Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей
Кафедра информатики
Дисциплина: Операционные среды и системное программирование

К защите допустить:

И.О. Заведующего кафедрой информатики
_____С. И. Сиротко

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовому проекту
на тему

РЕАЛИЗАЦИЯ ПРОСТОЙ БАЗЫ ДАННЫХ. ПРОГРАММНЫЙ МОДУЛЬ, ПЕРЕНОСИМЫЙ БЕЗ ИНСТАЛЛЯЦИИ. ХРАНЕНИЕ ДАННЫХ СРЕДСТВАМИ ФАЙЛОВОЙ СИСТЕМЫ С ДОСТУПОМ ПОСРЕДСТВОМ УПРОЩЕННОГО ЯЗЫКА ЗАПРОСОВ.

БГУИР КП 1-40 04 01 020 ПЗ

Студент А. В. Скворцов

Руководитель Н. Ю. Гриценко

Минск 2024

СОДЕРЖАНИЕ

Введение 6
1 Базы данных 7
1.1 Общая информация 7
1.2 Классификация баз данных 10
1.3 Примеры систем управления базами данных 14
2 Платформа программного обеспечения 18
2.1 Linux 18
2.2 Устройство диска в Linux 19
2.3 Файловая система 22
3 Структурированный язык запросов 24
3.1 Общая информация про язык запросов 24
3.2 Достоинства и недостатки SQL 25
4 Проектирование функциональных возможностей программы 28
5 Взаимодействие с разработанной базой данных 30
Заключение 32
Список литературных источников 33
Приложение А (обязательное) Листинг кода 34
Приложение Б (обязательное) Функциональная схема алгоритма 43
Приложение В (обязательное) Блок-схема алгоритма 44
Приложение Г (обязательное) Скриншоты реализации 45
Приложение Д (обязательно) Ведомость документов 46

ВВЕДЕНИЕ

Технология баз данных появилась почти полвека назад и с тех пор не только оказала огромное влияние на развитие информационных технологий, но и кардинально изменила методы работы многих организаций и предприятий. В наше время сложно найти такую компанию, в которой не использовались бы базы данных, точнее, информационные системы, основанные на базах данных. Возросший поток информации приводит к необходимости разработки новых приемов ее осмысления и обработки. Накопленный опыт и развитие программного обеспечения позволяет переосмыслить такую традиционную область обработки информации как хранение и управление данными. Новый подход к организации процесса обработки информации приводит к понятию база данных и методам работы с ней.

Целью данного курсового проекта является разработка программного модуля, который позволит реализовать простую базу данных с хранением данных средствами файловой системы и обеспечит доступ к данным через упрощенный язык запросов. Основными задачами работы являются создание программного модуля, который будет обеспечивать функциональность базы данных. Модуль должен быть переносимым, то есть его можно использовать на различных платформах без необходимости установки дополнительных компонентов.

Так же в базе данных будет разработан механизм хранения данных, используя файловую систему компьютера. Будет рассмотрен язык запросов, который применяют, чтобы работать с базами данных, структурированных особым образом, и будет представлена его упрощенная реализация для управления разработанным модулем. Язык будет являться легким для понимания и использования, а также будет поддерживать основные операции для работы с данными, такие как добавление, обновление, удаление и выборка данных.

1 БАЗЫ ДАННЫХ

1.1 Общая информация

Высокая ценность информации признавалась во все времена, но только во второй половине XX в., когда появилась возможность эффективного управления действительно большими объемами информации, она стала важнейшим стратегическим ресурсом, обслуживание которого потребовало создания специализированных программно-технических средств — автоматизированных информационных систем (АИС).

АИС обеспечивают надежное хранение и оперативное обновление информации, а также ее поиск, извлечение и аналитическую обработку по запросам потребителей. При всем разнообразии архитектур, решаемых задач и условий использования АИС в структуре их программного обеспечения (ПО) явно выделяют два относительно автономных компонента: подсистему хранения данных и подсистему обработки информации. Основу подсистемы хранения составляют база данных (БД) — программно-реализованная информационная модель предметной области АИС, управляемая специализированной программной системой (СУБД), и подсистема обработки информации, включающая множество прикладных программ, получающих доступ к СУБД для чтения и/или модификации информации, хранимой в базе данных.

АИС относятся к категории социотехнических систем, важнейшим компонентом которых, наряду с аппаратным и программным обеспечением, являются пользователи — разработчики, администраторы и конечные пользователи, участвующие в создании или эксплуатации АИС на различных стадиях ее жизненного цикла.

Различают АИС, основанные на знаниях, и АИС, основанные на данных. К первым можно отнести, например, экспертные системы (ЭС), интеллектуальные системы поддержки принятия решений (СППР) и т.п. Ко вторым – всевозможные прикладные системы, которые сейчас активно используются и на предприятиях, и в учреждениях. Такие прикладные системы применяются очень широко.

Существуют две основные предпосылки создания таких систем:

1 Разработка методов конструирования и эксплуатации систем, предназначенных для коллективного использования.

2 Возможность собирать, хранить и обрабатывать большое количество данных о реальных объектах и явлениях, то есть оснащение этих систем "памятью".

Массив данных общего пользования в системах, основанных на данных, называется базой данных. База данных (БД) является моделью предметной области информационной системы.

На заре развития вычислительной техники обрабатываемые данные являлись частью программ: они располагались сразу за кодом программы в так называемом сегменте данных (рисунок 1.1, а). Следующим шагом стало хранение данных в отдельных файлах (рисунок 1.1, б). Недостатком этих двух подходов являлась зависимость программ от данных: сведения о структуре данных включались в код программы. При изменении структуры данных необходимо было вносить изменения в программу.

Логичным продолжением этой эволюции является перенос описания данных в массив данных (рисунок 1.1, в). Это позволило обеспечить независимость данных от программ.

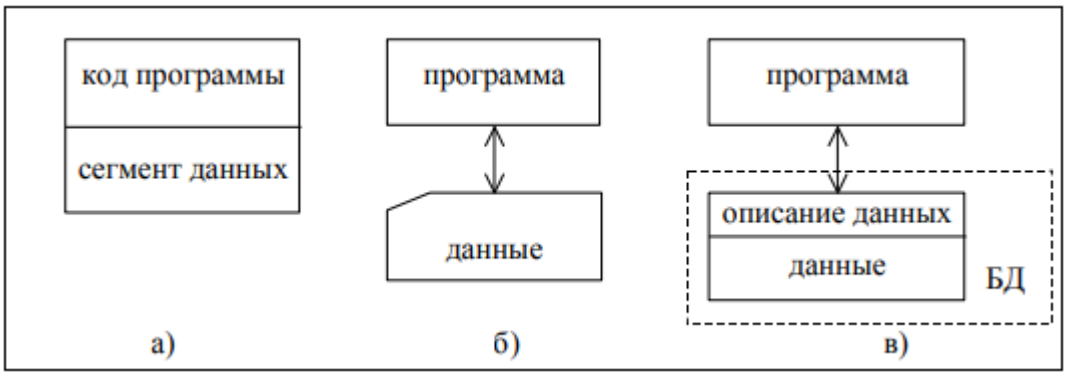

Рисунок 1.1 – Развитие принципов разработки баз данных

База данных – это организованная коллекция данных, хранящихся и обрабатываемых в компьютерной системе. Она предназначена для эффективного хранения, управления и извлечения информации. Базы данных используются в различных областях, включая бизнес, науку, образование и государственное управление.

Базы данных позволяют эффективно хранить большие объемы данных, обеспечивать структурированный доступ к информации и обрабатывать данные с помощью запросов и операций. Они позволяют пользователям добавлять, изменять, удалять и извлекать данные в удобном формате. Базы данных также поддерживают функции безопасности, целостности данных и резервного копирования, чтобы обеспечить сохранность и доступность информации.

Система управления базами данных (СУБД) – это комплекс языковых и программных средств, предназначенный для создания, ведения и совместного использования базой данных многими пользователями.

В процессе исследований, посвященных тому, как именно должна быть устроена СУБД, американским комитетом по стандартизации ANSI (American National Standards Institute) сформулирована трехуровневая система организации БД (рисунок 1.2).

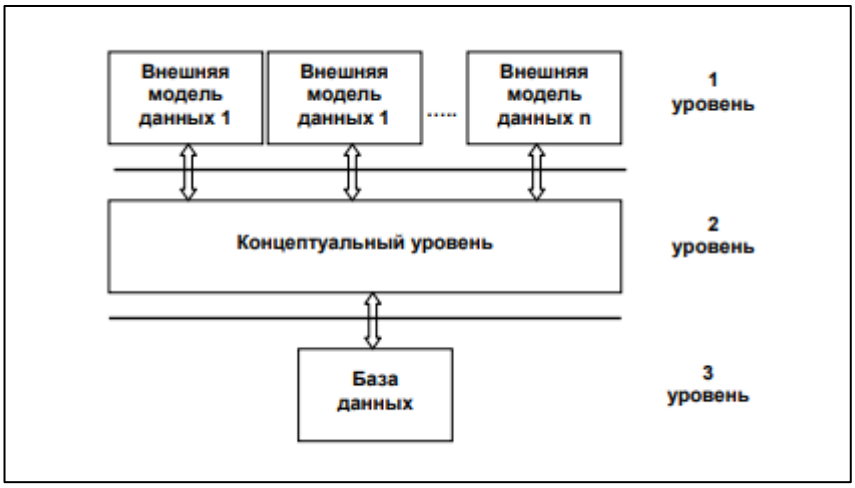

Рисунок 1.2 – Трехуровневая модель системы управления базами данных

1 *Уровень внешних моделей* – самый верхний уровень, где каждая модель имеет свое «видение» данных. Этот уровень определяет точку зрения на БД отдельных приложений. Каждое приложение видит и обрабатывает только те данные, которые необходимы именно этому приложению.

