СОДЕРЖАНИЕ ЛЕКЦИИ 2

Индексы. Объединения. Метаданные в базах данных. Компилятор запросов. Оптимизатор запросов. Статистика исполнения запросов. Диспетчеры: диспетчер клиентов, диспетчер запросов, диспетчер транзакций. Базы данных и XML.

Примеры Web-приложений, использующих базы данных. Использование специальных инструментальных средств разработки Web-приложений. Архитектуры web-приложений с базами данных. Способы интеграции баз данных в среду Web.

Индексы

Индексы в базах данных представляют собой структуры данных, создаваемые для ускорения операций поиска, сортировки и фильтрации данных по одному или сразу нескольким столбцам. Эти структуры СУБД хранит отдельно от основной таблицы.

Рассмотрим пример индексации по столбцу Title (название книги) на основе сортировки по алфавиту. Индекс по названию книги представляет собой структуру данных, которая содержит пары ключ-значение. В этом индексе ключом является название книги, а значением – индекс. В качестве индекса выступает ссылка (ID_Book – первичный ключ) на соответствующую запись в таблице Book. В реальной базе данных такой индекс можно создать с помощью SQL-команды CREATE INDEX:

```
CREATE INDEX book_title_index ON Book (Title)
```

, где book_title_index – это название индекса, Book – название таблицы с книгами, а Title – поле с названием книги.

Таблица Book

ID_Book	Title	…	...	...
1	Война и мир			
2	Идиот			
3	Мать			
4	1984			
5	Щелкунчик			
6	Золушка			
7	Старик и море			
8	Каштанка			
9	Алхимик			
10	Анна Каренина			

Индекс

Ключ	Значение
1984	4
Алхимик	9
Анна Каренина	10
Война и мир	1
Золушка	6
Идиот	2
Каштанка	8
Мать	3
Старик и море	7
Щелкунчик	5

Когда выполняется запрос на поиск книги по названию, СУБД вначале достаточно быстро отыскивает требуемый ключ в отсортированном столбце созданной структуры «ключ-значение», извлекает из второго столбца соответствующий индекс – номер записи в таблице, и использует его для быстрого определения местоположения записи (или записей, если таких названий несколько) в основной таблице Book. Обычный поиск может потребовать просмотра каждой записи в базе данных для сопоставления с введенным названием книги. Этот процесс может быть медленным, особенно при большом объеме данных, так как СУБД обязана просматривать каждую запись в исходной таблице. А использование оптимизированных алгоритмов поиска на отсортированной структуре позволяет быстро найти нужные данные.

Если книг в базе данных очень много, то построение индекса по отсортированному полю может стать сложной задачей из-за большого объема данных. Чтобы эффективно организовать и индексировать большие объемы данных СУБД строит индексы на специализированных структурах данных, такие как B-деревья (B-tree) или хэш-таблицы (HASH-TABLE).

B-деревья названы в честь их создателя Р. Байера (R. Bayer), немецкого информатика и профессора, который разработал эту структуру данных в 1972 году. B-деревья являются одной из наиболее широко используемых структур данных в базах данных эффективного хранения и поиска. Как и обычные двоичные деревья (Binary tree), B-деревья представляют собой структуру данных, которая сама по себе обеспечивает отсортированность ключей в индексе и может быть построена как на отсортированных, так и на неотсортированных данных. В отличие от обычного дерева, в этой более сложной структуре каждый узел может содержать от одного до N ключей, что обеспечивает эффективное выполнение операций вставки без перестройки всей структуры. Однако операции удаления элементов из B-дерева могут потребовать дополнительных ресурсов и времени на перестройку его структуры. В примере, приведенном на рисунке 7, показан

процесс построения B-дерева на неотсортированном столбце Title таблицы Book с числом ключей в узле не более 3-х, с числом узлов на уровне не более 4-х. Видно, что при каждой операции INSERT дерево остается сбалансированным.

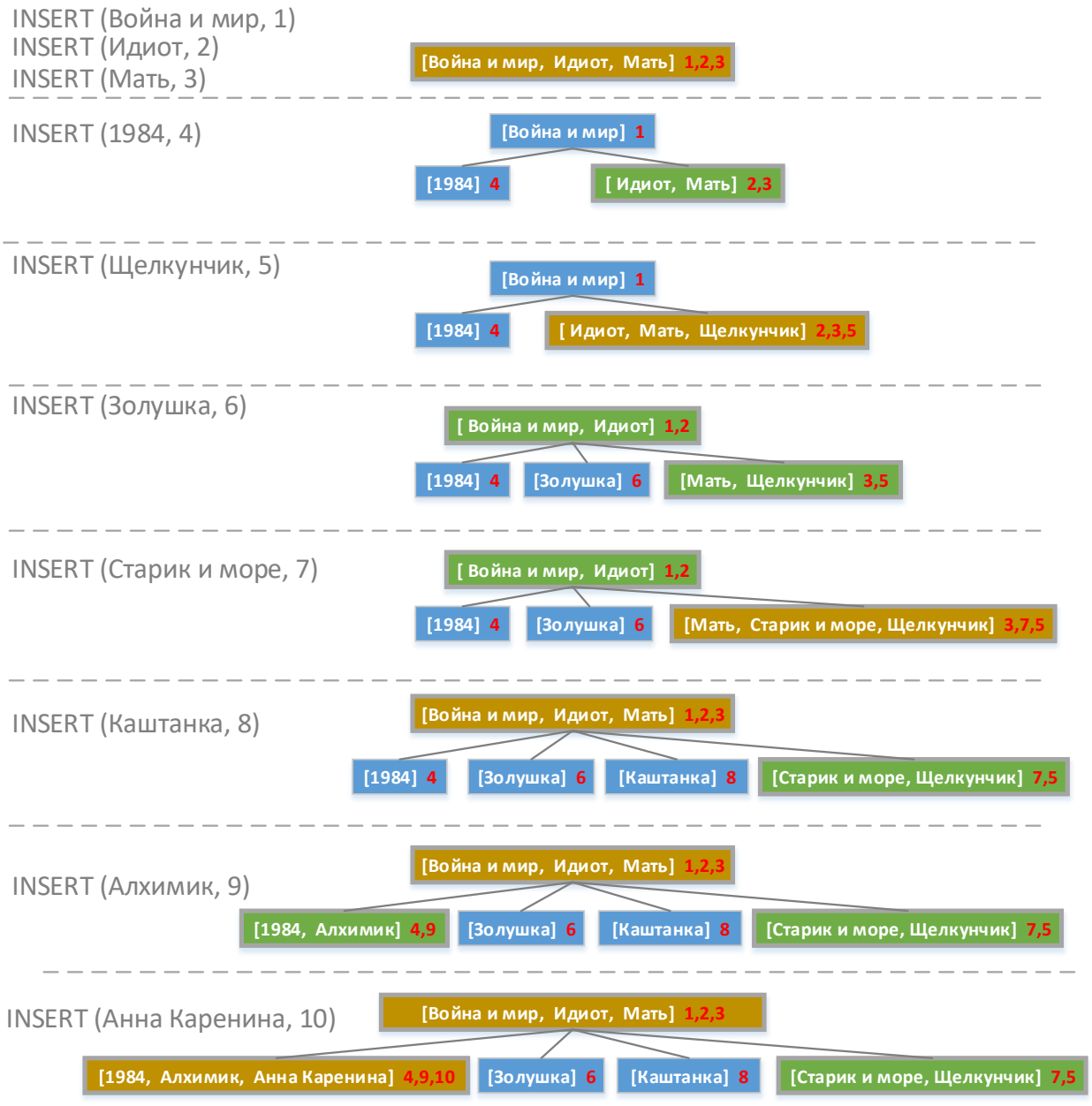

Рисунок 7 – Пример построения B-дерева

При работе с большими объемами данных, когда требуется быстрый доступ к группам записей, а не к отдельным элементам, весьма полезна так называемая пакетная индексация. Пакетная индексация – это метод, при котором несколько элементов индексируются одновременно как единое целое. B-дерево хорошо подходит для пакетной индексации. Вместо того чтобы индексировать каждый элемент столбца отдельно, их группируют в пакеты и индексируют как единое целое. Например, можно создать индекс на основе первой буквы названия книги или на основе группы близких по алфавиту книг. Это позволяет снизить накладные расходы СУБД на создание и обслуживание индексов, так как операции индексации на меньших по объёму структурах выполняются более эффективно. Пакетная индексация может существенно повысить производительность запросов, основанных на критериях группировки данных, таких как запросы на вычисление суммарных значений, средних показателей, общего количества для каждой группы записей в больших таблицах данных.

Хэш-таблицы (англ. hash-table) – это структуры данных, используемые для реализации ассоциативных массивов или словарей, где данные хранятся в виде пар ключ-значение. Они основаны на функции хеширования, которая выполняет алгебраические и логические преобразования ключей в целочисленные индексы массива, где фактически будут храниться данные. Это позволяет мгновенно находить значения по ключу, независимо от объема данных. Хэш-таблицы были придуманы и впервые описаны в 1953 году американским математиком и информатиком А. Шёнбергом (Arnold Schonberg). Он предложил использовать хэш-функции для создания эффективной структуры данных, которая позволяла бы быстро находить элементы по ключу. Даже при увеличении размера набора данных время доступа в такой структуре остается постоянным и быстрым. Иллюстрация формирования хэш-таблицы представлена на рисунке 8. Функция

хэширования преобразует ключ – строку с названием книги – в целое число, а СУБД помещает это число и ID ключа в хэш-таблицу.

