МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ   
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ   
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«ОРЛОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ   
имени И.С. ТУРГЕНЕВА»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

по направлению подготовки 01.03.02 Прикладная математика и информатика

направленность (профиль) Системное программирование   
и компьютерные технологии

Студента Тютюнова Игоря Андреевича шифр 150710

Факультет физико-математический

Тема выпускной квалификационной работы

РАЗРАБОТКА МЕТОДОВ ОПТИМИЗАЦИИ ВЗАИМОДЕЙСТВИЯ ВЕБ-САЙТА С БАЗОЙ ДАННЫХ

Студент \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Тютюнов И.А.

Научный   
руководитель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Федяев Ю.С.

Зав. кафедрой \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Дорофеева В.И.

Орёл 2022

**Содержание**

[ВВЕДЕНИЕ 3](#_Toc102406910)

[ГЛАВА 1. ТЕОРИТИЧЕСКИЕ СВЕДЕНИЯ ОБ ОПТИМИЗАЦИИ РАБОТЫ С БАЗОЙ ДАННЫХ 7](#_Toc102406911)

[1.1. Общая архитектура реляционных СУБД 7](#_Toc102406912)

[1.1.1. Структура данных 7](#_Toc102406913)

[1.1.2. Кластеризация 7](#_Toc102406914)

[1.1.3. Индексирование 7](#_Toc102406915)

[1.1.4. Кэширование в базах данных 7](#_Toc102406916)

[1.2. Оптимизация запросов 7](#_Toc102406917)

[1.2.1. Синтаксическая оптимизация запросов 7](#_Toc102406918)

[1.2.2. Семантическая оптимизация запросов 7](#_Toc102406919)

[1.2.3. Генерация и выбор плана выполнения 7](#_Toc102406920)

[1.2.4. Практика написания эффективных SQL-запросов 7](#_Toc102406921)

[ГЛАВА 2. РАЗРАБОТКА ИНСТРУМЕНТА ДЛЯ РАСПРЕДЕЛЕНИЯ НАГРУЗКИ НА БАЗУ ДАННЫХ В ВЫСОКОНАГРУЖЕННЫХ ИНФОРМАЦИОННЫХ СИСТЕМАХ 8](#_Toc102406922)

[2.2.1. Теоретические сведения об использовании брокера очередей RabbitMQ 8](#_Toc102406923)

[2.2.2. Разработка инструмента для распределения нагрузки на базу данных с использованием брокера очередей RabbitMQ 8](#_Toc102406924)

[ЗАКЛЮЧЕНИЕ 9](#_Toc102406925)

[СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ 10](#_Toc102406926)

[ПРИЛОЖЕНИЕ 11](#_Toc102406927)

# ВВЕДЕНИЕ

**Обоснование выбора темы и ее актуальность**

Нужно обязательно изменить

Кандидат на удаление

На заре повсеместного внедрения информационных технологий в научные, производственные, образовательные и другие сферы человеческой деятельности, очень остро начал подниматься вопрос хранения и эффективной обработки больших объемов данных. Технологии хранения данных, на тот период времени, доставляли множество неудобств разработчикам программного обеспечения, т.к. разработка каждой прикладной программы, требующей сохранения данных на внешний носитель, требовала собственной реализации логической структуры хранения блоков данных.

Концептуально новым шагом в развитии информационных технологий стало внедрение понятия файла, как именованной области данных, и, файловых систем, как регламента, определяющего способ организации и хранения файлов, а также доступа к содержащемся в них данным [5]. Файловые системы связывают физическое расположение данных на носителе информации с прикладными программами, посредствам интерфейса программирования приложений (Application Programming Interface - API), предоставляемым драйвером файловой системы.

Однако эти средства все же не позволяли полностью избавиться от описанных выше проблем. В связи с тем, что структуры данных, которыми оперировали новые программные комплексы, постоянно усложнялись, разработчики программного обеспечения были вынуждены, как и раньше, создавать индивидуальные средства структуризации данных для каждого своего программного продукта. Для того чтобы повысить эффективность разработки программных продуктов, работающих со сложно структурированными данными, было необходимо средство, позволяющее обобщить элементы информационных систем (ИС), отвечающих за работу с этими данными. Это логичное и вполне рациональное стремление положило начало введения нового термина - Базы данных.

Итак, База данных — это совокупность связанных данных, организованных по определенным правилам, предусматривающим общие принципы описания, хранения и манипулирования, независимая от прикладных программ.

Очевидно, что такая методология является целесообразным решением задачи обобщения принципов работы с данными. Изначально данные в базах данных хранились в виде древовидных иерархических структур, в которых малые компоненты являются частью более крупных, которые, в свою очередь, могли быть частью других - таким образом, собиралась единая система данных.

Сегодня SQL является стандартным инструментом управления данными, включающим в себя операторы определения данных (Data Definition Language, DDL), операторы манипуляции данными (Data Manipulation Language, DML), операторы определения доступа к данным (Data Control Language, DCL), операторы управления транзакциями (Transaction Control Language, TCL).