2 *Концептуальный уровень* – центральное управляющее звено, здесь БД представлена в наиболее общем виде, который объединяет данные, используемые всеми приложениями, работающими с данной БД. Фактически концептуальный уровень отражает обобщенную модель предметной области, для которой создавалась БД.

3 *Физический уровень* – собственно данные, расположенные в файлах или в страничных структурах, расположенных на внешних носителях информации. Эта архитектура позволяет обеспечить логическую (между уровнями 1 и 2) и физическую (между уровнями 2 и 3) независимость при работе с данными. Первая предполагает возможность изменения одного приложения без корректировки других приложений, работающих с этой же базой данных. Физическая независимость предполагает возможность переноса хранимой информации с одних носителей на другие при сохранении работоспособности всех приложений, работающих с данной базой данных.

1.2 Классификация баз данных

Важнейшим достоинством применения баз данных является обеспечение независимости данных от прикладных программ. Это даст возможность пользователям не заниматься проблемами представления данных на физическом уровне: размещения данных в памяти, методов доступа к ним и т.д. Такая независимость достигается поддерживаемым системами управления базами данных многоуровневым представлением данных в базе на логическом (пользовательском) и физическом уровнях. Благодаря системам управления базами данных и наличию логического уровня представления данных обеспечивается отделение концептуальной модели базы данных от ее физического представления в памяти ЭВМ.

Логическую структуру хранимых в базе данных называют моделью представления данных. К основным моделям представления данных относятся следующие*: реляционная, иерархическая, сетевая, объектно-ориентированная.* Некоторые системы управления базами данных могут одновременно поддерживать несколько моделей данных.

Реляционная модель является простейшей и наиболее привычной формой представления данных в виде таблицы. В теории множеств для нее имеется развитый математический аппарат – реляционное исчисление и реляционная алгебра. Достоинством реляционной модели является сравнительная простота инструментальных средств ее поддержки, недостатком – жесткость структуры данных (невозможность, например, задания строк таблицы произвольной длины) и зависимость скорости ее работы от размера базы данных.

Реляционная модель данных предложена сотрудником фирмы IBM Эдгаром Коддом и основывается на понятии отношение (relation). Отношение представляет собой множество элементов, называемых кортежами.

Наглядной формой представления отношения является привычная для человеческого восприятия двумерная таблица. Таблица имеет строки (записи) и столбцы (поля). Каждая строка таблицы имеет одинаковую структуру и состоит из полей. Строкам таблицы соответствуют кортежи, а столбцам – атрибуты отношения. С помощью одной таблицы удобно описывать простейший вид связей между данными, а именно деление одного объекта (явления, сущности, системы и проч.), информация о котором хранится в таблице, на множество подобъектов, каждому из которых соответствует строка или запись таблицы. При этом каждый из подобъектов имеет одинаковую структуру или свойства, описываемые соответствующими

значениями полей записей. Например, таблица может содержать сведения о группе обучаемых, о каждом из которых известны следующие характеристики: фамилия, имя и отчество, пол, возраст и образование. Поскольку в рамках одной таблицы не удается описать более сложные логические структуры данных из предметной области, применяют связывание таблиц.

Физическое размещение данных в реляционных базах на внешних носителях легко осуществляется с помощью обычных файлов.

Достоинство реляционной модели данных заключается в простоте, понятности и удобстве физической реализации на ЭВМ. Именно простота и понятность для пользователя явились основной причиной их широкого использования. Проблемы эффективности обработки данных этого типа также оказались технически вполне разрешимыми.

Основными недостатками реляционной модели является отсутствие стандартных средств идентификации отдельных записей и сложность описания иерархических и сетевых связей.

Постреляционная модель данных представляет собой расширенную реляционную модель, снимающую ограничение неделимости данных, хранящихся в записях таблиц.

Постреляционная модель данных допускает многозначные поля - поля, значения которых состоят из подзначений. Набор значений многозначных полей считается самостоятельной таблицей, встроенной в основную таблицу.

Достоинством постреляционной модели является возможность представления совокупности связанных реляционных таблиц одной постреляционной таблицей. Это обеспечивает высокую наглядность представления информации и повышение эффективности ее обработки.

Недостатком постреляционной модели является сложность решения проблемы обеспечения целостности и непротиворечивости хранимых данных. Для описания структуры иерархической модели используется тип данных – «дерево». Тип «дерево» является составным и включает в себя подтипы («поддеревья»), каждый из которых, в свою очередь, является типом «дерево». Каждый из типов «дерево» состоит из одного «корневого» типа и упорядоченного набора подчиненных типов (рисунок 1.3).

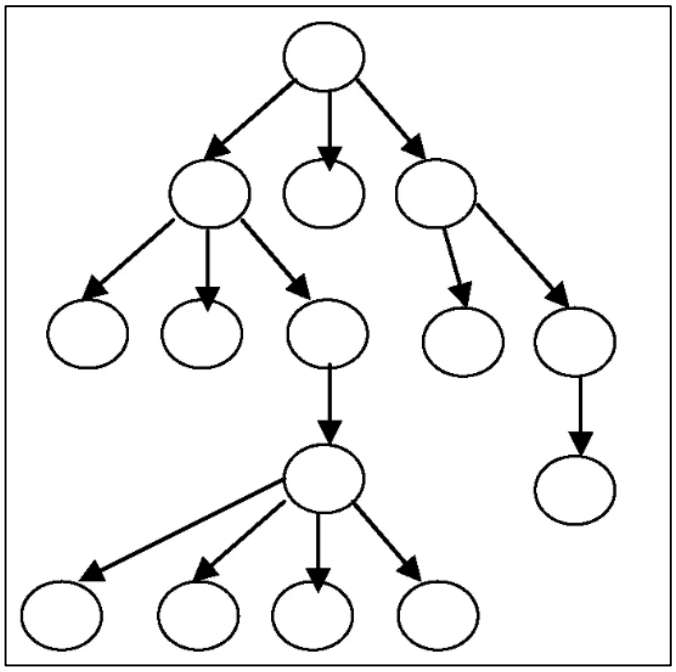

Рисунок 1.3 – Иерархическая модель данных

Корневым называется тип, который имеет подчиненные типы и сам не является подтипом. Подчиненный тип (подтип) является потомком по отношению к типу, который выступает для него в роли предка (родителя). Потомки одного и того же типа являются близнецами по отношению друг к другу. Каждый из элементарных типов, включенных в тип «дерево», является простым или составным типом «запись». Поля записей хранят собственно числовые или символьные значения, составляющие основное содержание базы данных.

Обход всех элементов иерархической базы данных обычно производится сверху вниз и слева направо. Иерархическая модель предполагает наличие связей между данными, имеющими какой-либо общий признак.

В иерархической модели такие связи могут быть отражены в виде дерева-графа, где возможны только односторонние связи от старших вершин к младшим. Это облегчает доступ к необходимой информации, но только если все возможные запросы отражены в структуре дерева. Никакие иные запросы удовлетворены быть не могут.

К достоинствам иерархической модели данных относятся эффективное использование памяти ЭВМ и неплохие показатели времени выполнения основных операций над данными. Иерархическая модель данных удобна для работы с иерархически упорядоченной информацией. Недостатком иерархической модели является ее громоздкость для обработки информации с

достаточно сложными логическими связями, а также сложность понимания для обычного пользователя.

Сетевая модель данных позволяет отображать разнообразные взаимосвязи элементов данных в виде произвольного графа, обобщая тем самым иерархическую модель. Для описания схемы сетевой базы данных используется две группы типов: «запись» и «связь». Тип «связь» определяется для двух типов «запись»: предка и потомка. Переменные типа «связь» являются экземплярами связей (рисунок 1.4).

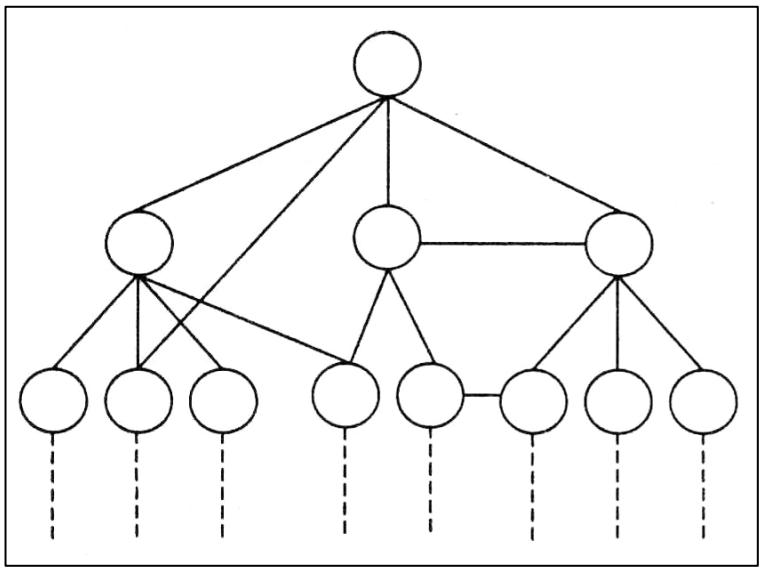

Рисунок 1.4 – Иерархическая модель данных

Сетевая БД состоит из набора записей и набора соответствующих связей. На формирование связи особых ограничений не накладывается. Если в иерархических структурах запись-потомок могла иметь только одну запись-предка, то в сетевой модели данных запись-потомок может иметь произвольное число записей-предков (сводных родителей).

Достоинством сетевой модели данных является возможность эффективной реализации по показателям затрат памяти и оперативности. В сравнении с иерархической моделью сетевая модель предоставляет большие возможности в смысле допустимости образования произвольных связей. Недостатком сетевой модели данных является высокая сложность и жесткость схемы БД, построенной на ее основе, а также сложность для понимания и выполнения обработки информации в БД обычным пользователем. Кроме того, в сетевой модели данных ослаблен контроль целостности связей вследствие допустимости установления произвольных связей между записями.