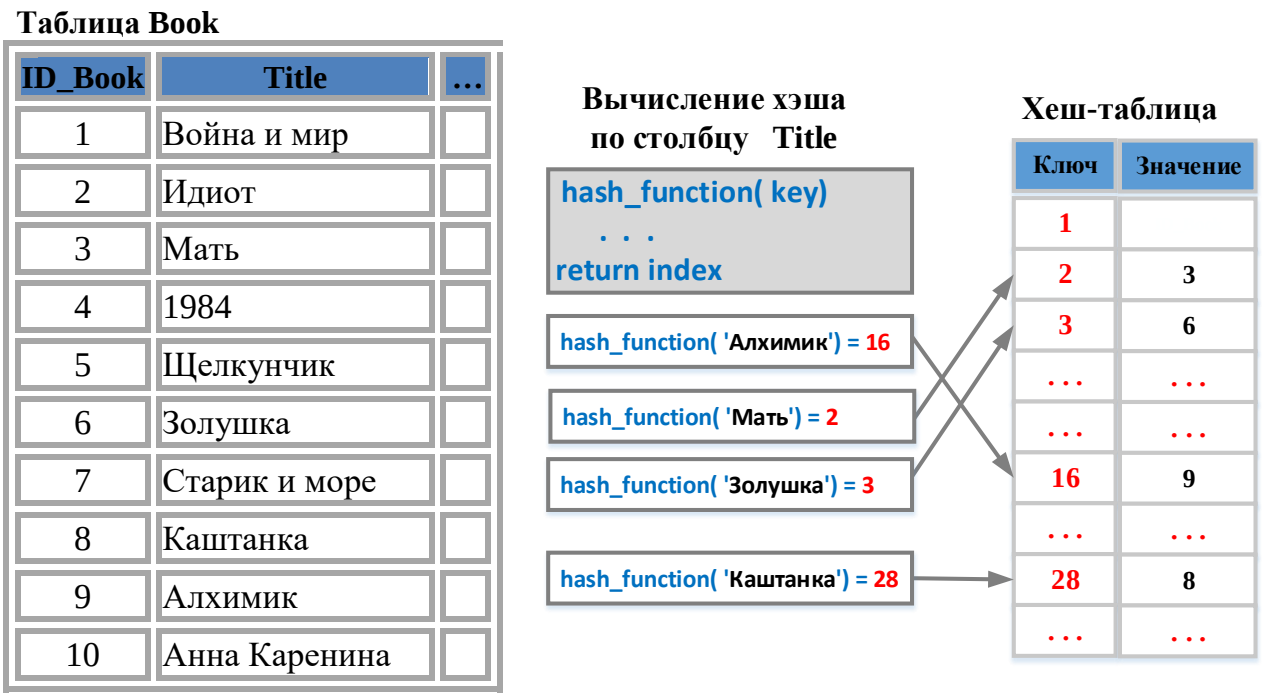

Рисунок 8 – Иллюстрация заполнения хэш-таблицы

Обычно выбор метода индексации данных в реляционной базе данных определяется самой системой управления базами данных (СУБД) и ее оптимизатором запросов. СУБД автоматически выбирает наиболее подходящий метод индексации, исходя из структуры данных, объема данных, типа запросов и других факторов. Однако знание различных методов индексации, их особенностей и преимуществ может быть полезным для разработчиков баз данных и администраторов баз данных. Понимание того, как работают B-деревья, хэш-таблицы и другие методы индексации, может помочь в оптимизации производительности запросов, понимании причин возможных проблем и принятии решений о настройке индексов.

Использование одного индекса для нескольких полей в базе данных позволяет оптимизировать запросы, которые сортируются или фильтруются по комбинации значений этих полей. В этом случае речь идет о составных индексах. Это позволяет системе управления базой данных эффективно

обрабатывать запросы, которые требуют сортировки или фильтрации данных по комбинированным критериям. Использование одного составного индекса для нескольких полей может снизить накладные расходы на обслуживание индексов в базе данных, так как это позволяет объединить несколько индексов в один эффективный индекс, а не хранить индекс для каждого поля.

Примером может быть создание составного индекса для полей «Фамилия» и «Имя» в таблице с данными о клиентах. Этот составной индекс позволит эффективно выполнять запросы, которые фильтруют записи по обоим полям, например, поиск клиента по его фамилии и имени одновременно. Однако чаще всего составные индексы создаются либо для ключевых полей, числовых полей и полей с датой, поскольку обычно фильтрации и сортировки чаще всего выполняются по числовым данным или датам. В целом, создание составных индексов зависит от конкретной структуры базы данных и перечня запросов, которые выполняются чаще всего.

Как правило, индексация не проводится при каждом изменении в таблице. Проведение индексации при каждом изменении может быть излишним, так как это может привести к избыточным операциям обновления индексов и лишней нагрузке на систему. Вместо этого, большинство СУБД оптимизированы для автоматического управления индексами и проведения индексации по мере необходимости. Автоматическая индексация включает в себя механизмы, которые следят за изменениями в данных и оптимизируют индексы самостоятельно, чтобы обеспечить оптимальную производительность базы данных. Тем не менее, в некоторых случаях администраторы баз данных могут вмешиваться и проводить ручное обновление индексов. Такие вмешательства могут быть обусловлены изменениями в структуре базы данных, изменениями в функциональности приложения или в связи с требованиями конечных пользователей (требования бизнеса) о повышении скорости отдельных типов запросов. Автоматическая индексация осуществляется фоновыми процессами в

соответствии с заданными расписаниями или в ответ на изменения в данных. Это позволяет обеспечить непрерывную работу системы, не блокируя выполнение других операций. Часто используется комбинация обоих методов: регулярное выполнение индексации по расписанию и запуск обновлений в ответ на существенные изменения в данных.

В крупных торговых площадках, финансовых учреждениях, телекоммуникационных компаниях, где данные постоянно изменяются из-за большого количества пользователей и операций, индексация является критически важной для поддержания высокой производительности системы. В таких системах индексация обычно проводится очень часто или даже непрерывно, чтобы гарантировать актуальность индексов и оптимальную производительность запросов. В таких БД рекомендуется использовать специализированные методы индексации, такие как инкрементальная индексация или частичная индексация, которые позволяют обновлять индексы эффективно и без значительного влияния на производительность системы. Частота индексации в таких системах рассчитывается исходя из нескольких факторов, таких как объем данных, частота изменений, требования к производительности, возможности аппаратных средств, на которых функционируют СУБД и БД и другие.

Индексы не являются эксклюзивной особенностью реляционных баз данных. В NoSQL базах данных индексы играют важную роль в ускорении доступа к данным, а также в обеспечении эффективного выполнения запросов. Например, в MongoDB, индексы могут быть созданы для полей документов, чтобы ускорить поиск и фильтрацию данных. В графовых базах данных индексы также используются для оптимизации поиска и навигации по графовым структурам. Они позволяют быстро находить узлы и ребра в графе, что делает выполнение запросов более эффективным. В объектно-ориентированных базах данных индексы могут использоваться для быстрого поиска объектов по определенным атрибутам или для оптимизации навигации по объектным структурам. В NoSQL базах данных разработчики

обычно могут контролировать выбор и определение методов индексации данных, работая более непосредственно с механизмами индексации, настраивая их в соответствии с требованиями приложения.

Обычно индексы определяются и создаются при проектировании базы данных. Разработчики и администраторы баз данных учитывают требования приложения, типы запросов, объем данных и другие факторы при принятии решения о создании индексов. Это важный этап процесса проектирования базы данных, который направлен на обеспечение оптимальной производительности системы при выполнении запросов.

Объединения

Объединение (JOIN) – это команда SQL, которая соединяет данные из двух разных таблиц базы банных с целью получения нужного подмножества данных. Команда JOIN одна из наиболее важных в SQL. В реляционных базах данных информация распределена по таблицам, а большая часть работы в таких базах подразумевает поиск в нужных сведений из разных таблиц одновременно. Для этого используются разные команды, и JOIN – одна из них. Он сочетается с оператором SELECT, который выбирает данные из таблицы, а также с операторами WHERE, ORDER BY и GROUP BY, используемыми в рамках SELECT для фильтрации, сортировки и группировки данных соответственно.

JOIN используется для объединения данных из двух или более таблиц на основе совпадающих значений в определенных полях. Примерами таких объединений могут быть: комплектующий_1 и комплектующий_2, у которых один производитель, студенты, записанные на одни и те же учебные курсы, пациенты с одинаковыми заболеваниями или симптомами, абоненты сотовой связи, использующие один и тот же тариф. В этих примерах объекты размещаются в разных таблицах, но имеют общие атрибуты или схожие характеристики. Использование разных режимов работы JOIN не только

позволяет выполнить соединение двух таблиц в одну, но отфильтровать (выбирать) записи из двух таблиц по определенным критериям, а также исключать строки, которые не удовлетворяют условиям запроса.

Изначально JOIN – бинарный оператор, то есть он работает с двумя переданными ему объектами – с двумя таблицами. На практике современные реализации могут воспринимать и больше двух таблиц, просто операция в таком случае выполняется несколько раз. Команде JOIN передаются таблицы, которые нужно объединить, и критерий для объединения – логическое выражение, которое называется ключом. Ключ может быть определен на основе значений одного или нескольких столбцов в каждой из таблиц. JOIN проверяет соответствие этих значений и формирует результат, включающий комбинацию данных из обеих таблиц, где соответствующие значения ключа совпадают.

