рейтингом не ниже 85, то проверку этих нетривиальных ограничений достаточно просто реализовать в хранимой процедуре, а в случае нарушения сгенерировать исключение.

Также при использовании хранимых процедур возможна реализация простой модели аудита. В этом случае при вызове той или иной процедуры факт вызова, пользователь и время могут фиксироваться в некоторой таблице аудита.

Использование хранимых процедур и функций, как и использование представлений, позволяет повысить защищенность данных. Доступ к данным большинству пользователей может быть предоставлен исключительно через ограниченный набор хранимых процедур, прямой доступ к таблицам предоставляется исключительно узкому кругу пользователей (администраторов БД). Это серьезно

снижает риск возникновения инцидентов. Также хранимые процедуры могут поддерживать дополнительную логику при выполнении операций над данными, связанную с проверкой нетривиальных ограничений или формированием аудиторского следа.

Также следует отметить повышение производительности: хранимые процедуры хранятся в скомпилированном и оптимизированном виде. Как следствие выполнение хранимой процедуры происходит быстрее, чем запуск аналогичного кода динамического SQL.

Кроме того, использование хранимых процедур позволяет снизить объем передаваемых данных: для вызова хранимой процедуры достаточно указать ее имя и значения параметров, а не передавать полный текст запроса.

Наконец, хранимые процедуры, выполняющие однотипные действия, могут переноситься между базами данных с незначительной модификацией, что позволяет повторно использовать код процедур.

12.3.3 Триггеры

Триггер представляет собой специальный тип хранимых процедур, запускаемых сервером автоматически при попытке изменения данных в таблице, с которой триггер связан. Можно сказать, что триггер – это скомпилированная SQL-процедура, исполнение которой обусловлено наступлением определенных событий внутри реляционной базы данных.

Триггеры в основном предназначены для обеспечения целостности и непротиворечивости данных, а также для отката транзакций:

– триггер гарантированно срабатывает только при наступлении определенного события, обычно связанного с модификацией значений в строке таблицы;

– триггер проверяет условия выполнения операции, вызвавшей его срабатывание;

– если условия верны, то триггер выполняет определенные действия (например, разрешает добавить в таблицу новую строку), а если условия ложны – триггер отвергает операцию.

Право на создание триггера имеет только владелец базы данных. Это ограничение позволяет избежать случайного изменения структуры таблиц, способов связи с ними других объектов и т.п.

Каждый триггер привязывается к конкретной таблице и выполняется в составе соответствующей транзакции модификации данных. В случае обнаружения ошибки или нарушения целостности данных в коде триггера можно произвести откат транзакции. Тем самым внесение изменений (событие, вызвавшее триггер) отменяется. Отменяются также все изменения, уже сделанные триггером. В отличие от обычной процедуры, триггер выполняется неявно в каждом случае возникновения триггерного события.

С точки зрения безопасности данных триггеры в первую очередь стоит рассматривать как дополнительный инструмент обеспечения целостности данных. Их используют, когда ограничения целостности и значений по умолчанию не позволяют добиться нужного уровня функциональности. Часто требуется реализовать сложные алгоритмы проверки данных, отслеживать изменения значений таблицы, чтобы нужным образом изменить связанные данные. Кроме того, триггеры могут использоваться для формирования аудиторского следа, например, при операциях модификации данных записывать в таблицу аудита пользователя, который выполнил изменения, и время изменения.

В ситуациях, когда представления не являются изменяемыми, триггеры для этих представлений могут использоваться для реализации модификации данных.

Применение триггеров большей частью весьма удобно для пользователей базы данных. Однако их использование часто связано с дополнительными затратами ресурсов на операции ввода/вывода. В том случае, когда тех же результатов можно добиться с помощью хранимых процедур или прикладных программ, применение триггеров нецелесообразно.

Можно отметить следующие недостатки использования триггеров:

– скрытая функциональность: перенос части функций в базу данных и сохранение их в виде одного или нескольких триггеров иногда приводит к сокрытию от пользователя некоторых функциональных возможностей. Хотя это в определенной степени упрощает работу, но может стать причиной незапланированных, потенциально нежелательных и вредных побочных эффектов, поскольку в этом случае пользователь не в состоянии контролировать все процессы, происходящие в базе данных;

– влияние на производительность: перед выполнением каждой команды по изменению состояния базы данных СУБД должна проверить условие триггера с целью выяснения необходимости запуска триггера для этой команды. Выполнение подобных вычислений сказывается на общей производительности СУБД, а в моменты пиковой нагрузки ее снижение может стать особенно заметным;

– неправильно написанные триггеры могут привести к серьезным проблемам, таким, например, как появление «мертвых» блокировок. Триггеры способны длительное время блокировать множество ресурсов, поэтому следует обратить особое внимание на сведение к минимуму конфликтов доступа.