В объектно-ориентированной модели при представлении данных имеется возможность идентифицировать отдельные записи базы. Между записями базы данных и функциями их обработки устанавливаются взаимосвязи с помощью механизмов, подобных соответствующим средствам в объектно-ориентированных языках программирования.

Для выполнения действий над данными в рассматриваемой модели БД применяются логические операции, механизмы инкапсуляции, наследования и полиморфизма. Инкапсуляция ограничивает область видимости имени свойства пределами того объекта, в котором оно определено. Смысл такого свойства будет определяться тем объектом, в который оно инкапсулировано. Наследование, наоборот, распространяет область видимости свойства на всех потомков объекта. Если необходимо расширить действие механизма наследования на объекты, не являющиеся непосредственными родственниками (например, между двумя потомками одного родителя), то в их общем предке определяется абстрактное свойство. Полиморфизм означает способность одного и того же программного кода работать с разнотипными данными. Другими словами, он означает допустимость в объектах разных типов иметь методы с одинаковыми именами.

Во время выполнения объектной программы одни и те же методы оперируют с разными объектами в зависимости от типа аргумента. Основным достоинством объектно-ориентированной модели данных в сравнении с реляционной является возможность отображения информации о сложных взаимосвязях объектов. Объектно-ориентированная модель данных позволяет идентифицировать отдельную запись базы данных и определять функции их обработки. Недостатками объектно-ориентированной модели являются высокая понятийная сложность, неудобство обработки данных и низкая скорость выполнения запросов.

1.3 Примеры систем управления базами данных

Самая популярная система управления базами данных в мире – PostgreSQL. Об этом говорят не только результаты опросов разработчиков и предпринимателей, но и количество вакансий, в которых владение ею указывается среди необходимых навыков.

PostgreSQL — это свободно распространяемая объектно-реляционная система управления базами данных (СУБД) с открытым исходным кодом, написанном на языке C. Для объяснения, почему PostgreSQL является самой популярной в мире СУБД следует рассмотреть ряд ее достоинств, относительно других:

1 **Расширяемость и богатый набор типов данных.** Помимо стандартных, в PostgreSQL есть типы для геометрических расчётов, сетевых адресов и полнотекстового поиска. Мощная система расширений позволяет добавлять новые возможности и типы данных. Кроме того, администратор может писать свои функции и процедуры на Python, PHP, Java, Ruby и многих других языках программирования, а также загружать модули на языке C из центрального репозитория PGXN.

2 **Масштабируемость.** Для повышения производительности и масштабируемости в PostgreSQL используются разные виды блокировок на уровне таблиц и строк; шесть видов индексов, среди которых B-дерево и обобщённое дерево поиска (GiST) для полнотекстового поиска; наследование таблиц — для быстрого создания таблиц на основе имеющейся структуры. Вы также можете анализировать скорость выполнения запросов с помощью команды explain, очищать диск от мусора командой vacuum и собирать статистику по таблицам с analyze.

3 **Кросс-платформенность.** PostgreSQL поддерживается всеми популярными операционными системами, среди которых различные дистрибутивы Linux и BSD, macOS, Windows, Solaris и другие. Интерфейсы для этой СУБД реализованы практически во всех языках программирования.

4 **Безопасность.** В PostgreSQL есть множество инструментов для защиты данных от злоумышленников: пароль, Kerberos, LDAP, GSSAPI, SSPI, PAM и другие. Она позволяет управлять доступом к объектам БД на нескольких уровнях — от базы данных до отдельных столбцов, а также шифровать данные на аппаратном уровне.

5 **Возможности NoSQL.** Помимо стандартных форматов, PostgreSQL поддерживает XML, а с девятой версии — JSON и JSONB. Последний позволяет не разбирать JSON-документ перед записью в базу данных, что существенно ускоряет его сохранение в БД.

6 **Свободное распространение и открытый код.** Проект распространяется под лицензией BSD, что позволяет бесплатно его использовать, модифицировать и распространять. Исходный код можно посмотреть на зеркале официального Git-репозитория.

7 **Живое сообщество и исчерпывающее официальное руководство.** PostgreSQL поддерживается большим активным сообществом пользователей и разработчиков, которые получают новости о проекте через еженедельные рассылки, обсуждают вопросы администрирования и ведут дискуссии. Официальная документация регулярно обновляется и на момент публикации содержит около 3000 страниц подробных и понятных инструкций.

Но даже при всех этих достоинствах, PostgreSQL, как и другие системы управления базами данных, обладаем рядом недостатков, но даже эти недостатки могут проявиться только в некоторых сценариях использования:

1 **Сложность настройки.** Наверняка вы уже оценили обилие возможностей, которые даёт Postgres. Очевидно, что оно влечёт за собой разнообразие конфигураций, что может создавать сложности для начинающих пользователей. Настройка базы данных требует глубокого понимания архитектуры и параметров.

2 **Высокое потребление ресурсов.** PostgreSQL может потреблять больше ресурсов (памяти и процессорного времени) по сравнению с некоторыми другими СУБД. Особенно это заметно при работе с большими объёмами данных и сложными запросами.

3 **Отсутствие некоторых функций.** В сравнении с некоторыми коммерческими СУБД PostgreSQL может слегка отставать в функциональности.

MongoDB — это документоориентированная нереляционная СУБД, которая распространяется по лицензии SSPL и имеет открытый исходный код. Её создатели были довольно авторитетными в мире IT разработчиками. В частности, именно они основали в начале 2000-х DoubleClick — одну из первых компаний, специализирующихся на интернет-рекламе, с фантастической по тем временам скоростью показа до 400 000 объявлений в секунду.

В 2005 году Мерриман с коллегами выгодно продали её Google и смогли вплотную приступить к давно волновавшей их проблеме. Существовавшие в середине нулевых базы данных не имели чёткой структуры, хранившиеся в них информационные фрагменты не были связаны друг с другом, постоянно возникали проблемы с масштабируемостью и гибкостью. Так возникла фирма 10gen, позже переименованная в честь своего флагманского продукта в MongoDB Inc.

В обычных реляционных базах данных информация хранится в виде взаимосвязанных таблиц. Их структура жёстко задана, и поменять её непросто. Строки каждой таблицы имеют одинаковый набор полей, данные обрабатывают с помощью запросов на языке SQL. Эти базы наглядны, но не всегда удобны — например, в тех случаях, когда вам нужно хранить информацию без определённой структуры: представить её в виде двумерных таблиц нельзя.

В MongoDB всё устроено немного по-другому. Базы состоят из коллекций и документов — иерархических структур, содержащих пары «ключ

— значение» (поля). Если проводить аналогии с реляционной базой, коллекции при таком способе хранения соответствуют таблицам, а документы — строкам. Информация отформатирована в BSON — двоичной кодировке JSON-подобных документов. Это позволяет поддерживать данные типа Date и двоичных файлов, что невозможно в JSON.

Конечно, учитывая тот факт, что реляционные базы данных являются наиболее частым выбором пользователей, MongoDB так же имеет ряд своих преимуществ:

– Гибкая система хранения информации: в приложениях не обязательно преобразовывать объекты в элементы таблиц, не нужно пересоздавать схему базы при изменении структуры данных, например при добавлении нового поля. В документах хранится информация разных типов, что важно при работе с большими данными, которые имеют разную структуру и взяты из разных источников.

– Базы легко масштабируются.

– Большинство языков имеют специальные инструменты для работы с Mongo — например, в JavaScript это Mongoose.

– Благодаря индексации, системе запросов и другим особенностям можно быстро искать, читать и записывать данные в базах.

– Базы MongoDB могут работать сразу на нескольких серверах: сегментирование позволяет распределять нагрузку, а репликация — создавать копии. Поэтому система работает быстро и без перебоев.

В данной главе были рассмотрены общие сведения про базы данных, необходимые для использования в реализации курсового проекта. База данных является мощным инструментом хранения и упорядочивания данных, поэтому приведенные выше сведения являются необходимыми для изучения.

2 ПЛАТФОРМА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

2.1 Linux

Linux – это свободная и открытая операционная система, которая является платформой программного обеспечения с огромным разнообразием приложений и инструментов. Она была создана Линусом Торвальдсом в 1991 году и с тех пор стала одной из самых популярных и широко используемых операционных систем в мире.

Первоначально Линус Торвальдс не хотел продавать свою разработку. И не хотел, чтобы её продавал кто-то другой. Это было чётко прописано в уведомлении об авторских правах, помещённом в файл COPYING самой первой версии — 0.01. Причём требование Линуса налагало значительно более жёсткие ограничения на распространение Linux, чем те, которые провозглашались в лицензии GNU: не разрешалось взимать никаких денег за передачу или использование Linux. Но уже в феврале 1992-го года к нему стали обращаться за разрешением брать плату за распространение дискет с Linux, чтобы покрыть временные затраты и стоимость дискет. Кроме того, необходимо было считаться и с тем, что при создании Linux использовалось множество свободно распространяемых по интернету инструментов, самым важным из которых был компилятор GCC.

Авторские права на него оговорены в общественной лицензии GPL, которую изобрёл Ричард Столлман. Торвальдсу пришлось пересмотреть свое заявление об авторских правах, и, начиная с версии 0.12, он тоже перешёл на использование лицензии GPL.

С технической точки зрения, Linux представляет собой только ядро Unix-подобной операционной системы, отвечающее за взаимодействие с аппаратной частью компьютера и выполнение таких задач, как распределение памяти, выделение процессорного времени различным программам и так далее.

Кроме ядра, операционная система включает в себя множество различных утилит, которые служат для организации взаимодействия пользователя с системой. Успех Linux как операционной системы во многом обусловлен тем, что к 1991-му году в рамках проекта GNU уже было разработано множество утилит, свободно распространяемых в интернете. Проекту GNU не хватало ядра, а ядро, скорее всего, осталось бы невостребованным, если бы отсутствовали необходимые для работы утилиты. Линус Торвальдс оказался со своей разработкой в нужном месте в нужное время. И Ричард Столлман прав, когда настаивает на том, что операционную

систему следует называть не Linux, а GNU/Linux. Но название Linux исторически закрепилось за этой ОС.