Строки чаще всего выбираются из разных таблиц, но это не обязательно. Главное преимущество оператора JOIN заключается в том, что он позволяет объединять данные из разных таблиц в один результат, что часто используется для составления связанных данных из нескольких источников.

Команда JOIN имеет разные режимы работы или разные реализации. Стандарт языка SQL определяет два типа объединений – внутреннее (INNER JOIN) и внешнее (OUTER JOIN). Причем внешнее объединение представляет собой семейство, состоящее из левого (LEFT OUTER JOIN или просто LEFT JOIN), правого (RIGHT OUTER JOIN или просто RIGHT JOIN), а также полного (FULL OUTER JOIN или просто FULL JOIN). Для левого и правого объединений существуют еще и условные варианты – «LEFT JOIN (if NULL)» и «RIGHT JOIN (if NULL)». Они не являются официальными частями стандарта SQL, потому что могут быть реализованы через другие конструкции языка SQL. Однако их применение оправдано в случае, когда нет совпадений в столбцах, указанных в условии JOIN. Особняком стоят еще две команды SELF JOIN и CROSS JOIN. Строго говоря, они не являются

типами JOIN. SELF JOIN относится к ситуации, когда таблица присоединяется сама к себе, обычно для сравнения значений в одной таблице. Это не совсем тип JOIN, скорее концепция использования JOIN на одной и той же таблице. CROSS JOIN – это операция, которая производит декартово произведение двух таблиц, возвращая результат, который является комбинацией каждой строки из первой таблицы с каждой строкой из второй таблицы. Особенности работы каждого типа легче всего объяснить с помощью диаграмм Венна, которые также называют кругами Эйлера (рисунок 9). На них множества представляются как круги, а объекты, которые относятся к обоим множествам, – как пересечения этих кругов. В контексте баз данных каждый круг обозначает отдельную таблицу, а пересечение кругов соответствует результатам операций JOIN, где строки из разных таблиц удовлетворяют определенным условиям объединения.

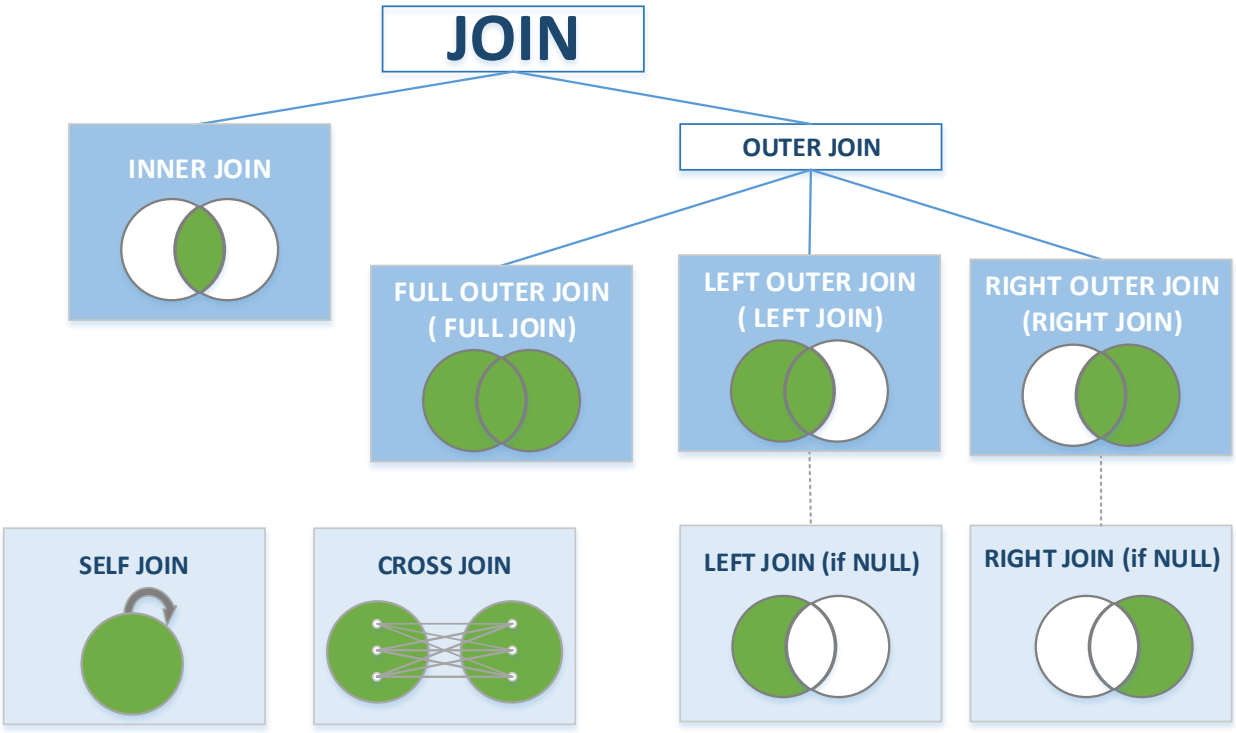

Рисунок 9 – Варианты команды JOIN

INNER JOIN – внутреннее объединение. Это самый простой и часто используемый вариант команды. Если режим работы команды явно не

указан, то компилятор языка SQL автоматически воспримет его как внутренний JOIN. При использовании INNER JOIN передаются две таблицы, и оператор возвращает их внутреннее пересечение по заданному критерию объединения. Это означает, что результирующий набор данных будет содержать только записи, которые имеют соответствующие значения в обеих таблицах, основываясь на заданных условиях объединения. Такое объединение имеет сходство с логической операций «И». Например, в базе данных есть две таблицы: «Кафе» и «Рестораны». Обе таблицы содержат атрибут «Тип кухни». Если выполнить INNER JOIN между этими двумя таблицами по атрибуту «Тип кухни», то результирующий набор данных будет содержать только сходные объекты – кафе и рестораны, предлагающие одинаковую кухню. Пример SQL-запроса:

```
SELECT *
FROM Cafe
INNER JOIN Restaurant ON Cafe.Cuisine_type = Restaurant.Cuisine_type
```

Table "Café"

ID	CafeName	Cuisine_type
1	Кафе А	Итальянская
2	Кафе Б	Французская
3	Кафе В	Японская
4	Кафе Г	Итальянская
5	Кафе Д	Русская

Table "Restaurant"

ID	RestaurantName	Cuisine_type
1	Ресторан 1	Французская
2	Ресторан 2	Итальянская
3	Ресторан 3	Китайская
4	Ресторан 4	Русская
5	Ресторан 5	Китайская

Result Table

ID	CafeName	Cuisine_type	ID	RestaurantName	Cuisine_type
1	Кафе А	Итальянская	2	Ресторан 2	Итальянская
4	Кафе Г	Итальянская	2	Ресторан 2	Итальянская
2	Кафе Б	Французская	1	Ресторан 1	Французская
3	Кафе Д	Русская	4	Ресторан 4	Русская

Порядок следования таблиц в запросе INNER JOIN обычно не имеет значения. Обе таблицы рассматриваются как равноправные и объединяются на основе указанных условий. Таким образом, можно поменять местами «Кафе» и «Рестораны» в запросе, и результат будет тем же самым. Важно

только убедиться, что атрибут, по которому необходимо объединить таблицы, существует и имеет одинаковый формат данных в обеих таблицах.

OUTER JOIN – внешнее соединение. Если внутреннее объединение имеет сходство с бинарным «И», то внешнее – это уже несколько вариаций логического «ИЛИ». Команды OUTER JOIN возвращают не только строгое пересечение между двумя таблицами, но и отдельные элементы, которые принадлежат только одному из множеств.

LEFT JOIN возвращает результат, который включает в себя все строки из левой таблицы, независимо от того, есть ли соответствующие строки в правой таблице. Если для строки из левой таблицы нет соответствующей строки в правой таблице, то в результирующем наборе для соответствующих столбцов из правой таблицы будут возвращены значения NULL. Здесь левая таблица – та, которая в запросе указана первой, а правая – второй. Например, в базе данных есть две таблицы: «Клиент» и «Заказ». Клиенты могут делать заказы, и в базе данных есть информация о клиентах и их заказах. Пример SQL-запроса:

```
SELECT Client.Name, Order.OrderID, Order.Product
FROM Client
LEFT JOIN Orders ON Client.ClientID = Order.ClientID
```

Table "Client"

ClientID	Name
1	Анна
2	Иван
3	Мария

Table "Order"

OrderID	ClientID	Product
101	1	Кофе
102	1	Пирожное
103	2	Чай

Result Table

ClientID	Name	OrderID	Product
1	Анна	101	Кофе
4	Анна	102	Пирожное
2	Иван	103	Чай
3	Мария	(NULL)	(NULL)

В этом запросе используется LEFT JOIN, чтобы получить всех клиентов вместе с их заказами. Для этого производится соединение таблицы

«Клиенты» с таблицей «Заказы» по полю «ClientID». LEFT JOIN гарантирует, что все клиенты из таблицы «Клиент» будут включены в результирующий набор, даже если у них нет заказов. Если у клиента нет заказов, то для полей из таблицы «Заказы» будут возвращены значения NULL.