В современном мире трудно переоценить значимость баз данных. Базы данных прочно укрепили свои позиции (и продолжают все активнее применяться) повсеместно в промышленных, образовательных, здравоохранительных, правоохранительных и в др. общественно важных структурах, а также в сферах бизнеса.

Язык SQL, как основной инструмент работы с базами данных, играет первостепенную роль в вопросе разработки СУБД, а также администрирования баз данных. Так как базы данных во многих сферах могут достигать очень больших объемов (как по количеству таблиц, записей и полей, так и по общему объему хранимой информации), то естественно встает вопрос об эффективной, с точки зрения быстродействия, работы с этими данными.

К сожалению, многие разработчики не так часто задумываются о оптимизации SQL-запросов, напрасно полагая, что современное аппаратное обеспечение поможет компенсировать лишние затраты, возникающие при выполнении неэффективных SQL-запросов. Современные СУБД имеют встроенные средства оптимизации запросов, перед их непосредственным выполнением. Понимание принципов работы этих средств необходимо для написания эффективных запросов. Поэтому изучение данного вопроса является актуальным на сегодняшний момент. В данной работе будут рассмотрены встроенные в современные СУБД инструменты оптимизации работы с базой данных, будут описаны общие рекомендации по написанию эффективных SQL-запросов, а также будет разработан инструмент для взаимодействия между информационными системами, позволяющий управлять нагрузкой на базу данных в процессе взаимодействия.

Потребность в инструменте, который позволит обмениваться данными между информационными системами и управлять нагрузкой на базу данных в процессе обмена, возникла в проекте, разработкой и поддержкой которого мне пришлось заниматься.

Проект представляет из себя крупный интернет-магазин, насчитывающий более 14 000 товаров и более 40 000 торговых предложений. Проект работает на базе CMS 1С Bitrix и содержит интеграцию с системой учёта 1С. Интеграция с системой учёта 1С предусмотрена в CMS 1C Bitrix. Обмен данными между 1С и сайтом происходит по HTTP протоколу, передаваемые данные формируются в виде xml файлов, размер которых может превышать несколько сотен мегабайт. Обмен односторонний и инициируется только системой учёта 1С.

Штатный обмен системы учёта 1С и CMS 1C Bitrix в проекте используется для импорта товаров из 1С (информация о товарах, цены и количество товаров на множестве складов).

Большое количество товаров и не оптимальность передачи данных в xml файлах вызывает следующие проблемы:

1. Актуализация данных о товарах на сайте происходит с задержкой из-за большого размера передаваемых xml-файлов;
2. На получение и обработку xml-файлов используется значительное количество ресурсов сервера;
3. В CMS 1C Bitrix есть встроенный механизм кеширования результатов запросов к базе данных. Но штатный обмен 1С и сайта работает таким образом, что при каждом обмене кеш сбрасывается и теряет свою актуальность.

Третья проблема, из списка выше, стала основной, которую необходимо было решить. Ведь самый эффективный механизм оптимизации взаимодействия сайта и базы данных – это кеширование результатов запросов. Но из-за непрерывного обмена сайта и 1С, механизм кеширования результатов запросов не работал должным образом. Кеш результатов запросов создавался при посещении страниц сайта пользователями и был актуальным только до начала очередного обмена системы учёта 1С и сайта – эта особенность в формировании кеша касается именно взаимодействия системы учёта 1С и CMS 1С Bitrix.

В результате, была поставлена задача по оптимизации обмена сайта и системы учёта 1С. Также в рамках этой задачи необходимо было предусмотреть механизм, с помощью которого можно было управлять нагрузкой на базу данных сайта в процессе импорта товаров из 1С, а именно должна быть возможность управлять интенсивностью обработки полученных из 1С данных в зависимости от времени суток. При этом изменение в интенсивности обработки данных из 1С не должна влиять на интенсивность передачи данных из 1С на сайт.

В рамках этой задачи было принято решение разработать инструмент для получения произвольных данных сайтом из сторонних сервисов. В рамках описанного выше проекта сторонним сервисов является система учёта 1С.

Требования к разрабатываемому инструменту:

1. Универсальность – инструмент можно использовать в рамках любого web-проекта, написанного на языке программирования PHP;
2. Минимально-возможное время отправки и получения данных между сервисами при использовании разрабатываемого инструмента;
3. Возможность управлять нагрузкой, которая может оказываться на базу данных при обмене данными с помощью создаваемого инструмента.

**Предмет исследования**

Оптимизация работы веб-сайта.

**Объект исследования**

Оптимизация взаимодействия веб-сайта с базой данных.

**Цель работы**

Разработать инструмент для взаимодействия между информационными системами, позволяющий управлять нагрузкой на базу данных в процессе взаимодействия.

**Основные задачи исследования**

1. Рассмотреть встроенные в современные СУБД инструменты оптимизации работы с базой данных;
2. Описать общие рекомендации по написанию эффективных SQL запросов;
3. Разработать инструмент для взаимодействия между информационными системами, позволяющий управлять нагрузкой на базу данных в процессе взаимодействия.

**Структура работы**

Работа состоит из введения, двух глав, заключения и списка источников.