Linux является модульной и гибкой платформой, которая поддерживает различные архитектуры и аппаратное обеспечение. Он может работать на серверах, настольных компьютерах, мобильных устройствах, встроенных системах и даже на суперкомпьютерах. Благодаря этой универсальности Linux нашел применение в различных областях, включая веб-серверы, базы данных, научные исследования, разработку программного обеспечения, мультимедиа, мобильные устройства и многое другое.

Linux имеет мощную командную строку, которая предоставляет пользователю множество инструментов для управления системой и выполнения задач. Кроме того, существует множество графических сред разработки (IDE) и интерфейсов пользователя, которые делают работу с Linux более удобной для начинающих пользователей. Linux также поддерживает большое количество программного обеспечения. Существует обширный выбор приложений для различных задач, включая офисные пакеты, браузеры, мультимедийные инструменты, графические редакторы, разработчикам программного обеспечения и многое другое. Большинство из них доступны бесплатно и имеют открытый исходный код.

В целом, Linux представляет собой мощную и гибкую платформу программного обеспечения, которая обеспечивает большую свободу и контроль над компьютерной системой. Благодаря своей открытой природе и активному сообществу разработчиков, Linux продолжает развиваться и оставаться одним из главных игроков в мире операционных систем.

2.2 Устройство диска в Linux

Логический диск в Linux представляет собой раздел или том на физическом диске. Разделы позволяют разделить физический диск на несколько изолированных областей, каждая из которых может иметь свою файловую систему. Логические диски и их разделы представлены в виде файлов в каталоге /dev.

Для работы с логическими дисками в Linux используются утилиты, такие как *fdisk, parted* и *lsblk*. *fdisk* и *parted* позволяют создавать, изменять и удалять разделы на физическом диске, а *lsblk* выводит информацию о физических и логических дисках, а также о разделах.

Логические диски могут быть отформатированы с использованием различных файловых систем, таких как ext4, XFS, Btrfs и другие.

Форматирование диска создает файловую систему, позволяющую хранить файлы и каталоги. На рисунке 2.1 приведена схема типичного диска в Linux.

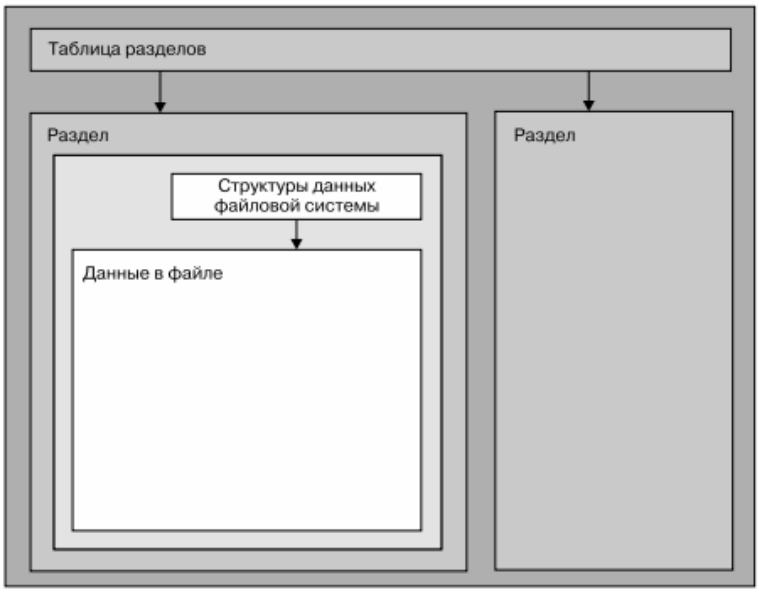

Рисунок 2.1 – Схема диска в Linux

Разделы являются более мелкими частями всего диска. В Linux они обозначаются с помощью цифры после названия блочного устройства и, следовательно, получают такие имена, как, например, /dev/sda1 и /dev/sdb3. Ядро представляет каждый раздел в виде блочного устройства, как если бы это был целый диск. Разделы определяются в небольшой области диска, которая называется таблицей разделов.

Хотя ядро и позволяет иметь одновременный доступ ко всему диску и к одному из его разделов, этого не придется делать, если только нет цели скопировать весь диск. Как можно заметить на рисунке 2.1, если пользователю необходим доступ к данным в файле, ему потребуется выяснить из таблицы разделов расположение соответствующего раздела, а затем отыскать в базе данных файловой системы этого раздела желаемый файл с данными.

Чтобы обращаться к данным на диске, ядро Linux использует систему слоев, показанную на рисунке 2.2. Подсистема SCSI и все связанное с ней представлены в виде одного контейнера. Следует обратить внимание на то, что с дисками можно работать как с помощью файловой системы, так и непосредственно через дисковые устройства.

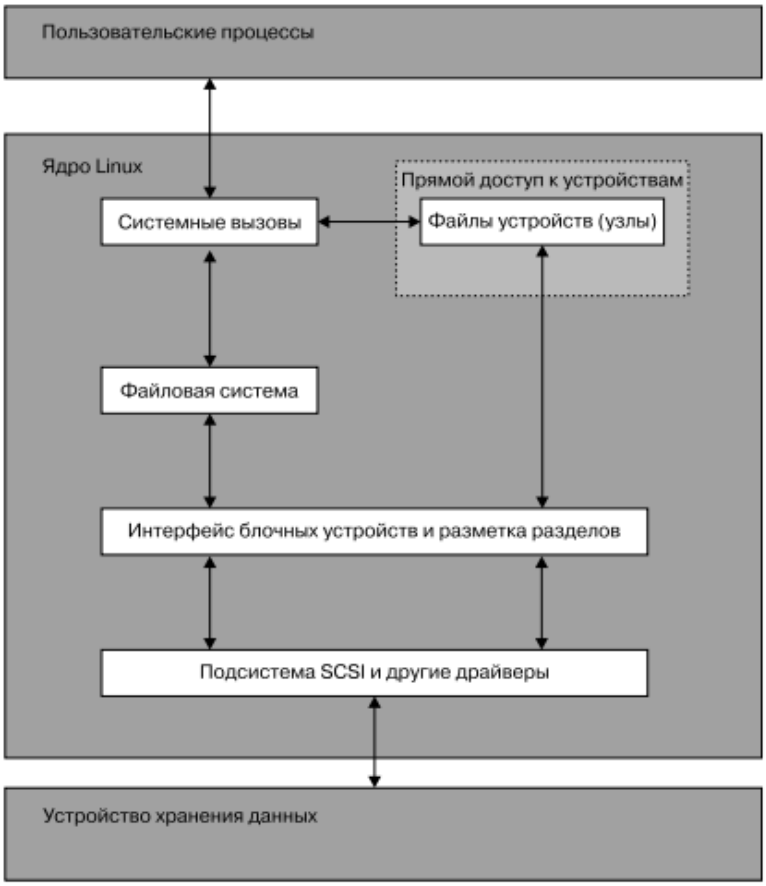

Рисунок 2.2 – Схема доступа ядра к диску

Существуют различные типы таблиц разделов. Традиционная таблица – та, которая расположена внутри главной загрузочной записи MBR (Master Boot Record). Новым, набирающим силу стандартом является глобальная таблица разделов с уникальными идентификаторами GPT (Globally Unique Identifier Partition Table).

Просмотр таблиц разделов – операция сравнительно простая и безвредная. Изменение таблиц разделов также осуществляется довольно просто, однако при таком типе изменений диска могут возникнуть опасности:

1 Изменение таблицы разделов сильно усложняет восстановление любых данных в удаляемых разделах, поскольку при этом меняется начальная точка привязки файловой системы. Обязательно нужно создавать резервную копию диска, на котором меняются разделы, если он содержит важную информацию.

2 Следует убедится в том, что на целевом диске ни один из разделов в данный момент не используется. Это важно, поскольку в большинстве версий Linux автоматически монтируется любая обнаруженная файловая система.

2.3 Файловая система

Последним звеном между ядром и пространством пользователя для дисков обычно является файловая система. С ней помогают взаимодействовать, такие команды, как *ls* и *cd*. Как отмечалось ранее, файловая система является разновидностью базы данных; она поддерживает структуру, призванную трансформировать простое блочное устройство в замысловатую иерархию файлов и подкаталогов, которую пользователи способны понять.

В свое время файловые системы, располагавшиеся на дисках и других физических устройствах, использовались исключительно для хранения данных. Однако древовидная структура каталогов, а также интерфейс ввода-вывода довольно гибки, поэтому теперь файловые системы выполняют множество задач, например роль системных интерфейсов. Файловые системы традиционно реализованы внутри ядра, однако инновационный протокол **9P** из операционной системы **Plan 9** способствовал разработке файловых систем в пространстве пользователя. Функция FUSE (*File System in User Space*, файловая система в пространстве пользователя) позволяет применять такие файловые системы в Linux.

В Linux включена поддержка таких файловых систем, как «родные» разработки, оптимизированные для Linux, «чужеродные» типы, например семейство Windows FAT, универсальные файловые системы вроде ISO 9660 и множество других. В приведенном ниже списке перечислены наиболее распространенные типы файловых систем для хранения данных. Имена типов систем, как их определяет Linux, приведены в скобках после названия файловых систем.

1 Четвертая расширенная файловая система (ext4) является текущей реализацией в линейке «родных» для Linux файловых систем. Вторая расширенная файловая система (ext2) долгое время была системой по умолчанию в системах Linux, которые испытывали влияние традиционных файловых систем Unix, таких как файловая система Unix (UFS, Unix File System) и быстрая файловая система (FFS, Fast File System). В третьей расширенной файловой системе (ext3) появился режим журналирования (небольшой кэш за пределами нормальной структуры данных файловой системы) для улучшения целостности данных и ускорения загрузки системы. Файловая система ext4 является дальнейшим улучшением, с поддержкой файлов большего размера по сравнению с допустимым в системах ext2 или ext3, а также большего количества подкаталогов. Среди расширенных файловых систем присутствует некоторая доля обратной совместимости.