RIGHT JOIN вместо левой таблицы включает все строки из правой таблицы, независимо от того, есть ли соответствующие строки в левой таблице. Если для строки из правой таблицы нет соответствующей строки в левой таблице, то в результирующем наборе для соответствующих столбцов из левой таблицы будут возвращены значения NULL. Например, в базе данных имеется таблица «Абонент», содержащая информацию о различных абонентах мобильной связи, и таблица «Тарифный план», содержащая информацию о доступных тарифных планах. Возможна ситуация, когда есть тарифные планы, которые пока еще не используются ни одним абонентом. В этом случае можно использовать RIGHT JOIN, чтобы получить список всех тарифных планов, включая те, к которым не подключены абоненты:

```
SELECT *
FROM Tariff
RIGHT JOIN Abonent ON Tariff.TariffID = Abonent.TariffID
```

Table "Abonent"

AbonentID	AbonentName	TariffID
1	Анна	2
2	Иван	3
3	Мария	1

Table "Tariff"

TariffID	TariffName	Cost	IncludedData
1	План А	300 р.	10 Гб
2	План Б	400 р.	20 Гб
3	План С	500 р.	30 Гб
4	План Д	600 р.	40 Гб

Result Table

AbonentID	AbonentName	TariffID	TariffID	TariffName	Cost	IncludedData
1	Анна	2	2	План Б	400 р.	20 Гб
4	Иван	3	3	План С	500 р.	30 Гб
2	Мария	1	1	План А	300 р.	10 Гб
NULL	NULL	NULL	4	План Д	600 р.	40Гб

Если тарифный план не имеет подключенных абонентов, в соответствующих столбцах таблицы «Абоненты» будут возвращены значения NULL.

FULL JOIN возвращает обе таблицы, объединенные в одну. Результаты, возвращаемые этим типом соединения, включают все строки из обеих таблиц. Там, где встречаются совпадения, значения связаны. Если соответствия из любой таблицы нет, вместо этого возвращается NULL. Например, в базе данных имеется таблица «Студент» и таблица «Курс». Каждый студент может быть записан на несколько курсов, и каждый курс может иметь несколько студентов. Третья таблица «Зачисление» (Enrollments) является связующей и содержит сведения о том, кто из студентов уже записался на один или несколько курсов. После применения FULL JOIN между таблицами «Студент» и «Зачисление», а также между таблицами «Курс» и «Зачисление» результирующая таблица будет содержать сведения обо всех студентах и всех курсах, даже если они не имеют соответствующих записей в связующей таблице записи на курс.

```
SELECT Student.Name, Course.CourseName
FROM Student
FULL JOIN Enrollment ON Student.StudentID = Enrollment.StudentID
FULL JOIN Course ON Enrollment.CourseID = Course.CourseID;
```

Table "Student"

StudentID	Name
1	Анна
2	Иван
3	Мария

Table "Course"

CourseID	CourseName
101	История
102	Физика
103	Химия

Table "Enrollment "

StudentID	CourseID
1	101
1	102
2	102

Result Table

StudentID	Name	CourseID	CourseName
1	Анна	101	История
4	Анна	102	Физика
2	Иван	102	Физика
3	Мария	NULL	NULL
NUL	NULL	103	Химия

Таким образом, FULL JOIN позволяет посмотреть – кто из студентов еще не записался ни на один курс, и на какие курсы еще не записался ни один студент. FULL JOIN как раз и полезен для анализа данных с возможными пропущенными соответствиями, отладки запросов и проверки корректности данных, интеграции данных из нескольких источников.

LEFT JOIN (IF NULL) работает подобно LEFT JOIN, но он предоставляет дополнительную возможность для работы с NULL значениями. LEFT JOIN (IF NULL) обеспечивает включение всех строк из левой таблицы в результирующий набор, а также позволяет определить, как обрабатывать NULL значения, которые могут появиться в полях, полученных в результате объединения. Например, нужно объединить две таблицы - Table_A и Table_B их по колонке id и заменить NULL значения в Column2 из Table_B на строку 'Нет данных', если они встречаются:

```
SELECT Table_A.id, IFNULL(Table_B.Column2, 'Нет данных') AS Column2
FROM Table_A
LEFT JOIN Table_B ON Table_A.id = Table_B.id;
```

Также можно выполнить замену NULL значений на определенное значение с помощью функций COALESCE(), которая позволяет указать несколько аргументов и возвращает первое непустое значение из списка, в то время как функция IFNULL просто заменяет NULL на альтернативное значение. Например:

```
SELECT COALESCE(Column1, Column2, 'Значение по умолчанию') AS new_column
FROM table_name;
```

RIGHT JOIN с NULL позволяет управлять обработкой NULL-значений в результирующем наборе аналогично LEFT JOIN (IF NULL), только с учетом правой таблицы

CROSS JOIN – это своеобразный вариант соединения, который используется не так часто, как вышеперечисленные способы объединения, но

важен для понимания. Он возвращает декартово произведение, т.е. комбинацию строк из обеих таблиц без каких-либо условий соединения между ними. В качестве примера соединение CROSS JOIN может использоваться при создании фильтров в интернет-магазине. Например, человек ищет обувь по характеристикам «тип» и «размер»:

```
SELECT TypesOfShoes.ShoeType, Sizes.ShoeSize
FROM TypesOfShoes
CROSS JOIN Sizes;
```

Table " TypesOfShoes"

ShoeType
Кроссовки
Туфли
Сапоги

Table "Size"

ShoeSize
36
37
38
39
40

Result Table "

ShoeType	ShoeSize
Кроссовки	36
Кроссовки	37
Кроссовки	38
Кроссовки	39
Кроссовки	40
Туфли	36
Туфли	37
Туфли	38

Результатом такого запроса должны быть все возможные комбинации типа с размером. Однако такое «картезианское произведение» может привести к огромному количеству строк в результирующей таблице. Если исходные таблицы содержат большое количество записей, а также, если не используются надлежащие фильтры или ограничения, результат CROSS JOIN может привести к экспоненциальному росту числа комбинаций строк.

SELF JOIN – это «самосоединение» или объединение строк внутри одной таблицы в том случае, когда нужно проанализировать зависимости внутри одной таблицы. Примеры использования SELF JOIN – обработка деревьев или графов, которые хранятся в одной таблице – генеалогические деревья, иерархии в компаниях, сложные структуры данных, такие как сетевой маркетинг. Механизм SELF JOIN полезен и при поиске дубликатов, поскольку позволяет сопоставить записи в одной таблице для определения повторяющихся данных. Чтобы сформировать запрос, следует указать одну и

ту же таблицу дважды под разными именами – псевдонимами. В результате оператор «воспринимает» переданные ему сущности как разные. Этот вариант объединения может быть и внутренним, и внешним. Его особенность заключается в том, что таблица при таком режиме присоединяется сама к себе. Рассмотрим пример таблицы «Сотрудники», хранящей иерархию компании. Каждая строка представляет собой запись о сотруднике. У каждого сотрудника есть уникальный идентификатор EmployeeID, имя Employee_Name и, если он является подчиненным, идентификатор его менеджера ManagerID. В запросе реализуется SELF JOIN таблицы «Employees» с использованием псевдонимов «e1» и «e2». Таблица соединяется сама с собой по условию равенства идентификатора менеджера в первой таблице идентификатору сотрудника во второй таблице.

```
SELECT e1.EmployeeName AS Employee, e2.EmployeeName AS Manager
FROM Employee e1
JOIN employee e2 ON e1.ManagerID = e2.EmployeeID;
```

Table "Employee"

EmployeeID	EmployeeName	ManagerID
1	Мария	2
2	Иван	4
3	Анна	2
4	Евгений	NULL

Result Table

Employee	Manager
Мария	Иван
Иван	Евгений
Анна	Иван

После выполнения запроса будут получены имена сотрудников и их менеджеров. Слово SELF в теле запроса указывать не нужно. Однако необходимо указать имена столбцов результирующей таблицы. Названия столбцов должны отражать смысл их содержимого, чтобы обеспечить понимание структуры данных.

Важную роль в оптимизации JOIN операций играют индексы, ускоряя выполнение запросов. Использование индексов на столбцах, используемых для соединения таблиц, помогает системе управления базой данных быстро находить нужные строки, повышает эффективность запросов и снижает

нагрузку на базу данных. Это особенно важно при работе с большими объемами данных или при выполнении запросов, объединяющих несколько таблиц. При использовании объединений таблиц есть несколько тонкостей, о которых следует помнить:

Выбор правильного типа объединения зависит от конкретной задачи. Например, если нужно получить только те строки, которые имеют соответствующие значения в обеих таблицах, то подойдет внутреннее объединение.

Если необходимо сохранить все строки из одной таблицы и добавить к ним соответствующие значения из другой таблицы, то следует использовать левое или правое объединение в зависимости от того, какие данные в данном случае важнее.

При объединении таблиц нужно убедиться, что в столбцах, по которым объединяются таблицы, значения уникальны или взаимно однозначно соответствуют друг другу. Иначе могут возникнуть непредвиденные дубликаты в результирующей таблице.

При работе с большими объемами данных важно оптимизировать запросы на объединение таблиц, для этого следует включать в команду JOIN выбор только необходимых столбцов, использование индексов, а также правильное использование условий соединения.

В традиционном понимании JOIN не является стандартной операцией для NoSQL баз данных, поскольку их модели данных в большинстве случаев существенно отличаются от табличной структуры. Вместо JOIN NoSQL базы данных предлагают свои, специфические для данной модели методы, которые могут использоваться для агрегации, комбинирования, группировки и анализа данных из нескольких источников.

JOIN – это мощный инструмент в SQL, который позволяет получать более сложные и полные результаты запросов, объединяя информацию из различных источников данных. Понимание JOIN позволяет эффективно работать с данными в базе данных, выполнять сложные запросы и получать

нужную информацию. Знание JOIN также помогает оптимизировать структуру базы данных, улучшить производительность запросов и создавать более гибкие и мощные приложения, основанные на базе данных.