Во введении рассматривается актуальность работы, ставится цель и формулируются задачи, необходимые для достижения поставленной цели.

Первая глава представляет собой совокупность теоретических данных об изучаемом предмете исследования.

Во второй главе описывается практическое применение инструмента для оптимизации клиентской и серверной части веб-сайтов.

В заключении делаются выводы по проделанной работе.

В конце работы приводится список использованных источников

# ГЛАВА 1. ТЕОРИТИЧЕСКИЕ СВЕДЕНИЯ ОБ ОПТИМИЗАЦИИ РАБОТЫ С БАЗОЙ ДАННЫХ

## Общая архитектура реляционных СУБД

### Структура данных

Опишем общие черты архитектур наиболее распространенных СУБД, и, используемых в них, стандартных решений, направленных на повышение производительности.

Любое расположение данных на диске называется структурой хранения [1, С. 29]. Можно организовать самые разные структуры хранения, обладающие различной производительностью и оптимальные для различных способов использования. Однако, не существует идеальной структуры хранения, которая была бы оптимальна для любых задач. Исходя из этого, можно заключить, что эффективная СУБД должна содержать несколько разных структур хранения для различных частей системы. Кроме того, следует также предусмотреть возможность изменения структуры хранения по мере изменения требований к производительности системы.

В современных СУБД существует два принципиальных подхода к хранению таблиц в оперативной памяти. Первый из них предполагает построчное (кортежное) хранение данных. При этом подходе, каждый кортеж имеет уникальный идентификатор (tuple identifier - tid), который остается неизменным на всем протяжении существования данного кортежа. Физически, каждый идентификатор представляет собой пару чисел, которые соответствуют номеру страницы и описателю идентификатора кортежа. На каждой такой странице памяти существуют две динамические области - область описателей и область, в которой, размещаются кортежи. Образно выражаясь, выделение памяти для описателей и кортежей происходит с разных сторон страницы. Как правило, выделение памяти под область описателей происходит, начиная с младших адресов, с дальнейшим их увеличением, а область для хранения кортежей - со старших адресов, так что новый прибывший кортеж располагается по меньшему адресу относительно текущего. Данная схема проиллюстрирована на рисунке 1.



Рисунок 1. Совместное расположение описателей и кортежей в одной странице

Распространенной ситуацией является увеличение размера кортежа после его обновления. В конечном итоге, это приводит к отсутствию свободной памяти на странице и к невозможности размещения в ней нового описателя кортежа. В этом случае, описатель располагается в другой странице памяти, как показано на рисунке 2.



Рисунок 2. Хранение кортежей и описателей в разных страницах

Такой подход, прежде всего, гарантирует быстрый доступ к целому кортежу. Но кортежное хранение информации не лишено таких недостатков как:

* Дублирование общих значений разных кортежей, что негативно сказывается на количестве потребляемой оперативной памяти;
* Могут потребоваться лишние обмены с внешней памятью, в случае если требуются, лишь отдельные поля целевого кортежа.

Другим подходом к хранению отношений является вертикальное разделение таблиц, при котором все данные хранятся не построчно, как в предыдущем примере, а по столбцам (по полям). Предполагается, что при таком способе хранения данных, все столбцы хранятся отдельно друг от друга. Обработка данных при использовании данного подхода отличается от предыдущего тем, что в момент обработки запроса к БД, считываются лишь существенные столбцы, из которых конструируются кортежи, и, уже над полученными строками, производятся привычные операции. Преимуществом колоночных хранилищ также является то, что любая сущность (то есть столбец) содержит данные одного типа (что следует из самой реляционной модели). Это благоприятно сказывается на производительности некоторых алгоритмов (например, сжатие, поиск и др.), оптимизированных под конкретный тип данных. К недостаткам данного подхода можно отнести более медленное, относительно кортежного метода хранения, считывание целых кортежей из таблицы.

В связи с тем, что, на сегодняшний день, СУБД с использованием колоночного подхода к хранению данных являются, по большей части, экспериментальными и еще не получившими широкое распространение, они не будут приняты в рассмотрение в настоящей работе.

### Кластеризация

Внимание специалистов к структурам хранения и методам доступа вызвано очень низкими скоростными характеристиками внешней памяти. Основным способом повышения производительности является минимизация числа дисковых операций ввода-вывода данных, одним из способов достижения которой является кластеризация.

В основе кластеризации лежит принцип как можно более близкого физического размещения на диске логически связанных между собой и часто используемых данных. Физическая кластеризация данных - чрезвычайно важное условие высокой производительности, что можно продемонстрировать следующим примером. Допустим, что наиболее часто используется хранимая запись r1 страницы p1, для работы с которой также требуется вызывать хранимую запись r2 страницы p2. Тогда возможно возникновение следующих ситуаций:

1. Если страницы р1 и р2 совпадают, то для доступа к записи r2 не потребуется выполнять еще одну физическую операцию ввода-вывода, поскольку нужная страница уже будет находиться в оперативной памяти;
2. Если страницы р1 и р2 не совпадают, но физически размещаются достаточно близко (например, смежные страницы), то для доступа к записи r2 потребуется выполнить еще одну физическую операцию ввода-вывода. Однако, поскольку головка чтения/записи магнитного носителя уже будет находиться в непосредственной близости от нужного положения, время поиска будет весьма небольшим.