Например, можно смонтировать систему ext2 как ext3 или наоборот, а также смонтировать файловые системы ext2 и ext3 как ext4, однако нельзя смонтировать файловую систему ext4 как ext2 или ext3 (рисунок 2.3).

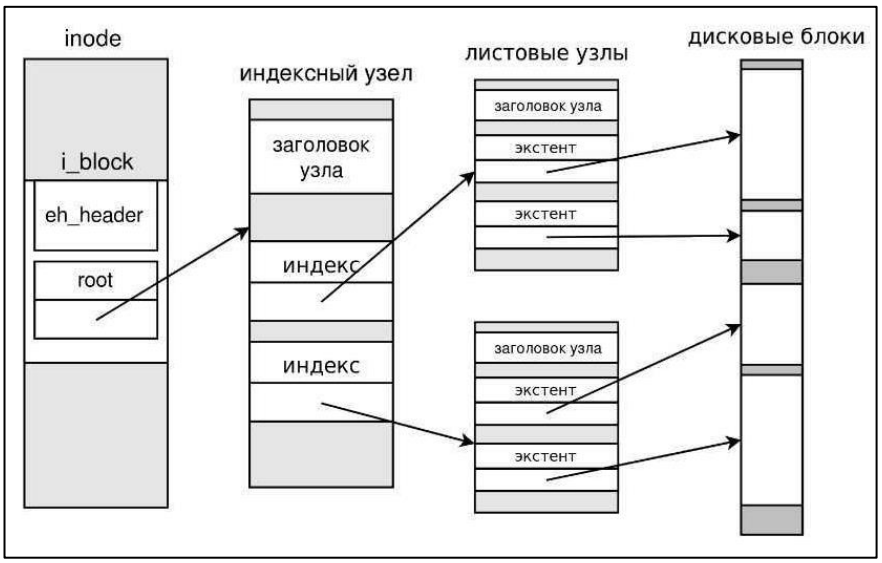

Рисунок 2.2 – Схема файловой системы ext4

2 Файловая система ISO 9660 (iso9660) — это стандарт для дисков CD-ROM. Большинство дисков CD-ROM использует какой-либо вариант стандарта ISO 9660.

3 Файловые системы FAT (msdos, vfat, umsdos) относятся к системам Microsoft. Простой тип msdos поддерживает весьма примитивное унылое многообразие систем MS-DOS. Для большинства современных файловых систем Windows следует использовать тип vfat, чтобы получить возможность полного доступа из ОС Linux. Редко используемый тип umsdos представляет интерес для Linux: в нем есть поддержка таких особенностей Unix, как символические ссылки, которые находятся над файловой системой MS-DOS.

4 Тип HFS+ (hfsplus) является стандартом Apple, который используется в большинстве компьютеров Macintosh.

Хотя расширенные файловые системы были абсолютно пригодны для применения обычными пользователями, в технологии файловых систем были произведены многочисленные улучшения, причем такие, что даже система ext4 не может ими воспользоваться в силу требований обратной совместимости.

Во второй главе данного курсового проекта была рассмотрена платформа программного обеспечения, основные плюсы и минусы Linux и его файловой системы.

3 СТРУКТУРИРОВАННЫЙ ЯЗЫК ЗАПРОСОВ

3.1 Общая информация про язык запросов

SQL (Structured Query Language) – структурированный язык запросов – является инструментом, предназначенным для выборки и обработки информации, содержащейся в компьютерной базе данных. SQL является языком программирования, применяемым для организации взаимодействия пользователя с базой данных (рисунок 3.1).

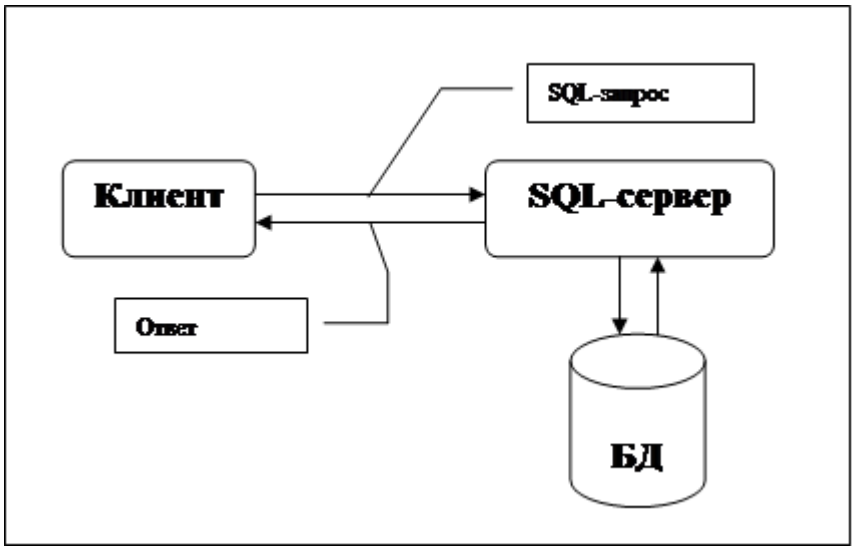

Рисунок 3.1 – Взаимодействие с базой данных с помощью SQL

SQL работает только с реляционными базами данных и предоставляет пользователю следующие функциональные возможности:

– изменение структуры представления данных;

– выборка данных из базы данных;

– обработка базы данных, т. е. добавление новых данных, изменение, удаление имеющихся данных;

– управление доступом к базе данных;

– совместное использование базы данных пользователями, работающими параллельно;

– обеспечение целостности базы данных.

SQL – это не полноценный компьютерный язык типа PASCAL, C++, JAVA. Еще раз отметим, что SQL, также как и QBE, является непроцедурным языком. С помощью SQL описываются свойства и взаимосвязи сущностей (объектов, переменных и т. п.), но не алгоритмы решения задачи. Он не

содержит условных операторов, операторов цикла, организации подпрограмм, ввода-вывода и т. п. В связи с этим SQL автономно не используется. Инструкции SQL встраиваются в программу, написанную на традиционном языке программирования и дают возможность получить доступ к базам данных. Кроме того, из таких языков, C, C++, JAVA инструкции SQL можно посылать СУБД в явном виде, используя интерфейс вызовов функций.

Язык SQL является многофункциональным языком. Во-первых, SQL используется в качестве языка интерактивных запросов пользователей с целью выборки данных и в качестве встроенного языка программирования баз данных. Кроме того, SQL используется в качестве языка администрирования БД для определения структуры базы данных и управления доступом к данным, находящимся на сервере; в качестве языка создания приложений клиент/сервер, доступа к данным в среде Internet, распределенных баз данных.

С помощью SQL можно динамически изменять и расширять структуру базы данных даже в то время, когда пользователи работают с ее содержимым. Таким образом, SQL обеспечивает максимальную гибкость. Статические языки определения данных запрещают доступ к БД во время изменения ее структуры.

3.2 Достоинства и недостатки SQL

Язык SQL является основой многих СУБД, т.к. отвечает за физическое структурирование и запись данных на диск, а также за чтение данных с диска, позволяет принимать SQL-запросы от других компонентов СУБД и пользовательских приложений. Таким образом, SQL - мощный инструмент, который обеспечивает пользователям, программам и вычислительным системам доступ к информации, содержащейся в реляционных базах данных.

Основные достоинства языка SQL заключаются в следующем:

1 *Стандартность* – как уже было сказано, использование языка SQL в программах стандартизировано международными организациями

2 *Независимость от конкретных СУБД* – все распространенные СУБД используют SQL, т.к. реляционную базу данных можно перенести с одной СУБД на другую с минимальными доработками;

3 *Возможность переноса с одной вычислительной системы на другую* – СУБД может быть ориентирована на различные вычислительные системы, однако приложения, созданные с помощью SQL, допускают использование как для локальных БД, так и для крупных многопользовательских систем;

4 *Реляционная основа языка* – SQL является языком реляционных БД, поэтому он стал популярным тогда, когда получила широкое распространение

реляционная модель представления данных. Табличная структура реляционной БД хорошо понятна, а потому язык SQL прост для изучения;

5 *Возможность создания интерактивных запросов* – SQL обеспечивает пользователям немедленный доступ к данным, при этом в интерактивном режиме можно получить результат запроса за очень короткое время без написания сложной программы;

6 *Возможность программного доступа к БД* – язык SQL легко использовать в приложениях, которым необходимо обращаться к базам данных. Одни и те же операторы SQL употребляются как для интерактивного, так и программного доступа, поэтому части программ, содержащие обращение к БД, можно вначале проверить в интерактивном режиме, а затем встраивать в программу;

7 *Обеспечение различного представления данных* – с помощью SQL можно представить такую структуру данных, что тот или иной пользователь будет видеть различные их представления. Кроме того, данные из разных частей БД могут быть скомбинированы и представлены в виде одной простой таблицы, а значит, представления пригодны для усиления защиты БД и ее настройки под конкретные требования отдельных пользователей;

8 *Возможность динамического изменения и расширения структуры БД* – язык SQL позволяет манипулировать структурой БД, тем самым обеспечивая гибкость с точки зрения приспособленности БД к изменяющимся требованиям предметной области;

9 *Поддержка архитектуры клиент-сервер* – SQL – одно из лучших средств для реализации приложений на платформе клиент-сервер. SQL служит связующим звеном между взаимодействующей с пользователем клиентской системой и серверной системой, управляющей БД, позволяя каждой из них сосредоточиться на выполнении своих функций.

Язык SQL может использоваться широким кругом специалистов, включая администраторов баз данных, прикладных программистов и множество других конечных пользователей. Язык SQL – первый и пока единственный стандартный язык для работы с базами данных, который получил достаточно широкое распространение. Практически все крупнейшие разработчики СУБД в настоящее время создают свои продукты с использованием языка SQL либо с SQL-интерфейсом.