Метаданные в базах данных

Термин метаданные сочетает в себе слово «данные» и греческий префикс «мета-», который означает «после чего-то « или «за чем-то». А в лингвистике «мета-» обычно используется для указания абстракции или высшего уровня, что отражает идею превосходства чего-то над чем-то другим. Таким образом, в контексте данных, «метаданные» можно интерпретировать как «данные о данных» или «данные, которые выходят за пределы других данных», что подчеркивает их вспомогательную роль в описании, классификации, управлении и интерпретации основных данных. Например, если имеется некий текстовый документ, метаданные могут включать информацию об авторе документа, дате создания, объеме и прочих атрибутах, которые сами по себе не являются частью основного содержимого документа, но описывают его свойства и контекст.

В контексте баз данных метаданные представляют собой информацию о самой базе данных, её структуре, хранимых объектах и свойствах.

Метаданные о структуре базы данных обычно хранятся во внутренних системных базах данных, также называемых системными схемами, системными базами данных или каталогами. Каждая СУБД имеет свою собственную системную схему, где хранятся системные таблицы, содержащие те или иные метаданные. Например, в СУБД MySQL метаданные хранятся в системной схеме information_schema, а в Microsoft SQL Server – в системной схеме sys. Системные таблицы содержат информацию о таблицах, столбцах, индексах, ключах, представлениях и других объектах базы данных. Например, information_schema содержит следующие таблицы:

TABLES – эта таблица содержит информацию о таблицах в базе данных, такую как их названия, размеры, способы хранения на диске, способы управления доступом и другие атрибуты.

COLUMNS – в этой таблице хранится информация о столбцах каждой таблицы, включая их названия, типы данных, длину и другие свойства.

KEY_COLUMN_USAGE – эта таблица содержит информацию о столбцах, используемых в индексах и внешних ключах. Это может включать названия индексов, таблиц и столбцов, участвующих в индексах.

INDEXES – эта таблица содержит информацию обо всех индексах, определенных для таблиц в базе данных. Она включает названия индексов, таблиц, типы индексов и другие свойства.

STATISTICS – в этой таблице хранится статистическая информация о распределении данных в таблицах. Это данные о кардинальности (количество уникальных значений) и распределении данных в столбцах, помогают оптимизатору запросов принимать решения о выборе наилучших индексов для выполнения запросов.

ROUTINES – содержит информацию о так называемых хранимых процедурах (Stored Procedure) в базе данных – их названия, типы (процедура или функция) и собственно код. Хранимые процедуры представляют собой набор инструкций SQL, которые выполняются по требованию приложений или администраторов баз данных. Они используются для упрощения и оптимизации выполнения повторяющихся операций или сложных бизнес-логик в рамках базы данных.

Это только небольшой набор таблиц из information_schema, и каждая из них содержит дополнительные столбцы и информацию для обеспечения полного описания структуры базы данных.

В каждой СУБД доступ к метаданным может быть реализован по-разному, и имена и структура системных таблиц могут различаться. Однако идея о предоставлении доступа к метаданным для администрирования и анализа базы данных остается общей чертой для всех СУБД. Обращение к

системным таблицам, хранящим метаданные, подобно обращению к данным, но в различных СУБД тоже организовано по-разному. Можно выделить несколько общих способов обращения к метаданным:

1. Использование специальных команд или процедур для получения информации из системных таблиц. Например, в Oracle можно использовать команду DESCRIBE для получения информации о структуре таблицы. В MS SQL Server для получения информации о структуре таблицы и её индексах используются системные хранимые процедуры sp_help и sp_helpindex.

2. Использование стандартных SQL-запросов к системным таблицам. Например, чтобы получить список всех таблиц в базе данных, можно выполнить запросы:

```
#в PostgreSQL
SELECT * FROM information_schema.tables WHERE table_schema = 'public'

#в MySQL
SELECT * FROM information_schema.TABLES WHERE TABLE_SCHEMA = 'database_name'
```

3. Использование специализированных API: Некоторые СУБД предоставляют специализированные API или библиотеки для работы с метаданными. Например, в Python для работы с метаданными PostgreSQL можно использовать библиотеку psycopg2, а для работы с метаданными MySQL - mysql-connector-python.

Наиболее активными, но не единственными пользователями метаданных являются администраторы баз данных. Метаданные могут быть полезными для широкого круга пользователей, включая разработчиков, аналитиков, тестировщиков и архитекторов приложений. Разработчики могут использовать метаданные для проектирования и оптимизации запросов к базе данных, понимания структуры данных и определения необходимых изменений схемы для поддержки новых функций приложения. Специалисты по бизнес-аналитике и интеллектуальному анализу могут использовать метаданные для понимания структуры данных, определения ключевых

показателей производительности и анализа использования данных для принятия бизнес-решений. Архитекторы приложений могут использовать метаданные для проектирования и оптимизации архитектуры баз данных, а также для определения стратегии резервного копирования, масштабирования и обеспечения надежности системы. Аудиторы и регуляторные органы могут использовать метаданные для проверки соответствия баз данных законодательству и стандартам безопасности, а также для анализа аудиторских следов и обеспечения целостности данных.

Помимо описания набора таблиц и их полей, метаданные могут указывать, какие пользователи имеют доступ к определенным объектам базы данных, какие разрешения на чтение, запись, выполнение у них есть, и какие ограничения применяются к этому доступу.

Некоторые системы баз данных сохраняют метаданные о, истории изменений структуры базы данных, включая информацию о том, когда и кем были внесены изменения в объекты базы данных. Другие позволяют добавлять комментарии и документацию к объектам базы данных, которые могут быть сохранены в метаданных и использоваться для их описания и понимания.

В целом, метаданные в базах данных не только описывают структуру и характеристики данных, но и играют ключевую роль в управлении и анализе базы данных, обеспечивая целостность, безопасность и эффективное управление данных.

Компилятор запросов

Компилятор запросов или парсер (англ., parse – разбор) – это компонент системы управления базами данных, который отвечает за анализ и разбор запросов, поступающих от пользователей или приложений, и их преобразование во внутренние операции для выполнения соответствующими компонентами СУБД.

Вот основные шаги, которые обычно выполняет компилятор запросов:

– Лексический анализ (токенизация, от английского token – знак, символ). На этом шаге текст запроса разбивается на лексемы или токены, такие как ключевые слова, идентификаторы, операторы и знаки пунктуации. Результатом этого шага является построение лексического дерева (lexical tree), которое отражает иерархию токенов и их отношения в начальной форме.

– Синтаксический анализ (парсинг). На этом этапе анализируются токены, проверяется их синтаксис согласно грамматике языка запросов. Результатом этого шага является построение абстрактного синтаксического дерева AST (abstract lexical tree), представляющего операции и операнды в виде иерархии.

– Семантический анализ. На этом этапе проверяется семантика запроса, то есть его согласованность с существующими объектами базы данных, а также корректность использования атрибутов, таблиц и других сущностей.

– Генерацию исполняемых планов. Компилятор создает исполняемый план запроса, который определяет порядок выполнения операций и выбранные методы доступа к данным. План выполнения запроса представляет собой концептуальную структуру, описывающую порядок действий, необходимых для получения результата запроса. Этот план не является непосредственно исполняемым кодом, а скорее представляет собой логическую последовательность операций, которые СУБД должна выполнить для выполнения запроса: сканирование таблиц, применение фильтров, объединение результатов, выбор используемых индексов, методы доступа к данным и т.д. Обычно компилятор генерирует несколько возможных планов исполнения запроса на основе доступных индексов, накопленной им статистики исполнения подобных запросов и других факторов. Эти планы могут различаться по эффективности выполнения запроса, использованию

ресурсов и времени. Количество сгенерированных планов исполнения зависит от сложности запроса, структуры таблиц и доступных индексов.

– Оптимизацию запроса. Далее компилятор выбирает наиболее эффективный план выполнения запроса, который минимизирует количество ресурсов (времени и памяти), необходимых для выполнения этого запроса.

– Генерацию исполняемого кода. На основе оптимизированного плана выполнения запроса, компилятор генерирует соответствующий исполняемый код или инструкции для выполнения запроса на целевой платформе. Сгенерированный код или инструкции выполняются на целевой платформе, и результаты запроса возвращаются пользователю или приложению.

Все этапы процесса компиляции обеспечивают эффективное выполнение запросов к базе данных, учитывая синтаксические, семантические и оптимизационные аспекты запроса.

Поясним это более простым языком. Представьте, что компилятор запросов – это такой «переводчик», который превращает запросы к базе данных из человеческого языка в язык, который СУБД может понять и выполнить. Строго говоря, в самой программе происходит преобразование высокоуровневых запросов пользователя (текстового представления, голосового или выбора через интерфейсные элементы) в соответствующие SQL-запросы для передачи их компилятору запросов. Например, программой был инициирован запрос:

```
SELECT Title, Author FROM Book WHERE Genre='фантастика'
```

Сначала компилятор разбивает запрос на отдельные слова или «токены», чтобы понять, из каких элементов состоит запрос: ключевые слова SELECT, FROM, WHERE; названия таблиц и столбцов Book, genre, BookName, Author; строковые константы 'фантастика', операторы '='. Результат этого разбора – лексическое дерево – представлено на рисунке 10.

Рисунок 10 – Лексическое дерево запроса

Затем компилятор анализирует, как эти слова связаны между собой, чтобы понять смысл всего запроса и понять – что нужно сделать. Результат – абстрактное синтаксическое дерево – представлено на рисунке 11.