Кластеризация внутри СУБД возможна только в том случае, если администратор базы данных организует ее. В современных СУБД часто предусмотрено задание нескольких различных типов кластеризации данных из разных файлов.

### Индексирование

Таблицы в базах данных могут иметь достаточно большое количество неупорядоченных строк. В таких случаях, поиск необходимого кортежа методом полного сканирования таблицы по некоторому условию является малоэффективным, как по времени выполнения, так и по использованию системных ресурсов. Полное сканирование больших таблиц также негативно сказывается на производительности за счет большого количества производимых дисковых операций ввода-вывода.

Индекс — это вспомогательная структура данных, используемая СУБД для доступа к данным. Индекс представляет собой упорядоченный (буквенный или числовой) список столбцов или групп столбцов в таблице. Общей идеей любой организации индекса, поддерживающего прямой доступ по ключу, называемым индексным, и последовательный просмотр в порядке возрастания или убывания значений ключа, является хранение упорядоченного списка значений ключа с привязкой к каждому значению ключа списка идентификаторов кортежей. Физически индекс — это упорядоченный набор значений из индексированного столбца с указателями на места физического размещения исходных строк в структуре базы данных. Когда выполняется обращающийся к индексированному столбцу запрос, СУБД автоматически анализирует индекс для поиска требуемых значений.

Можно провести аналогию между индексом и предметным указателем книги, подобно которому индекс содержит ключи (названия глав в книге) и указатели на физическое расположение кортежей (номера страниц).

Выделяется два типа индексов - простой и составной индекс. Простой индекс — это индекс, определяемый только по одному столбцу таблицы. Для того чтобы при выполнении запроса использовался простой индекс, необходимо использовать индексированное поле в качестве предиката запроса (то есть необходимо указывать ссылку на это поле после SQL оператора WHERE). Составной индекс - индекс, определенный более чем по одному полю. Доступ к составному индексу может осуществляться с помощью одного или нескольких индексных ключей. Это означает, что для запросов, включающих составной индекс, не требуется помещать все индексные ключи в предложение WHERE оператора SQL. Однако, использование более одного индексного поля в условии выборки данных, может благоприятно повлиять на время выполнения запроса, поскольку, в ином случае, будет инициировано сканирование индекса. При сканировании индекса происходит сканирование узлов внутри индекса для считывания нескольких записей данных.

Существуют также уникальные и неуникальные индексы. Уникальный индекс — это индекс, который гарантирует уникальность данных в индексированных столбцах. Данные в индексированном уникальным индексом столбцах должны обладать свойством уникальности (то есть не должны повторятся). Если попытаться вставить в таблицу строку, которая дает дублированное значение индексного ключа в уникальном индексе, то любая СУБД вернет сообщение об ошибке и добавление записи не будет выполнено. Иногда, бывает разумно иметь в таблице уникальный индекс не по одному полю, а по нескольким - для этого служит составной уникальный индекс. Суть этого заключается в том, что уникальным индексным ключом является совокупность значений этих полей.

Индексы могут создаваться не явно СУБД, при создании первичного ключа (PRIMARY KEY), или же средствами языка SQL. Оператор создания индекса имеет следующий синтаксис:

[ UNIQUE ] [ CLUSTERED | NONCLUSTERED ]

INDEX имя\_индекса ON имя\_таблицы(имя\_столбца [ASC|DESC][,...n])

[WITH [PAD\_INDEX]

[[,] FILLFACTOR=фактор\_заполнения]

[[,] IGNORE\_DUP\_KEY]

[[,] DROP\_EXISTING]

[[,] STATISTICS\_NORECOMPUTE]]

[ON имя\_группы\_файлов]

Необязательный UNIQUE позволяет сделать создаваемый индекс уникальным. Взаимоисключающие необязательные параметры CLUSTERED и NONCLUSTERED (параметр по умолчанию) позволяет создавать кластерные и некластерные индексы, соответственно. Параметр ASC предписывает, что индексы (или же сами кортежи, при создании кластерного индекса) должны быть отсортированы по возрастанию, а DESC - по убыванию. Параметр FILLFACTOR осуществляет настройку разбиения индекса на страницы и заметно оптимизирует работу СУБД. Коэффициент FILLFACTOR определяет в процентном соотношении размер создаваемых индексных страниц. При этом имеется обратно пропорциональная зависимость частоты работы с таблицей и коэффициента FILLFACTOR. Параметр PAD\_INDEX определяет заполнение внутреннего пространства индекса и применяется совместно с FILLFACTOR. Параметр DROP\_EXISTING при использовании кластерного индекса определяет его повторное создание, что позволяет предотвратить нежелательное обновление кластерных индексов. Параметр STATISTICS\_NORECOMPUTE определяет функции автоматического обновления статистики для таблицы.