Но также, несмотря на количество своих достоинств и преимуществ, структурированнный язык запросов имеет и ряд недостатков:

1 Несоответствие реляционной модели данных – создатели реляционной модели данных Эдгар Кодд, Кристофер Дейт и их сторонники указывают на то, что SQL не является истинно реляционным языком. В опубликованном

Кристофером Дейтом и Хью Дарвеном Третьем Манифестеони излагают принципы СУБД следующего поколения и предлагают язык Tutorial D, который является подлинно реляционным.

2 Сложность – хотя SQL и задумывался как средство работы конечного пользователя, в конце концов он стал настолько сложным, что превратился в инструмент программиста.

3 Отступления от стандартов – несмотря на наличие международного стандарта ANSI SQL-92, многие компании, занимающиеся разработкой СУБД (например, Oracle, Sybase, Microsoft, MySQL AB), вносят изменения в язык SQL, применяемый в разрабатываемой СУБД, тем самым отступая от стандарта. Таким образом, появляются специфичные для каждой конкретной СУБД диалекты языка SQL.

4 Ограничения производительности – в некоторых случаях SQL может быть неэффективным с точки зрения производительности. Сложные запросы, особенно при работе с большими объемами данных, могут требовать большого количества ресурсов и занимать много времени для выполнения. Неправильно спроектированная база данных или неправильно написанный SQL-запрос могут привести к медленной работе или даже блокировкам базы данных.

В целом, SQL – мощный и широко используемый язык для работы с реляционными базами данных. Он обладает множеством преимуществ, но также имеет некоторые недостатки, и даже несмотря на них, многие из них могут быть преодолены с помощью правильного подхода, использования оптимизаций и дополнительных инструментов. SQL остается одним из наиболее популярных и широко применяемых языков для работы с данными, и его преимущества перевешивают недостатки в большинстве сценариев.

Важно разумно использовать SQL, учитывая его ограничения и особенности конкретной задачи или проекта. Иногда может потребоваться комбинировать SQL с другими инструментами или языками программирования для эффективной обработки данных и выполнения сложных задач.

В данной главе был рассмотрен структурированный язык запросов, его главные особенности и выяснены причины, почему он считается лучшим выбором для разработчиков СУБД, а также его слабые стороны.

4 ПРОЕКТИРОВАНИЕ ФУНКЦИОНАЛЬНЫХ ВОЗМОЖНОСТЕЙ ПРОГРАММЫ

В настоящее время создано множество систем управления базами данных имеющих приблизительно одинаковые возможности. Все СУБД позволяют создавать структуру БД и заполнять таблицы (создавать и просматривать файлы базы данных), редактировать их (обновлять данные, удалять ненужные данные, добавлять новые данные). Данные в базе можно упорядочить по одному или нескольким полям. Можно осуществить поиск данных в базе по различным критериям поиска. Важна возможность изменения существующей структуры базы данных. Также имеются возможности конструировать формы, формировать отчеты, реализовать нестандартные процедуры обработки данных (макросы, VB), создавать приложения пользователя (генераторы приложений, средства создания меню и панелей управления приложениями). Для реализации перечисленных возможностей в состав СУБД входят языковые средства.

Язык современной СУБД включает подмножество команд, относящихся к следующим специализированным языкам:

– *язык описания данных* — высокоуровневый непроцедурный язык декларативного типа, предназначенный для описания логической структуры данных;

– *язык манипулирования данными* — командный язык СУБД обеспечивающий выполнение основных операций по работе с данными — ввод, модификацию и вывод данных по запросам;

– *структурированный язык запросов* — обеспечивает манипулирование данными и определение схемы реляционной БДП, является стандартным средством доступа к серверу БД.

СУБД общего назначения не ориентированы на какую-либо предметную область или на информационные потребности какой-либо группы пользователей. Каждая система такого рода реализуется как программный продукт, способный функционировать на некоторой модели ЭВМ в определенной операционной системе и поставляется многим пользователям как коммерческое изделие. Такие СУБД обладают средствами настройки на работу с конкретной базой данных. Использование СУБД общего назначения в качестве инструментального средства для создания автоматизированных информационных систем, основанных на технологии баз данных, позволяет существенно сокращать сроки разработки, экономить трудовые ресурсы. Этим

СУБД присущи развитые функциональные возможности и даже определенная функциональная избыточность.

В рамках данного курсового проекта будет реализована простая система управления базами данных. Она будет включать в себя реализацию структурированного языка запросов для доступа к данным. Данные будут располагаться в зашифрованном виде в файловой системе Linux. Выбор данной ОС является обоснованным, потому что Linux позволяет взаимодействовать с ядром системы настолько, насколько это возможно и другие платформы не позволяют такого. Следовательно, данная система будет помощником в реализации курсового проекта.

Выбором вида модели данных является реляционная, так как она является наиболее популярной среди пользователей и простой для понимания. Плюсом также является ее производительность при не очень больших объемах данных. В целом, реляционная модель обладает рядом преимуществ, которые были описаны в главе 1, пункте 1.2.

Хранение данных в файловой имеет два больших преимущества по сравнению с другими метода хранения – простота реализации и переносимость. Файловая система – это естественное окружение для большинства операционных систем, поэтому реализация базы данных на основе файловой системы может быть относительно простой. База данных, хранящая данные в файлах, может легко переноситься между различными операционными системами без необходимости в сложных механизмах миграции данных.

Создание программного модуля, переносимого также имеет ряд преимуществ, а именно простота использования – пользователи могут запустить программный модуль сразу после загрузки, без необходимости выполнения процесса установки, который может быть сложным или занимать время и низкие требования к системе – поскольку программный модуль не требует установки, он не создает нагрузки на ресурсы компьютера, такие как реестр Windows или раздел диска, что делает его привлекательным для использования на компьютерах с ограниченными ресурсами.

Данная глава описывает основные функциональные возможности программы, также их практическое обоснование и некоторые тонкости реализации.

5 ВЗАИМОДЕЙСТВИЕ С РАЗРАБОТАННОЙ БАЗОЙ ДАННЫХ

Разработанное приложение представляет собой хранилище данных в файловой системе с возможность взаимодействия с ними через простой структурированный язык запросов. База данных реализована по принципу heap-таблиц, что позволяет хранить данные в файловой системе компьютера.

Куча (heap) представляет собой структуру данных, в которой записи хранятся в произвольном порядке на страницах данных, без явного порядка сортировки или кластеризации. В отличие от таблицы, организованной в виде кластеризованного индекса или хранящей данные в определенном порядке, куча не имеет определенного ключа сортировки.

В таблице-куче каждая запись вставляется в первое свободное место на странице данных. Когда страница заполняется, база данных автоматически выделяет новую страницу для хранения дополнительных записей. Таким образом, куча может эффективно использовать доступное пространство на диске, поскольку она не ограничена предопределенным размером страницы.

Перед тем, как вставить новую запись в кучу, SQL Server читает PFS-страницу (Page Free Space). И если на странице достаточно свободного места, то сохраняет новую строку на данной странице. А если свободного места недостаточно, то выделяет экстент (восемь новых страниц данных — 64 КБ) и сохраняет данные на новой странице. PFS отслеживает место на страницах данных, используя два бита. Уровень заполнения страницы данных можно указать только для страниц данных кучи. В отличие от кластеризованного индекса, строки данных не сортируются и не должны вводиться в отсортированном виде. Это дело базы данных решать, на какой странице сохранить запись данных.

Однако для доступа к месту записи страницы базы данных требуется знать, где имеется достаточно пространства на страницах данных для завершения транзакции. Такая информация может быть получена на основе записанного уровня заполнения страницы данных. Эта информация содержится на странице PFS.

Для создания таблицы используется простой SQL-запрос, который является стандартным для структурного языка запросов – CREATE TABLE, в нем дается название для таблицы и перечисляются названия и типы полей создаваемой таблицы. Как было упомянуто выше, таблица создается на куче, то есть, в случае конкретной реализации, создается бинарный файл, куда будут

записываться данные таблицы и создается файл схемы таблицы, который поясняет какие поля и типы полей используются в конкретной таблице.

Для добавления данных в таблицу используется немного измененное выражение INSERT INTO, изменено оно тем, что вместо конкретных полей и данных используется путь к файлу и название таблицы. Файл используется для удобства заполнения таблиц. Он должен иметь расширение .tbl и хранить в себе данные в виде CSV. Такой формат был использован для того, чтобы не повторять стандарты, реализованные во многих других системах управления базами данных. Также такой формат позволяет легко и быстро заполнить большим объемом данных таблицы, описанные в базе данных.

Имеется функционал удаления таблиц из базы данных, в данном аспекте больших отличий от других реализаций нет, используется скрипт DROP TABLE, в котором указывается название таблицы и он, соответственно, удаляет указанную таблицу и все привязанные к ней файлы.

Для получения данных из таблиц используются SELECT запросы, но их структура также отличается от стандартных. Например, в реализованной СУБД обязательно нужно перечислять все поля, которые требуется получить, т.е. нельзя использовать символ «*» для получения всех полей. Как и в обычных запросах пользователю предоставляется возможность использовать условия WHERE, которые работают по аналогичному принципу. Также реализована несложная алгебра SQL запросов, такая как SUM, JOIN, GROUP BY, но эти возможности находятся на стадии развития и могут быть дополнены в последующем. В целом выборку из базы данных делать довольно удобно и выглядит она наглядно.

Модуль базы данных не требует инсталлятора и может быть перенесен с одного компьютера на другой без применения дополнительных установок, что дает ему огромное преимущество в использовании из-за небольшого размера и отсутствия установки.

В этой главе курсового проекта была рассмотрена возможность того, как пользователь взаимодействует с разработанным модулем, рассмотрены основные преимущества использования конкретной системы управления базами данных.