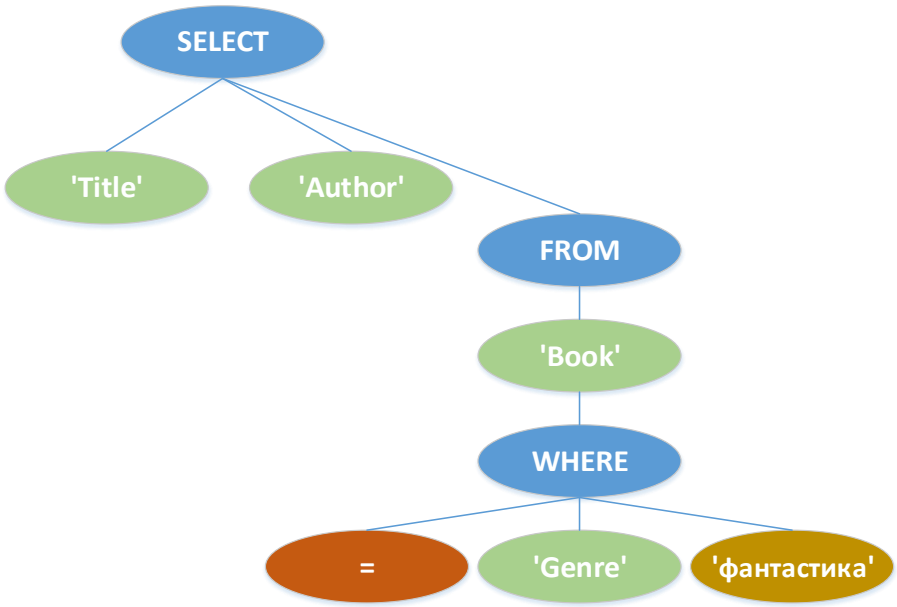

Рисунок 11 – Абстрактное синтаксическое дерево запроса

Далее компилятор убеждается, что запрос имеет правильную грамматику и смысл, так чтобы можно было его выполнить. В приведенном примере компилятору необходимо убедиться в существовании таблицы Book, наличия у пользователя прав на чтение этой таблицы, в наличии в ней

полей Title, Author и Genre. Поскольку поле Genre – внешний ключ, необходимо убедиться в наличии прав доступа к дочерней таблице Genre. А поскольку SQL-компилятор может просматривать метаданные схемы базы данных, чтобы узнать структуру таблиц и их поля, то он еще на этапе компиляции выполняет поиск строк, удовлетворяющих условию WHERE, т.е. проверяет существование значения 'фантастика' в таблице Genre. Схема проверок компилятора представлена на рисунке 12.

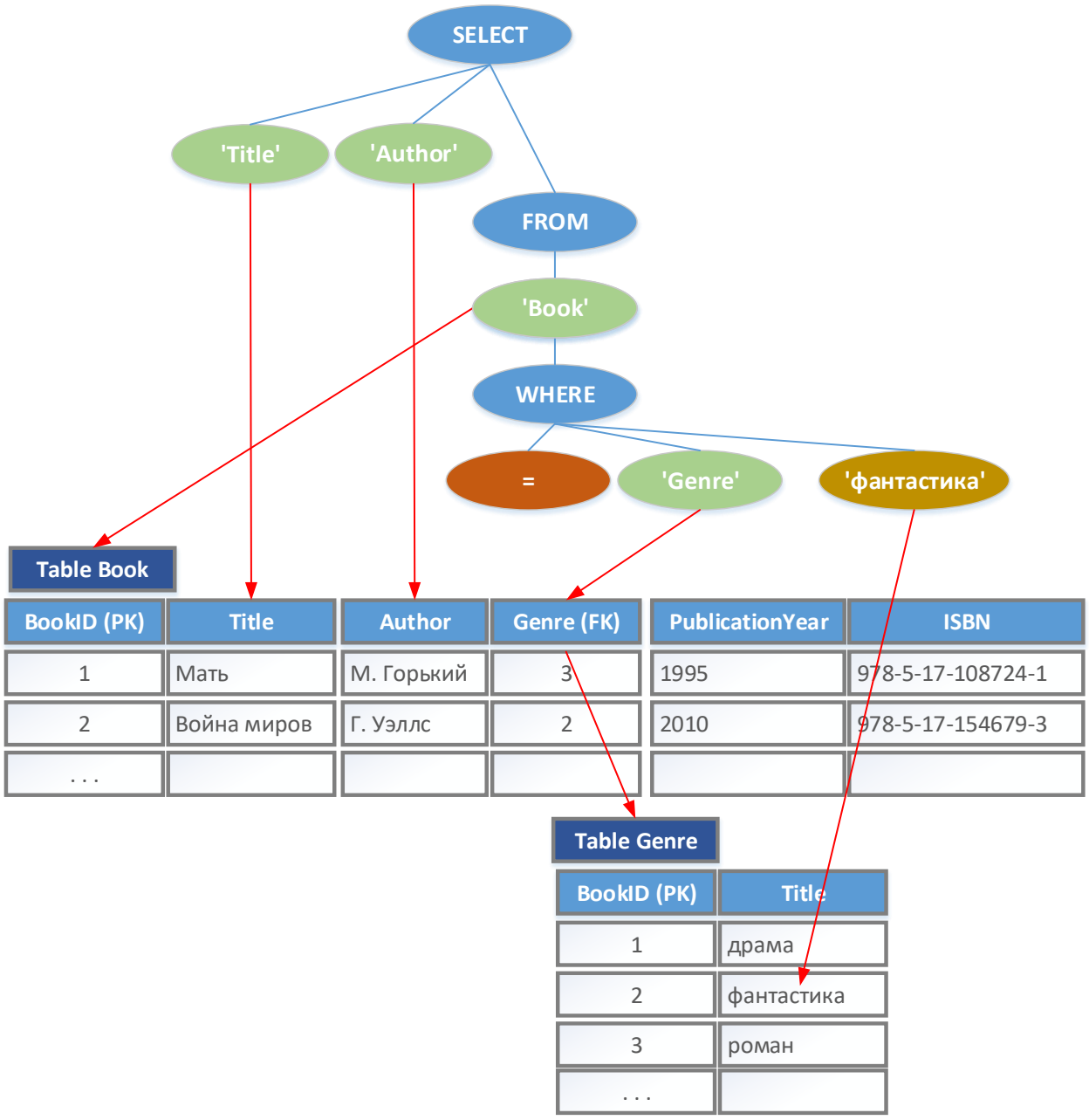

Рисунок 12 – Схема проверок компилятора на этапе синтаксического анализа

Иногда компилятор может подумать, как сделать запрос более эффективным, чтобы он работал быстрее. Например, если запрос был выполнен ранее с теми же параметрами, СУБД может вернуть результат из кэша, что также сократит время выполнения запроса.

И, наконец, компилятор превращает запрос в инструкции плана исполнения запроса, которые сервер базы данных может выполнить. Для приведенного примера логическая последовательность инструкций плана может выглядеть следующим образом (рисунок 13):