ЗАКЛЮЧЕНИЕ

В данном учебном пособии рассмотрены основы безопасности систем баз данных, которые являются неотъемлемой частью обеспечения защиты данных в современном информационном обществе. Были изучены различные угрозы и ключевые принципы обеспечения безопасности.

В рамках данного пособия были последовательно рассмотрены основные вопросы, связанные с обеспечением конфиденциальности, целостности и доступности информации. В частности, в первой части пособия были рассмотрены вопросы аутентификации пользователей и управления доступом к данным путем назначения прав и ролей пользователей. Далее были рассмотрены основные способы обеспечения доступности данных: настройка резервного копирования данных, репликация, с также секционирование и сегментирование данных. В завершающей части пособия были рассмотрены вопросы аудита и мониторинга баз данных, шифрование и обеспечение целостности данных. Отдельная глава пособия посвящена одной из наиболее распространенных атак на базы данных – внедрение SQL кода (SQL-инъекции). Практическая реализация методов обеспечения безопасности баз данных рассмотрена на примерах СУБД MySQL, PostgreSQL и MongoDB.

Следует отметить, что ключевым компонентом в обеспечении безопасности является необходимость соблюдения комплексного подхода к обеспечению безопасности систем баз данных. Это включает как технические аспекты, такие как аутентификация, авторизация, шифрование и контроль доступа, так и организационные меры, такие как политики безопасности, обучение персонала и управление учетными записями.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Осипов, Д.Л. Технологии проектирования баз данных / Д.Л. Осипов. – М.: ДМК Пресс, 2019. – 498 с.
2. Новиков, Б.А. Основы технологий баз данных / Б.А. Новиков, Е.А. Горшкова, Н.Г. Графеева. – М.: ДМК Пресс, 2020. – 582 с.
3. Diaz, C. Database security / C. Diaz. – Duxbury: Mercury Learning and Information, 2022. – 350 с.
4. Кэмпбелл, Л. Базы данных. Инжиниринг надежности / Л. Кэмпбелл, Ч. Мейджорс. – СПб.: Питер, 2020. – 304 с.
5. Волк, В.К. Базы данных. Проектирование, программирование, управление и администрирование / В.К. Волк. – СПб.: Лань, 2020. – 244 с.
6. Шварц, Б. MySQL по максимуму / Б. Шварц, П. Зайцев, В. Ткаченко. – СПб.: Питер, 2018. – 864 с.
7. MySQL: MySQL 8.0 Reference Manual [сайт]. – 2023. – URL: https://dev.mysql.com/doc/refman/8.0/en/ (дата обращения: 30.01.2023).
8. Джуба, С. Изучаем PostgreSQL 10 / С. Джуба, А. Волков. – М.: ДМК Пресс, 2019. – 400 с.
9. Документация к Postgres Pro Standard 15 [сайт]. – 2023 – URL: https://postgrespro.ru/docs/postgrespro/15 (дата обращения: 30.01.2023).
10. Брэдшоу, Ш. MongoDB: полное руководство. Мощная и масштабируемая система управления базами данных / Ш. Брэдшоу, Й. Брэзил, К. Ходоров. – М.: ДМК Пресс, 2020. – 540 с.
11. MongoDB Documentation [сайт]. – 2023. – URL: https://www.mongodb.com/docs/ (дата обращения: 30.01.2023).

12. Ёсу, М.Т. Принципы организации распределенных баз данных / М.Т. Ёсу, П. Вальдуриес. – М.: ДМК Пресс, 2021. – 672 с.
13. PGStats [сайт]. – 2023. – URL: https://pgstats.dev/ (дата обращения: 31.01.2023).
14. Хоффман, Э. Безопасность веб-приложений / Э. Хоффман. – СПб.: Питер, 2021. – 336 с.

Учебное издание

Агафонов Антон Александрович,
Юмаганов Александр Сергеевич

БЕЗОПАСНОСТЬ СИСТЕМ БАЗ ДАННЫХ

Учебное пособие

Редакционно-издательская обработка А.В. Ярославцевой

Подписано в печать 19.06.2023. Формат 60×84 1/16.
Бумага офсетная. Печать офсетная. Печ. л. 17,0.
Тираж 27 экз. Заказ № .

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА»
(САМАРСКИЙ УНИВЕРСИТЕТ)
443086, САМАРА, МОСКОВСКОЕ ШОССЕ, 34.

Издательство Самарского университета.
443086, Самара, Московское шоссе, 34.