ЗАКЛЮЧЕНИЕ

В рамках данного курсового проекта была выполнена разработка программного модуля, реализующего простую базу данных с возможностью хранения данных средствами файловой системы и обеспечения доступа к данным через упрощенный язык запросов. Целью проекта было создание переносимого программного модуля, который не требует установки дополнительных компонентов и обеспечивает основные операции работы с данными. В работе были рассмотрены основные аспекты баз данных, включая их классификацию, примеры систем управления базами данных и основные принципы их функционирования. Была изучена платформа программного обеспечения Linux, включая устройство диска и файловую систему, которые использовались для хранения данных. Также был рассмотрен структурированный язык запросов SQL и его достоинства и недостатки.

В процессе разработки программы был создан программный модуль, обеспечивающий основные операции работы с базой данных. Модуль был разработан с учетом переносимости и не требует дополнительных зависимостей, что позволяет его использование на различных платформах без установщика. Также был разработан механизм хранения данных с использованием файловой системы компьютера.

Для взаимодействия с разработанной базой данных был представлен упрощенный язык запросов, который обеспечивает основные операции добавления, обновления, удаления и выборки данных. Язык запросов был спроектирован с учетом простоты понимания и использования, чтобы пользователи могли легко работать с базой данных. Также язык запросов представлен отличным от стандартов для введения новых практик взаимодействия с базой данных.

В результате выполнения курсового проекта был достигнут основной поставленный целью – разработка программного модуля простой базы данных. Созданный модуль обладает функциональностью, позволяющей управлять данными, и обеспечивает возможность переносимого использования.

СПИСОК ЛИТЕРАТУРНЫХ ИСТОЧНИКОВ

[1] Уорд, Б. Внутреннее устройство Linux / Брайан Уорд. – Санкт-Петербург: Питер, 2016. – 384 с.

[2] Волк, В.К. Базы данных. Устройство баз данных / В.К. Волк. – Санкт-Петербург: Лань, 2020. – 244 с.

[3] Карпова, И.П. Базы данных / И.П. Карпова. – М.: Питер, 2013. – 240 с.

[4] Баканов, М.В. Системы управления базами данных / М.В. Баканов, В.В. Романова, Т.П. Крюкова. – Кемерово: Кем, 2010. – 166 с.

[5] Комаров, В. И. Путеводитель по базам данных / В. И. Комаров, В.В. Романова, Т.П. Крюкова. – М.: ДМК-Пресс, 2024. – 540 с.

[6] Хейн, Т. Unix и Linux. Руководство системного оператора / Т. Хейн, Э. Немет, Г. Снайдер. – М.: “И.Д. Вильямс”, 2012. – 1312 с.

[7] Бьюли, А. Изучаем SQL / А. Бьюли. – Москва : Символ, 2007. – 308 с.

[8] Что такое Linux [Электронный ресурс]. – Режим доступа: https://skillbox.ru/media/code/chto-takoe-linux-gayd-po-samoy-svobodnoy-operatsionnoy-sisteme/. – Дата доступа: 10.03.2024.

[9] Структура файловой системы Linux [Электронный ресурс]. – Режим доступа: https://antons-organization-1.gitbook.io/administrirovanie-linux/upravlenie-failami/struktura-failovoi-sistemy-linux. – Дата доступа: 15.03.2024.

[10] Шилдс, У. SQL / У. Шилдс. – Москва : Питер, 2022. – 223 с.