Не смотря на все преимущества использования индексов, они не лишены недостатков. Перечислим основные из них:

* Индексы занимают дополнительное пространство дисковой и оперативной памяти;
* Индексы существенно замедляют выполнение DDL операторов (например, ALTER TABLE), из-за того, что, как правило, это требует пересчета всех индексов.

Разумное применение индексов может существенно повысить производительность SQL-запросов, по выборке данных

### Кэширование в базах данных

Кэширование — это временное хранение часто используемых данных на носителях информации.

В современных базах данных для минимизации операций физического ввода-вывода, используется технология кэширования данных. При этом происходит выделение сравнительно большого сегмента памяти фиксированного размера, одновременно доступного всем сеансам базы данных. Этот сегмент памяти называется буфером блоков кэша [8, С. 30]. Буфер блоков кэша представляет собой связанный список, элементами которого являются блоки данных одного размера, считанные с диска. Любая операция доступа к данным называется логическим вводом-выводом. Обмен данными, находящимся в оперативной памяти, является строго логическим вводом-выводом, в то время как операции дискового ввода-вывода - как логического, так и физического.

Кэш заполняется и поддерживается достаточно простым способом. Каждый раз, когда базе данных необходимо обратится к блоку данных, еще не скопированному в кэш, она запрашивает операцию чтения с диска (физический ввод-вывод) и помещает только что полученный блок в голову буферного списка. Так как длинна списка во время работы базы данных остается фиксированной, добавление блока со стороны головы списка приводит к тому, что блок в хвосте списка удаляется (то есть более не является кэшированным).

Операции в кэше выполняются при помощи указателей в связном списке определенного вида. Новый головной блок — это в действительности та же область памяти, что старый хвостовой блок, в которую записаны новые данные, а указатели перемещены для изменения места блока в списке.

Когда база данных обнаруживает, что нужный блок данных уже находится в списке (что требует строго логического ввода-вывода), она перемещает этот блок из текущего положения в голову списка. Так как блок, участвующий в логическом вводе-выводе, всего лишь перемещается, а не добавляются в список, никакие блоки из хвоста списка не выталкиваются. И снова база данных обрабатывает перемещение логического блока при помощи указателей; на физическом уровне данные в памяти не копируются.

Так как при выполнении логического ввода-вывода блоки перемещаются обратно к голове списка, в итоге кэш становится отсортированным: последние по времени использования (most recently used, MRU) блоки находятся ближе к голове, а блоки с наиболее давним использованием (least recently used, LRU) - к концу списка" [8, С. 30].

## Оптимизация запросов

### Синтаксическая оптимизация запросов

Оптимизация запросов в реляционных СУБД — это способ обработки запросов, когда по начальному представлению запроса вырабатывается наиболее оптимальный план его выполнения путем преобразований этого запроса. Соответствующие преобразования начального представления запроса выполняются специальным компонентом СУБД - оптимизатором запросов, и оптимальность производимого им плана выполнения запроса зависит от критериев, которые в него заложены. Основной задачей встроенных оптимизаторов запросов является преобразование запроса в некоторое внутреннее представление, более удобное для машинных манипуляций, а также поиска наиболее оптимального плана выполнения (то есть выбор такой последовательности операций, которая является наименее требовательной, с точки зрения производительности) этого внутреннего представления.

Несмотря на то, что оптимизаторы запросов современных реляционных СУБД различаются по сложности и реализации, все они следуют одними и теме же основными этапами в выполнении процесса оптимизации запросов. Понимание принципов работы встроенных оптимизаторов, несомненно, является первостепенным в деле написания эффективных запросов.

В этой главе будут рассмотрены принципы работы оптимизаторов запросов реляционных СУБД, а также, на основе этого, будут даны рекомендации по написанию эффективных SQL-запросов.

На начальном этапе синтаксической оптимизации происходит ряд проверок на предмет корректности данного запроса. В том случае, если запрос записан с ошибкой, дальнейшее его выполнение прекращается и СУБД сообщает об ошибке, иначе - происходит переход к следующей фазе.

На этой фазе выполняется преобразование запроса в некоторое внутреннее представление, более удобное для машинных манипуляций. Это полностью исключает из рассмотрения конструкции внешнего уровня и готовит почву для последующих стадий оптимизации.

Обычно внутреннее представление запросов является определенной модификацией абстрактного синтаксического дерева, или дерева запроса.

На следующей стадии происходит преобразование запроса в более удобную, для СУБД, форму, называемую канонической формой, которая требует меньших затрат на выполнение запроса, но дает результат, полностью эквивалентный исходному запросу. Понятие канонической формы употребляется, во многих разделах математики и связанных с ней дисциплин. Каноническая форма может быть определена следующим образом. Пусть Q - множество объектов (запросов), и пусть существует понятие об эквивалентности этих объектов (а именно: запросы q1 и q2 эквивалентны тогда и только тогда, когда дают идентичные результаты). Говорят, что подмножество C множества Q является подмножеством канонических форм для запросов из Q в смысле определенной выше эквивалентности тогда и только тогда, когда каждому объекту q из Q соответствует только один объект c из C. Тогда говорят, что объект с является канонической формой объекта q. Все необходимые свойства, которыми обладает объект q, также характерны и объекту с. Поэтому, чтобы доказать различные необходимые результаты, достаточно изучить менее мощное множество объектов C, а не более мощное множество Q.