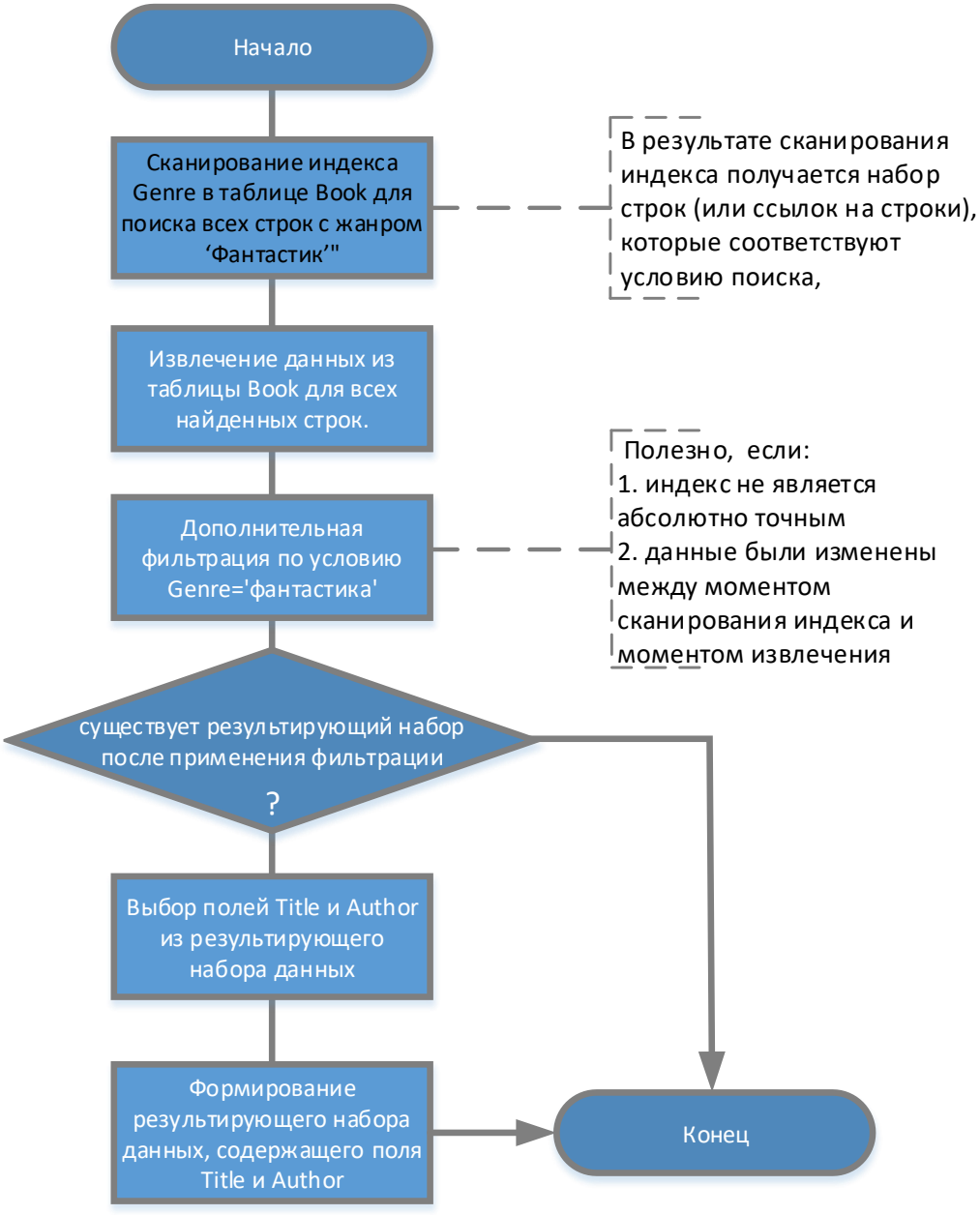

Рисунок 13 – Последовательность операций плана запроса

Получив исполняемый план, система управления базами данных выполняет запрос, обрабатывая его пошагово в соответствии с созданным планом. Этот процесс может включать доступ к данным на диске, выполнение вычислений и применение фильтров и условий.

Итак, компилятор запросов играет важную роль в процессе выполнения SQL-запросов, обеспечивая их корректность, эффективность и безопасность.

Оптимизатор запросов

Оптимизатор запросов (сленговый термин – рерайтер, который происходит от английского слова rewrite, что означает переписывать или переформулировать) – это компонент системы управления базами данных, который отвечает за анализ и преобразование запросов с целью улучшения их производительности. Он осуществляет оптимизацию запросов путем выбора наиболее эффективных способов выполнения запроса и порядка объединения операций. Обычно оптимизатор запросов является важным компонентом компилятора запросов, однако в некоторых СУБД этот компонент может быть реализован как отдельный модуль, который работает независимо от компилятора. Наличие или отсутствие компонента оптимизации запросов в составе компилятора запросов зависит от конкретной реализации СУБД и её архитектуры.

Если оптимизатор является частью компилятора запросов, то он работает в процессе компиляции запроса, выполняя анализ и преобразование запроса до того, как запрос будет преобразован в исполняемый план. Эта работа обычно осуществляется после лексического и синтаксического анализа запроса, но перед генерацией фактического исполняемого плана.

Если оптимизатор работает автономно перед компиляцией, то он может применяться для предварительной обработки запросов до их передачи в

компилятор. Например, когда пользователь отправляет запрос на выполнение, оптимизатор может проанализировать этот запрос и для улучшения его производительности выполнить предварительные преобразования. Например:

– реструктуризацию (переформулирование) запроса; например, переписывание подзапросов, замена одних операторов на другие, даже изменение порядка условий в предложении WHERE или удаления лишних условий и др.

– выбор подходящего метода соединения таблиц, например, вложенный цикл, хеш-соединение или слияние.

Затем уже оптимизированный запрос передается компилятору для создания исполняемого плана.

Автономный оптимизатор может применяться и на этапе выполнения запроса. Например, если в процессе выполнения запроса выясняется, что выбранный исполняемый план неэффективен из-за изменяющихся статистических данных или других факторов, оптимизатор может проанализировать текущий исполняемый план и попытаться оптимизировать его. Например:

- изменить порядок выполнения операций, например, поставить фильтрацию перед соединением или после;
- выбирать более эффективные индексы для доступа к данным.

Эти действия позволяют адаптировать исполняемый план в реальном времени для повышения производительности запроса.

Анализ различных способов выполнения запроса и выбор стратегии оптимизатора базируется на учете различных факторов: структуре таблиц, наличии индексов, объемах данных в используемых таблицах, доступности ресурсов и накопленной статистике исполнения запросов. Статистические данные о таблицах и индексах, о количестве записей в таблице, количестве уникальных значений в столбцах, распределение данных данные помогают

оптимизатору предсказать, сколько строк будет обработано при выполнении операции. Например, если одна из таблиц намного меньше другой, то может быть выгоднее выполнить цикл по меньшей таблице и использовать индекс для быстрого доступа к строкам большей таблицы.

Для каждой операции оптимизатор производит оценку стоимости выполнения. Процесс оценки направлен на определение (предсказание) того, как много времени и ресурсов потребуется для выполнения каждой операции в запросе, чтобы выбрать наиболее эффективный план выполнения запроса. На основе вычисленных оценок оптимизатор выбирает оптимальный план выполнения запроса, который минимизирует общую стоимость выполнения запроса.

Рассмотрим пример, который показывает, как изменение статистических сведений о хранимых данных может повлиять на оценку стоимости выполнения операций в запросе и, следовательно, на выбор оптимального плана выполнения запроса оптимизатором. Предположим, в базе данных имеется таблица User с информацией о пользователях, которая содержит 1000 записей. Предположим также, что в этой таблице есть столбец Age, содержащий возраст пользователей. Пусть средний возраст пользователей равен 30 годам. Оптимизатор запросов может использовать этот статистический параметр и, например, распределение значений в столбце для оценки стоимости выполнения операции выборки. Например, если запрос выбирает пользователей старше 25 лет, оптимизатор оценил, что приблизительно 70% пользователей будут удовлетворять этому условию. Таким образом, стоимость выполнения этой операции может быть оценена, исходя из предполагаемого количества строк, которые будут выбраны.

Пусть средний возраст пользователей равен 35 годам. Теперь, если средний возраст пользователей изменился, то и распределение пользователей по возрасту может быть разным. Предположим, что в этом случае только 60% пользователей будут удовлетворять условию возраста старше 25 лет. Таким образом, оценка стоимости выполнения операции выборки может

измениться в соответствии с этими новыми данными статистическими данными.

Стоимость выполнения операций обычно измеряется в условных единицах, которые представляют собой абстрактные единицы измерения ресурсов или времени, потребляемых для выполнения операции. Эти условные единицы могут варьироваться в зависимости от конкретной реализации СУБД и метода оценки стоимости, используемого оптимизатором запросов. Например, стоимость может измеряться в единицах I/O (ввода/вывода) для операций доступа к диску, в единицах CPU времени для операций обработки данных на стороне сервера баз данных, или в других подобных единицах. Важно понимать, что эти условные единицы не имеют строгого соответствия реальным физическим измерениям, таким как миллисекунды или байты. Они представляют собой абстрактные значения, используемые оптимизатором запросов для сравнения различных альтернативных планов выполнения запроса и выбора наиболее эффективного из них.

Обычно производители систем управления базами данных не раскрывают детали своих методов оценки стоимости. Эти методы являются частью коммерческой тайны и интеллектуальной собственности компании. Как правило, предоставляется общая информация о том, какие факторы учитываются при принятии решения о выборе плана выполнения запроса, но не детализируют конкретные методы оценки стоимости. Некоторые производители предоставляют рекомендации по сбору статистики данных, использованию индексов или других средств оптимизации в их конкретной СУБД с целью повышения эффективности работы оптимизатора. Также стоит отметить, что некоторые СУБД являются open source проектами, и их методы оптимизации запросов открыты для изучения и понимания, поскольку исходный код доступен для общественного просмотра и анализа.

Оптимизатор запросов является важной частью СУБД, особенно при работе с большими объемами данных и сложными запросами, где правильная оптимизация может существенно повысить производительность системы.

Статистика исполнения запросов

Статистика исполнения запросов (или просто статистика) – это собираемая сервером БД и хранимая в своих недрах информация об исполнении запросов к базе данных. Статистические данные об исполнении запросов являются частью общих статистических данных о базе данных, собираемых и хранящихся в СУБД.

Сборщики статистики – это основной компонент для получения информации о производительности запросов в базе данных. Эти компоненты работают внутри самой системы управления базами данных и автоматически собирают данные о производительности при выполнении запросов, такие как время выполнения запросов, использование ресурсов и количество обработанных строк. У различных систем управления базами данных могут быть свои механизмы сбора статистики, и не обязательно они называются «сборщиками статистики». Это могут быть внутренние механизмы, встроенные непосредственно в СУБД, или внешние инструменты, такие как агенты мониторинга или сборщики данных. Однако большинство крупных и современных СУБД обладают такими встроенными механизмами.

Первичные сборщики работают автоматически и непрерывно собирают данные о выполнении запросов и использовании ресурсов базы данных, что позволяет получить полную картину производительности базы данных без необходимости вручную настраивать сбор информации. Сборщики статистики работают в фоновом режиме и при оптимальной настройке могут собирать данные о производительности, не замедляя работу приложения с базы данных. Сбор статистики может осуществляться непрерывно или периодически в зависимости от настроек и конфигурации СУБД. Некоторые

СУБД предоставляют возможность настройки и расширения первичных возможностей сборщиков. Это может быть настройка выбора типов собираемой статистики, настройка интервалов сбора данных, сбор дополнительных метрик производительности. От настроек зависит и объем собираемых данных. Например, некоторые системы могут собирать статистику для каждого выполненного запроса и для этого требуются существенные объемы внутреннего хранилища, другие могут собирать данные каждые несколько минут или часов. Администраторы баз данных могут настраивать параметры сборщиков статистики в соответствии с собственными требованиями. Например, они могут установить пороги для сбора данных только для запросов, выполняющихся дольше определенного времени, или для запросов, которые потребляют определенный объем ресурсов. Важно отметить, что эффективность работы сборщиков статистики может существенно влиять на производительность системы, поэтому необходимо тщательно настраивать и оптимизировать их работу в соответствии с требованиями к базе данных и приложению.