ПРИЛОЖЕНИЕ А
(обязательное)
Исходный код программы

Листинг 1. QueryPlan.cpp

```
#include "QueryPlan.h"

QueryPlan::QueryPlan(unordered_map<char *, DBFile *> *dbFileMap, Statistics
*statistics, Query *query) {
    this->dbFileMap = dbFileMap;
    this->statistics = statistics;
    this->query = query;

    MakeQueryPlan();
}

QueryPlan::~QueryPlan() {
}

void QueryPlan::MakeQueryPlan() {
    unordered_map<string, AndList *> tableSelectionAndList;
    vector<AndList *> joins;
    vector<AndList *> joins_arranged;

    // Load all the tables using SelectFile RelOp.
    LoadAllTables();

    // Split the AndList into selection and joins.
    SplitAndList(&tableSelectionAndList, &joins);

    // Apply selection on tables using SelectPipe RelOp.
    ApplySelection(&tableSelectionAndList);

    // Rearrange joins, so that number of intermediate tuples generated will be
minimum.
    RearrangeJoins(&joins, &joins_arranged);
    // Apply joins on tables using Join RelOp.
    ApplyJoins(&joins_arranged);

    // Apply group by if it is in the query using GroupBy RelOp.
    ApplyGroupBy();

    // Apply Function if it is in the query using Sum RelOp.
    ApplySum();

    // Apply Project using Project RelOp.
    ApplyProject();

    // Apply Duplicate removal using DuplicateRemoval RelOp if distinct is
present.
    ApplyDuplicateRemoval();
}
```

```
void QueryPlan::LoadAllTables() {
    TableList *tableList = query->tables;

    while (tableList) {
        // Create SelectFileNode RelOp.
        SelectFilePlanNode  *selectFileNode  =  new  SelectFilePlanNode(NULL,
NULL);

        // Create Schema with aliased attributes.
        PathConfig *pathConfig = PathConfig::GetInstance();
        Schema  *schema  =  new  Schema(pathConfig->GetSchemaPath(tableList-
>tableName), tableList->tableName);
        schema->AliasAttributes(tableList->aliasAs);

        if (dbFileMap) {
            if (dbFileMap->find(tableList->tableName) == dbFileMap->end()) {
                cerr << "ERROR: DB File is not open for table " << tableList-
>tableName << ".\n";
                exit(1);
            }
            selectFileNode->dbFile = dbFileMap->at(tableList->tableName);
        }

        selectFileNode->outputSchema = schema;
        selectFileNode->outputPipeId = nextAvailablePipeId++;

        relNameToGroupNameMap[tableList->aliasAs] = tableList->aliasAs;
        groupNameToRelOpNode[tableList->aliasAs] = selectFileNode;

        tableList = tableList->next;
    }
}

void      QueryPlan::ApplySelection(unordered_map<string,      AndList      *>
*tableSelectionAndList) {
    for (auto const &item : *tableSelectionAndList) {
        string relName = item.first;
        string groupName = relNameToGroupNameMap[relName];
        SelectFilePlanNode  *inputRelOpNode  =  dynamic_cast<SelectFilePlanNode
*>(groupNameToRelOpNode[groupName]);

        // Create CNF
        AndList *andList = item.second;
        CNF *cnf = new CNF();
        Record *literal = new Record();
        cnf->GrowFromParseTree(andList,          inputRelOpNode->outputSchema,
*literal); // constructs CNF predicate

        if (!inputRelOpNode->selOp) {
            inputRelOpNode->selOp = cnf;
            inputRelOpNode->literal = literal;
        } else {
            // Create SelectPipe
```

```
            SelectPipePlanNode          *selectPipeNode          =          new
SelectPipePlanNode(inputRelOpNode, NULL);

            selectPipeNode->outputSchema = inputRelOpNode->outputSchema;
            selectPipeNode->outputPipeId = nextAvailablePipeId++;

            selectPipeNode->selOp = cnf;
            selectPipeNode->literal = literal;

            groupNameToRelOpNode[groupName] = selectPipeNode;
        }

        // Updating Statistics
        char *applyRelNames[] = {const_cast<char *>(relName.c_str())};
        statistics->Apply(andList, applyRelNames, 1);
    }
}

void QueryPlan::ApplyJoins(vector<AndList *> *joins) {
    for (AndList *andList : *joins) {
        OrList *orList = andList->left;

        string leftOperandName = string(orList->left->left->value);
        string          tableName1          =          leftOperandName.substr(0,
leftOperandName.find('.'));
        string groupName1 = relNameToGroupNameMap[tableName1];
        RelOpPlanNode *inputRelOp1 = groupNameToRelOpNode[groupName1];

        string rightOperandName = string(orList->left->right->value);
        string          tableName2          =          rightOperandName.substr(0,
rightOperandName.find('.'));
        string groupName2 = relNameToGroupNameMap[tableName2];
        RelOpPlanNode *inputRelOp2 = groupNameToRelOpNode[groupName2];

        // constructs CNF predicate
        CNF *cnf = new CNF();
        Record *literal = new Record();
        cnf->GrowFromParseTree(andList,             inputRelOp1->outputSchema,
inputRelOp2->outputSchema, *literal);

        JoinPlanNode *joinNode = new JoinPlanNode(inputRelOp1, inputRelOp2);

        Schema    *outputSchema    =    new    Schema(inputRelOp1->outputSchema,
inputRelOp2->outputSchema);
        joinNode->outputSchema = outputSchema;
        joinNode->outputPipeId = nextAvailablePipeId++;

        joinNode->selOp = cnf;
        joinNode->literal = literal;

        string newGroupName;
        newGroupName.append(groupName1).append("&").append(groupName2);
        relNameToGroupNameMap[tableName1] = newGroupName;
        relNameToGroupNameMap[tableName2] = newGroupName;
```

```
        groupNameToRelOpNode.erase(groupName1);
        groupNameToRelOpNode.erase(groupName2);
        groupNameToRelOpNode[newGroupName] = joinNode;
    }
}

void QueryPlan::ApplyGroupBy() {
    NameList *nameList = query->groupingAtts;
    if (!nameList)
        return;

    // Get Resultant RelOp Node.
    string finalGroupName = GetResultantGroupName();
    RelOpPlanNode *inputRelOpNode = groupNameToRelOpNode[finalGroupName];

    Schema *groupByInputSchema = inputRelOpNode->outputSchema;

    // Build Compute function.
    Function *function = new Function();
    function->GrowFromParseTree(query->finalFunction, *groupByInputSchema);

    // Build OrderMaker
    OrderMaker *orderMaker = new OrderMaker(groupByInputSchema, nameList);

    // Build Schema and keepMe From nameList
    vector<int> keepMeVector;
    Schema  *groupedAttsSchema  =  new  Schema(groupByInputSchema,  nameList,
&keepMeVector);

    Schema *outputSchema = new Schema(&sumSchema, groupedAttsSchema);

    // Create Group by Node.
    GroupByPlanNode *groupByNode = new GroupByPlanNode(inputRelOpNode, NULL);

    groupByNode->outputSchema = outputSchema;
    groupByNode->outputPipeId = nextAvailablePipeId++;

    groupByNode->groupAtts = orderMaker;
    groupByNode->computeMe = function;
    groupByNode->distinctFunc = query->distinctFunc;

    groupNameToRelOpNode[finalGroupName] = groupByNode;
}

void QueryPlan::ApplySum() {
    if (query->groupingAtts || !query->finalFunction) {
        return;
    }

    // Get Resultant RelOp Node.
    string finalGroupName = GetResultantGroupName();
    RelOpPlanNode *inputRelOpNode = groupNameToRelOpNode[finalGroupName];

    // Build Compute function.
    Function *function = new Function();
```

```
    function->GrowFromParseTree(query->finalFunction,         *inputRelOpNode-
>outputSchema);

    SumPlanNode *sumNode = new SumPlanNode(inputRelOpNode, NULL);
    sumNode->outputSchema = &sumSchema;
    sumNode->outputPipeId = nextAvailablePipeId++;

    sumNode->computeMe = function;
    sumNode->distinctFunc = query->distinctFunc;

    groupNameToRelOpNode[finalGroupName] = sumNode;
}

void QueryPlan::ApplyProject() {
    NameList *attsToSelect = query->attsToSelect;

    if (query->finalFunction) {
        NameList *sumAtt = new NameList();
        sumAtt->name = SUM_ATT_NAME;
        sumAtt->next = attsToSelect;
        attsToSelect = sumAtt;
    }

    if (!attsToSelect)
        return;

    // Get Resultant RelOp Node.
    string finalGroupName = GetResultantGroupName();
    RelOpPlanNode *inputRelOpNode = groupNameToRelOpNode[finalGroupName];

    Schema *inputSchema = inputRelOpNode->outputSchema;

    vector<int> *keepMeVector = new vector<int>;
    Schema *outputSchema = new Schema(inputSchema, attsToSelect, keepMeVector);

    int *keepMe = new int();
    keepMe = &keepMeVector->at(0);

    if (inputSchema->GetNumAtts() == outputSchema->GetNumAtts()) {
        return;
    }

    // Create Project RelOp Node
    ProjectPlanNode *projectNode = new ProjectPlanNode(inputRelOpNode, NULL);

    projectNode->outputSchema = outputSchema;
    projectNode->outputPipeId = nextAvailablePipeId++;

    projectNode->keepMe = keepMe;
    projectNode->numAttsInput = inputSchema->GetNumAtts();
    projectNode->numAttsOutput = outputSchema->GetNumAtts();

    groupNameToRelOpNode[finalGroupName] = projectNode;
}
```

```
void QueryPlan::ApplyDuplicateRemoval() {
    if (!query->distinctAtts)
        return;

    // Get Resultant RelOp Node.
    string finalGroupName = GetResultantGroupName();
    RelOpPlanNode *inputRelOpNode = groupNameToRelOpNode[finalGroupName];

    // Create Distinct RelOp Node.
    DuplicateRemovalPlanNode        *duplicateRemovalNode        =        new
DuplicateRemovalPlanNode(inputRelOpNode, NULL);

    duplicateRemovalNode->outputPipeId = nextAvailablePipeId++;
    duplicateRemovalNode->outputSchema = inputRelOpNode->outputSchema;

    duplicateRemovalNode->inputSchema = inputRelOpNode->outputSchema;

    groupNameToRelOpNode[finalGroupName] = duplicateRemovalNode;
}

void      QueryPlan::SplitAndList(unordered_map<string,       AndList       *>
*tableSelectionAndList, vector<AndList *> *joins) {

    AndList *andList = query->andList;
    while (andList) {
        unordered_map<string, AndList *> currentTableSelectionAndList;

        OrList *orList = andList->left;

        while (orList) {

            Operand *leftOperand = orList->left->left;
            Operand *rightOperand = orList->left->right;

            // Duplicate OrList
            OrList *newOrList = new OrList();
            newOrList->left = orList->left;
            if (leftOperand->code == NAME && rightOperand->code == NAME) {
                AndList *newAndList = new AndList();

                // Add to new or list to and list.
                newAndList->left = newOrList;

                // Push newly created and list to joins vector.
                joins->push_back(newAndList);
            } else if (leftOperand->code == NAME || rightOperand->code == NAME)
{
                Operand *nameOperand = leftOperand->code == NAME ? leftOperand
: rightOperand;
                string name = string(nameOperand->value);
                string relationName = name.substr(0, name.find('.'));

                if     (currentTableSelectionAndList.find(relationName)     ==
currentTableSelectionAndList.end()) {
                    AndList *newAndList = new AndList();
```

```
                    newAndList->left = newOrList;
                    currentTableSelectionAndList[relationName] = newAndList;
                } else {
                    OrList                  *currentOrList                  =
currentTableSelectionAndList[relationName]->left;

                    while (currentOrList->rightOr) {
                        currentOrList = currentOrList->rightOr;
                    }
                    currentOrList->rightOr = newOrList;
                }
            }
            orList = orList->rightOr;
        }

        // Iterate and merge and lists
        for (auto const &item : currentTableSelectionAndList) {
            if           (tableSelectionAndList->find(item.first)           ==
tableSelectionAndList->end()) {
                (*tableSelectionAndList)[item.first] = item.second;
            } else {
                AndList     *currentAndList     =     tableSelectionAndList-
>at(item.first);

                while (currentAndList->rightAnd) {
                    currentAndList = currentAndList->rightAnd;
                }

                currentAndList->rightAnd = item.second;
            }
        }

        andList = andList->rightAnd;
    }
}

void  QueryPlan::RearrangeJoins(vector<AndList  *>  *joins,  vector<AndList  *>
*joins_arranged) {
    int n = joins->size();
    if (n < 1) {
        return;
    }
    int initialPermutation[n];
    for (int i = 0; i < n; i++) {
        initialPermutation[i] = i;
    }

    vector<int *> permutations;

    HeapPermutation(initialPermutation,     joins->size(),     joins->size(),
&permutations);

    int minI = -1;
    double minIntermediateTuples = DBL_MAX;
    for (int i = 0; i < permutations.size(); i++) {
```

```
        double permutationIntermediateTuples = 0.0;
        Statistics dummy(*statistics);

        int relNamesIndex = 0;
        char **relNames = new char *[2 * n];
        unordered_set<string> relNamesSet;
        for (int j = 0; j < n; j++) {
            AndList *currentAndList = joins->at(permutations[i][j]);
            string attNameWithRelName1 = string(currentAndList->left->left->left->value);
            string attNameWithRelName2 = string(currentAndList->left->left->right->value);

            string relName1 = attNameWithRelName1.substr(0, attNameWithRelName1.find('.'));
            string relName2 = attNameWithRelName2.substr(0, attNameWithRelName2.find('.'));

            if (relNamesSet.find(relName1) == relNamesSet.end()) {
                relNamesSet.insert(string(relName1));
                char *newRel = new char[relName1.length() + 1];
                strcpy(newRel, relName1.c_str());
                relNames[relNamesIndex++] = newRel;
            }

            if (relNamesSet.find(relName2) == relNamesSet.end()) {
                relNamesSet.insert(relName2);
                char *newRel = new char[relName2.length() + 1];
                strcpy(newRel, relName2.c_str());
                relNames[relNamesIndex++] = newRel;
            }

            double intermediate = dummy.Estimate(currentAndList, relNames, relNamesIndex);
            permutationIntermediateTuples += intermediate;
            dummy.Apply(currentAndList, relNames, relNamesIndex);

        }
        if (permutationIntermediateTuples < minIntermediateTuples) {
            minIntermediateTuples = permutationIntermediateTuples;
            minI = i;
        }
    }

    for (int i = 0; i < n; i++) {
        joins_arranged->push_back(joins->at(permutations[minI][i]));
    }
}

void HeapPermutation(int *a, int size, int n, vector<int *> *permutations) {
    // if size becomes 1 then prints the obtained
    // permutation
    if (size == 1) {
        // Add new Permutation in the permutations vector.
        int *newPermutation = new int[n];
```

```
        for (int i = 0; i < n; i++) {
            newPermutation[i] = a[i];
        }
        permutations->push_back(newPermutation);
        return;
    }

    for (int i = 0; i < size; i++) {
        HeapPermutation(a, size - 1, n, permutations);

        // if size is odd, swap first and last
        // element
        if (size % 2 == 1)
            swap(a[0], a[size - 1]);

            // If size is even, swap ith and last
            // element
        else
            swap(a[i], a[size - 1]);
    }
}

string QueryPlan::GetResultantGroupName() {
    if (groupNameToRelOpNode.size() != 1) {
        cerr << "Query is not correct. Group by can be performed on only one
group\n";
        exit(1);
    }
    return groupNameToRelOpNode.begin()->first;
}

RelOpPlanNode *QueryPlan::GetQueryPlan() {
    return groupNameToRelOpNode[GetResultantGroupName()];
}

void QueryPlan::Print() {
    cout << "PRINTING TREE POST ORDER: " << "\n";
    PrintQueryPlanPostOrder(groupNameToRelOpNode[GetResultantGroupName()]);
}

void QueryPlan::PrintQueryPlanPostOrder(RelOpPlanNode *node) {
    if (node == nullptr)
        return;

    PrintQueryPlanPostOrder(node->child1);
    PrintQueryPlanPostOrder(node->child2);
    node->Print();
}
```

ПРИЛОЖЕНИЕ Б

(обязательное)

Функциональная схема алгоритма

ПРИЛОЖЕНИЕ В
(обязательное)
Блок схема алгоритма

ПРИЛОЖЕНИЕ Г
(обязательное)
Скриншоты реализации

ПРИЛОЖЕНИЕ Д
(обязательное)
Ведомость документов