Основной целью преобразование запроса в каноническую форму является приведение запроса к более удобоваримому, для конкретной СУБД, виду, который используется на следующих фазах преобразования.

Один из этапов построение канонической формы запроса осуществляется посредствам выполнения простых логических преобразований предикатов запроса (то есть условий, по которым будет осуществляться выборка из БД). Прежде всего, логическое преобразование применимо к простым предикатам, которые имеют следующий вид: "некоторое\_выражение ОС некоторое\_выражение", где некоторое выражение может представлять собой константу, значение определенного поля таблицы, допустимые агрегатные функции и т.п., а ОС - допустимая операция сравнения. Канонические представления предикатов различаются, в зависимости от их конкретных типов. В случае если предикат включает в себя только одно имя поля, он называется простым предикатом.

Пример:

Пусть дан предикат T1.C/(a + 250) ОС (m - 100), где T1.C - имя поля C таблицы T1, ОС - допустимая операция сравнения, а m и a - некоторые переменные. Тогда его каноническое представление будет иметь вид: T1.C ОС (a + 250) \* (m - 100). При использовании не преобразованного предиката в запросе, при сканировании таблицы, приходилось бы вычислять выражение T1.C / (a + 250) для каждой записи в таблице. Рассмотрев приведенный выше пример, становится очевидно, что такое преобразование позволяет минимизировать количество арифметических операций, за счет того, что арифметические выражения, сгруппированные в правой части, вычисляется только один раз, и, далее, сверять получившееся значение со следующими значениями поля T1.C.

Если предикат содержит два имени поля (одной или двух таблиц), то, после приведения к каноническому виду, данный предикат будет иметь вид "имя\_поля ОС некоторое\_выражение", где некоторое\_выражение будет представлять собой выражение, содержащее второе имя поля. Приведем пример такого предиката до преобразования:

30\*T1.A + (a + m)/T2.B ОС k

Тогда каноническое представление этого предиката будет иметь вид:

1.A ОС (T2.B\*k - a - m)/(T2.B\*30)

При приведении предикатов к каноническому представлению применяются вычисления константных выражений и ликвидация логических отрицаний.

Следующий класс логических преобразований связан с приведением к каноническому виду логического выражения, задающего условие выборки запроса. Эти преобразования направлены на выявление противоречащих друг другу предикатов, объединенных конъюнкцией или дизъюнкцией, и, на их дальнейшее упрощение. Например, предикат ((A > B) OR (B >= A)) OR C = n, очевидно, преобразуется в предикат, имеющий следующий вид: TRUE OR C = n. Также, могут применятся логические преобразования предикатов, приводящие к уменьшению их числа. Разумеется, что такие предикаты должны обладать определенной логической взаимосвязью. Проиллюстрируем это на примере. Пусть дан предикат (A < B) AND (B = C\*5). В связи с тем, что этот предикат представляет собой конъюнкцию двух логических выражений, очевидно, что оба выражения должны быть истинными, для истинности всего выражения. Следовательно, подстановка выражения C\*5 в левую часть предиката легитимно. В результате, преобразованный предикат имеет вид: (A < C\*5). Применительно к этому примеру, это упрощение выразилось в ликвидации одного условия и логической операции.

Упрощение запросов с вложенными подзапросами

Вложенный подзапрос — это инструмент создания временной таблицы, содержимое которой извлекается и обрабатывается внешним оператором. К подзапросам применяются следующие правила и ограничения:

* Фраза ORDER BY не используется, хотя и может присутствовать во внешнем подзапросе;
* Список в предложении SELECT состоит из имен отдельных столбцов или составленных из них выражений - за исключением случая, когда в подзапросе присутствует ключевое слово EXISTS. То есть, запрос, выбирающий все столбцы из таблицы (использующий "\*" в качестве списка параметров) не применим;
* По умолчанию имена столбцов в подзапросе относятся к таблице, имя которой указано в предложении FROM. Однако допускается ссылка и на столбцы таблицы, указанной во фразе FROM внешнего запроса, для чего применяются квалифицированные имена столбцов (то есть. с указанием таблицы);
* Если подзапрос является одним из двух операндов, участвующих в операции сравнения, то запрос должен указываться в правой части этой операции.

Сегодня SQL позволяет использовать вложенные подзапросы, используемые в логическом условии выборки. При этом вложенные подзапросы выступают в роли одной из логических частей предиката. Стандарт языка SQL не накладывает никаких ограничений на глубину вложенных подзапросов, что свидетельствует о теоретической неограниченности их числа.