Собранные данные обычно сохраняются в специальных таблицах или журналах внутри базы данных – «системных представлениях». Эти данные доступны для последующего анализа и использования другими инструментами для мониторинга и оптимизации производительности базы данных с течением времени. Статистика обычно сохраняется в течение определенного периода времени, который настраивается администратором базы данных или определен системными настройками СУБД. Обычно статистика хранится в течение нескольких дней или недель, чтобы обеспечить возможность анализа производительности в течение определенного периода времени и выявить тенденции или проблемы. Старые данные могут архивироваться или удаляться, чтобы освободить место и уменьшить нагрузку на сервер БД.

Сборщики статистики предоставляют общие показатели производительности базы данных, а каждая СУБД имеет свой набор

параметров и метрик для сбора, в зависимости от ее функциональности и возможностей. Наиболее общими для большинства известных СУБД являются следующие параметры:

- Время выполнения запросов – это набор временны́х метрик, таких ка общее время выполнения запроса и время, проведенное в каждом этапе выполнения запроса, например, парсинг, оптимизация, время ожидания блокировок, время операций ввода/вывода и другие.

- Использование ресурсов – эти статистики включают в себя использование процессора, памяти, дискового пространства и других ресурсов сервера базы данных.

- Количество обработанных строк – эти статистики отражают количество строк, которые были обработаны, отфильтрованы или иным образом затронуты при выполнении запроса.

- Статистика блокировок включает в себя информацию о блокировках, которые возникли во время выполнения запросов, такие как блокировки строк, таблиц, транзакций и т. д.

- Статистика ошибок – это сведения о любых ошибках, которые произошли во время выполнения запросов, такие как синтаксические ошибки, ошибки взаимодействия с базой данных и другие.

- Статистика о времени ожидания отражает время, которое запросы провели в ожидании определенных ресурсов, таких как блокировки, сетевые операции, ввод-вывод и т. д.

- Статистика выполнения планов запросов включает информацию о выполнении различных планов запросов, таких как выбранные индексы, соединения, методы соединения и другие аспекты выполнения запроса.

- Статистика кеширования и буферизации содержит сведения об эффективности использования кэшей и буферов в базе данных для ускорения выполнения запросов.

- Статистика репликации имеет место в том случае, если СУБД поддерживает репликацию данных, в этом случае статистики содержат информацию о как задержках, ошибках, статусе репликации и т. д.

Эти и другие первичные данные могут быть дополнены, обобщены или агрегированы еще одним компонентом сбора статистик исполнения запросов – анализатором производительности. Анализатор может проводить дополнительную обработку первичных данных для получения более общего представления о производительности базы данных, выявления общих тенденций в производительности базы данных, выделения узких мест, определения проблемных запросов или операций, а также для предоставления рекомендаций по оптимизации производительности. Обычно анализаторы производительности не сохраняют данные в отдельное хранилище, как это делают сборщики статистики. Вместо этого они обрабатывают данные в реальном времени как временные данные в памяти и предоставляют результаты анализа администраторам баз данных или разработчикам в виде отчетов, графиков или других форматов для принятия решений и оптимизации производительности. Однако некоторые анализаторы могут сохранять результаты своей работы в постоянном хранилище для последующего использования. Например, они могут сохранять историю выполнения запросов, результаты предыдущих анализов или предупреждений о возможных проблемах с производительностью. Эти данные могут использоваться для сравнения производительности в разные периоды времени, выявления тенденций или анализа эффективности принятых мер по оптимизации.

В наиболее популярных серверах баз данных реализованы следующие мощные компоненты для сбора статистики:

Oracle Automatic Workload Repository (AWR) – сборщик статистики, который автоматически собирает и хранит данные о времени выполнения запросов, использовании ресурсов, объеме работы и других метриках

производительности; эти данные сохраняются в специальной статистической базе данных, называемой AWR Repository.

Oracle Enterprise Manager (EM) – инструмент для мониторинга и управления базами данных Oracle, включает в себя различные инструменты анализа производительности, которые могут использоваться для обработки данных, собранных AWR, и предоставления отчетов, графиков и рекомендаций по оптимизации производительности баз данных Oracle.

SQL Server Profiler – инструмент для первичного сбора информации о выполнении запросов и событий в MS SQL Server; позволяет администраторам баз данных отслеживать и анализировать выполнение запросов, выявлять проблемные запросы и мониторить производительность SQL Server в реальном времени.

SQL Server Management Studio (SSMS) – инструмент, предоставляющий различные функции мониторинга, анализа производительности, формирования статистических отчетов и администрирования MS SQL Server.

pg_stat_statements – модуль PostgreSQL, который собирает статистику о выполненных SQL-запросах, включая информацию о времени выполнения, количестве вызовов, используемых ресурсах и другие параметры.

pg_stat_activity – компонент PostgreSQL, предоставляющий статистическую информацию о текущих активных сессиях и выполненных запросах, позволяющий отслеживать текущую нагрузку на сервер баз данных и выполнять мониторинг активности пользователей.

pgBadger и PostgreSQL Enterprise Manager (PEM) – сторонние инструменты для анализа системных журналов PostgreSQL, предоставляющие детальные отчеты о запросах, обобщенных характеристиках производительности и других метриках.

MySQL Performance Schema – встроенный компонент для сбора статистики производительности MySQL; предоставляет информацию о

работе сервера MySQL, включая активность сессий, использование ресурсов, временны́е и статистические характеристики запросов.

MySQL Enterprise Monitor (MEM) – коммерческое решение от Oracle для мониторинга и управления серверами MySQL; предоставляет набор инструментов для мониторинга производительности, включая анализ выполненных запросов, мониторинг ресурсов и оптимизацию конфигурации сервера.

MySQL Query Analyzer – инструмент анализа выполненных запросов в MySQL, который позволяет анализировать производительность запросов, выявлять проблемные запросы и оптимизировать их выполнение.

Это лишь несколько примеров инструментов из длинного списка для сбора статистики и анализа производительности в крупных СУБД. Эти инструменты могут предоставлять разнообразную информацию о времени и характере выполнения запросов, распределении ресурсов и других показателях производительности.

Существует еще один способ получения информации о характере исполнения запросов к БД. Для этих целей некоторые СУБД могут интегрироваться с системными средствами мониторинга операционной системы или вычислительной сети для сбора статистики исполнения запросов. Эти средства могут работать и как первичные сборщики статистики, получая данные непосредственно из СУБД, и как дополнительные инструменты, использующие информацию из системных ресурсов операционной системы для анализа работы базы данных. Примерами подобных систем могут служить Prometheus, Grafana, Zabbix, Nagios, которые устанавливаются на серверах с операционными системами Linux или Window. Некоторые из типичных статистических метрик, которые могут быть собраны и отображены в системе мониторинга, включают:

- среднее время выполнения запросов, 95-й и 99-й процентили времени выполнения запросов, минимальное и максимальное время выполнения запросов;

- общее количество запросов к базе данных за определенный период времени;

- распределение запросов по типам (чтение, вставка, обновление, удаление), количество успешных и неудачных запросов;

- использование центрального процессора, памяти, дискового пространства и других ресурсов базы данных при выполнении запросов;

- информация о блокировках, длительных транзакциях, ожиданиях на блокировки и других проблемах с производительностью;

- эффективность использования кэша базы данных для ускорения выполнения запросов.

Эти метрики могут быть визуализированы в виде графиков, диаграмм или таблиц в системе мониторинга, что позволяет администраторам системы анализировать производительность баз данных, выявлять проблемы и оптимизировать работу системы.

Статистика исполнения запроса может быть использована различными участниками системы, включая администраторов баз данных, разработчиков приложений и оптимизаторов запросов.

Администраторы баз данных могут использовать статистику для мониторинга производительности СУБД и выявления проблемных областей. Например, анализировать время выполнения запросов и количество обращений к диску для идентификации узких мест в производительности, использовать статистику для оптимизации конфигурации базы данных, такой как параметры памяти, размер буферного кэша и настройки индексов, чтобы улучшить производительность системы. Статистические данные об истории выполнения запросов помогают администраторам баз данных понять, какие запросы работают эффективно, а какие могут потреблять слишком много ресурсов или занимать слишком много времени. На основе этой информации они могут принимать решения о том, какие индексы создавать, какие запросы оптимизировать или какие аппаратные ресурсы выделить для улучшения производительности базы данных.

Разработчики могут использовать статистику для оптимизации запросов в своих приложениях. Например, анализировать выполнение запросов и идентифицировать медленные или неэффективные запросы, чтобы улучшить производительность своих приложений, тестировать и контролировать в реальном времени (профилировать) запросы при разработке нового функционала или внесении изменений в существующий код программной системы с БД.

Оптимизаторы запросов, как правило, используют статистику для принятия решений о выборе оптимального плана выполнения запроса. Например, анализировать статистику, чтобы оценить стоимость различных операций и выбрать наилучший способ выполнения запроса, использовать статистику для создания или модификации индексов, чтобы улучшить производительность выполнения запросов.

В целом, статистика исполнения запроса является важным инструментом для мониторинга, анализа и оптимизации производительности приложений с базами данных, предотвращать возможные проблемы и обеспечивать стабильную работу системы