Рассмотрим пример:

SELECT Tl.A FROM TlTl.B IN (SELECT T2.B FROM T2 WHERE Tl.C = T2.D)

Данный запрос производит выборку данных из таблицы T1 значений поля A, кортежи которой удовлетворяют условию, при котором множество значений поля B, таблицы T1, пересекается с множеством значений поля B, таблицы T2, образуемое при выполнении условия равенства значений поля C, таблицы T1, и поля D, таблицы T2. При такой семантике запроса, возможен единственный алгоритм выполнения - вложенный подзапрос будет вычисляться всякий раз при проверке условия предиката. Поэтому нет необходимости доказывать тот факт, что вложенные подзапросы не всегда являются самым эффективным решением, а также, что, в большинстве случаев, можно с успехом обойтись без их использования. Рассмотренный запрос может быть преобразован к следующему виду:

Tl.A FROM Tl.R2

WHERE Tl.B = T2.B AND Tl.C = R2.D

Результат выполнения преобразованного запроса является полностью эквивалентным результату первоначального запроса. Однако, несмотря на это, время выполнения этих запросов может существенно разниться. Как правило, запрос, в состав которого входит вложенный подзапрос, будет иметь большее время выполнения.

Однако, из вышеописанного не следует, что вложенные подзапросы не стоит применять. Иногда без них довольно сложно обойтись. Покажем это на примере следующей задачи:

Определить клиентов, совершивших сделки с максимальным количеством товара. Решить эту задачу можно следующим, вполне лаконичным и сравнительно простым, запросом:

SELECT Clients.Name FROM ClientsJOIN Deal ON Clients.ClientsID = Deal.ClientsIDDeal.Number = (SELECT Max(Deal.Number) FROM Deal)

### Семантическая оптимизация запросов

Существуют также преобразования, использующие семантику конкретной базы данных, а не языка запросов. Любое преобразование выполняется вне зависимости от конкретной базы данных. При этом база данных хранит не просто совокупность взаимосвязанных таблиц, а некоторую семантическую информацию, определяющую целостность базы данных. Поскольку СУБД гарантирует целостность базы данных, то ограничения целостности можно рассматривать как аксиомы, в окружении которых формируются запросы к базе данных [1, 4].

Семантическая оптимизация запросов - процесс проверки корректности и преобразования синтаксического дерева запроса в форму, пригодную для дальнейших шагов оптимизации.

Для получения более полного определения семантической оптимизации, обратимся к книге Кристофера Дейта, Введение в системы баз данных [2]: "Преобразование, которое является допустимым только в силу того, что имеется конкретное установленное ограничение целостности, называют семантическим преобразованием, а оптимизацию, достигаемую в результате подобных преобразований, - семантической оптимизацией. Семантическую оптимизацию можно определить как процесс преобразования одного запроса в другой, качественно отличный запрос, который, тем не менее, гарантирует результат, идентичный результату исходного запроса благодаря тому, что обрабатываемые данные удовлетворяют определенному ограничению целостности".

Рассмотрим абстрактное выражение, которое является соединением, относящимся к соединениям типа "внешниий-к-согласованному-потенциалъному-ключу". В этом соединении внешнему ключу (поле таблицы, предназначенное для хранения значения первичного ключа другой таблицы с целью организации связи между этими таблицами) в отношении T1 ставится в соответствие потенциальный ключ (т.е., атрибут, который уникально идентифицирует отдельные экземпляры сущности) отношения T2. Следовательно, кортеж в отношении T1 связан с определенным кортежем в отношении T2. Таким образом, из каждого кортежа в отношении T1 в общий результат поступает значение некоторого атрибута A. Другими словами, выполнение данного соединения является, в данном случае, излишним. Данное выражение можно заменить обычной выборкой данных по условию соответствия ключей обоих отношений. Важно понимать, что в принципе любое условие целостности может быть использовано для семантической оптимизации (если это условие не отсрочено и в данный момент действует на базу данных). При семантической оптимизации возможны и более интеллектуальные преобразования, опирающееся на семантику (смысл) запроса, и, на хранящуюся в базе данных, семантическую информацию. Например, предположим, что в базе данных компьютерных комплектующих и поставщиков установлено ограничение, по которому все принтеры должны находиться на оптовых складах в Москве. Предположим, что необходимо определить поставщиков, которые поставляют только принтеры и находятся в том же городе, где хранится хотя бы одна поставляемая ими деталь. Не трудно определить, что эта задача обладает избыточностью информации. Современные оптимизаторы в состоянии оптимизировать запрос, проверяющий все условия задачи, с помощью семантической информации о том, что принтеры могут находиться только на Московских складах, к виду: "Определить поставщиков из Москвы, поставляющих исключительно принтеры".

### Генерация и выбор плана выполнения

Рассмотренные выше методы оптимизации преследуют цель минимизации количества действий в выражениях, и, если это необходимо, замены некоторых выражений на эквивалентные, но имеющие меньшее время выполнения. После этих преобразований необходимо сгенерировать последовательность элементарных операций, которая, в конечном итоге, даст корректный результат. Процедурным представлением или планом выполнения запроса называется последовательность операций, необходимых для получения результата SQL-запроса в реляционной СУБД. Как правило, любой запрос может иметь несколько альтернативных планов выполнения. Критерием выбора плана выполнения запроса является минимизация стоимости выполнения, которая определяется временем выполнения запроса.

В общем случае, стадия оптимизации, рассмотренная в этой главе, делится на два следующих этапа:

1. найти набор потенциально возможных планов выполнения для данного запроса, руководствуясь внутренним представлением запроса и информаций, характеризующей управляющие структуры базы данных (например, индексы);

2. выбрать из возможных планов выполнения, отобранных на стадии из предыдущего пункта, план, имеющий наименьшую стоимость выполнения.

При поиске планов выполнения, оптимизатор запросов должен генерировать достаточный набор планов, чтобы в нем содержался оптимальный план, и, в то же время, достаточно умеренный, чтобы удержать накладные расходы на приемлемом уровне. Приведем пример простого запроса и генерации для него планов выполнения:

SELECT \* FROM T1T1.A > 20 AND T1.B < 10000

Для этого запроса могут быть сгенерированы следующие планы выполнения:

1. последовательное сканирование таблицы T1 с выборкой кортежей удовлетворяющих условию предиката;
2. сканирование по индексу поля A, в случае если оно индексировано, с проверкой полученных кортежей на соответствие условию T1.B < 10000;
3. сканирование по индексу поля B, в случае если оно индексировано, с проверкой полученных кортежей на соответствие условию A > 20;
4. сканирование по индексам полей A и B, в случае использования по этим полям кластерного индекса.

Оптимизатор использует так называемый стоимостной метод для определения наилучшего плана выполнения запроса. Оптимизатор оценивает стоимость или затраты на работу процессора и операции ввода-вывода для различных методов доступа к данным и операций над ними. Стоимость работы процессора оценивается в миллисекундах. Стоимость ввода-вывода, которая представляет собой количество операций ввода-вывода в секунду, также преобразуется в миллисекунды. Такой подход позволяет проводить сравнение нагрузки на процессор и подсистему ввода-вывода. Производительность методов доступа к данным и операций объединения зависит от размера и распределения данных. Информация о данных представлена в виде статистики таблиц и индексов. Запрос разбивается на ряд небольших индивидуальных шагов. Наименьший шаг - это доступ к данным одной таблицы, который выбирается из всех возможных методов доступа. При этом исследуются все возможности выбора данных из таблицы, а также методы объединения между любыми двумя таблицами или таблицей и промежуточным результатом. Оптимизатор выбирает наиболее простой и эффективный план выполнения.

В реальной обстановке оптимизатор выбирает план выполнения запроса на основе вычисления затрат на выполнение целого ряда операций, которые помимо простого использования процессора и подсистемы ввода-вывода включают следующие факторы:

* операции сравнения;
* перемещение данных;
* повторный доступ к той же таблице;
* методы объединения данных;
* использование буферов ввода/вывода;
* распределение предикатов.

Также существуют методы выбора плана выполнения, основанные на анализе синтаксиса запроса или жестко установленных правилах выбора.

### Практика написания эффективных SQL-запросов

Несмотря на то, что существующие оптимизаторы современных СУБД достаточно хорошо справляются со своей задачей и продолжают совершенствоваться, это вовсе не означает что программисту, пишущему запросы к базе данных, не нужно задумываться о эффективности этих запросов. Это особенно важно в том случае, если база данных весьма объемна. Наиболее ощутимое влияние на производительность оказывают условия выборки и операторы сортировки и группировки.

Написание запросов к нетривиально спроектированной базе данных, в большинстве случаев, требует использования в предикатах нескольких условий, объединенных логическими операциями конъюнкции или дизъюнкции.

Не последнюю роль в вопросе оптимизации играет порядок следования условий, объединенных операторами AND и OR. Рассмотрим выражение: A OR B OR C, где A, B и C - некоторые условия. Очевидно, что выражение в целом будет истинно, если хотя бы одно условие будет являться истинным. Большинство СУБД не производят проверку, в подобных выражениях, остальных условий, если текущее оказывается истинным. Иными словами, при истинности условия A, условия B и C проверяться не будут. Поэтому, в таких выражениях, разумно располагать условия в порядке убывания вероятностей их истинности. При использовании логического И (оператор AND), наоборот - рекомендуется располагать условия в порядке возрастания вероятностей их истинности.

При использовании группировки, для того чтобы запрос был эффективным, существует только одна рекомендация - необходимо всегда минимизировать число полей для группировки.

# ГЛАВА 2. РАЗРАБОТКА ИНСТРУМЕНТА ДЛЯ РАСПРЕДЕЛЕНИЯ НАГРУЗКИ НА БАЗУ ДАННЫХ В ВЫСОКОНАГРУЖЕННЫХ ИНФОРМАЦИОННЫХ СИСТЕМАХ

## 2.2.1. Постановка задачи

### 2.2.2. Теоретические сведения об использовании брокера очередей RabbitMQ

### 2.2.3. Разработка инструмента для распределения нагрузки на базу данных с использованием брокера очередей RabbitMQ

# ЗАКЛЮЧЕНИЕ

# СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ

# ПРИЛОЖЕНИЕ