Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования

Московский государственный технический университет
имени Н.Э. Баумана
(МГТУ им. Н.Э. Баумана)

Факультет: «Информатика и системы управления»
Кафедра: «Теоретическая информатика и компьютерные технологии»

Расчетно-пояснительная записка к выпускной квалификационной работе

Генератор высокопроизводительной NoSQL базы данных.

Руководитель ВКР: _____ (А.В. Коновалов)
(подпись, дата)

Исполнитель ВКР,
студент группы ИУ9-82: _____ (В.С. Барашко)
(подпись, дата)

Москва, 2017

Аннотация

Темой работы является «Генератор высокопроизводительной NoSQL базы данных». Объем дипломной работы 52 страницы. В ней размещены 7 рисунков, 21 листинг и 1 таблица. При написании работы использовалось 5 источников литературы.

Объектом исследования при написании работы послужили технологии эффективного хранения и обработки данных. Предметом исследования стала разработка генератора базы данных, осуществляющей высокоэффективную запись, хранение и выборку данных.

В дипломную работу входят пять глав. Первая посвящена описанию предметной области, во второй формулируется расширенная постановка задачи, а оставшиеся три посвящены разработке, реализации и тестированию.

Во введении выполняется абстрактная формулировка задачи. Расширенная постановка задачи описывается более формально, с учетом всех нюансов и особенностей предметной области.

Заключение посвящено основным выводам и идеям, дальнейшему развитии темы, поднятой в данной работе.

Содержание

Обозначения и сокращения

БД — база данных.

СУБД – система управления базами данных.

NoSQL (англ. not only SQL, не только SQL) — ряд подходов, направленных на реализацию хранилищ баз данных, имеющих существенные отличия от моделей, используемых в традиционных реляционных СУБД.

API (англ. Application Programming Interface) — программный интерфейс приложения.

BLOB (англ. Binary Large Object) — двоичный большой объект.

RAM (англ. Random Access Memory) — память с произвольным доступом, оперативная память, оперативное запоминающее устройство (ОЗУ).

Введение

Сегодня при решений многих прикладных задач проблема хранения большого объема собранных за многие годы данных и их обработки ощущается все острее. Данные постепенно накапливаются, и при большом объеме информации поиск может представлять собой довольно трудоемкий процесс. Поэтому и возникла необходимость создания баз данных.

Без баз данных сегодня невозможно представить работу любой финансовой, промышленной или торговой организации. Они позволяют структурировать, хранить и извлекать информацию оптимальным способом. В самом общем смысле база данных — это набор записей и файлов, организованных специальным образом.

Основным способом хранения данных при решении подавляющего числа задач, так или иначе связанных с данными, по умолчанию являются реляционные базы данных. Иногда появляются другие технологии баз данных, например, объектные базы данных, но эти альтернативы не получили широкого распространения. Современные приложения отличаются возросшей сложностью, поэтому число различных систем управления базами данных особенно велико, так как каждая из них, ограничивая круг решаемых задач, получает те или иные преимущества решения своих особенных задач.

Хранилища данных, будучи спроектированы правильным образом, предоставляют все необходимые механизмы для доступа к информации, важной для принятия решений в компании в любой момент времени. А так же, делают доступ к информации максимально удобным, а саму информацию максимально достоверной. Гибкость хранилищ данных позволяет обеспечить все будущие потребности в компании, за счет

внесения исключительно небольших изменений в архитектуру хранилища.

В настоящее время активно развивается идея баз данных NoSQL. Термин «NoSQL» прижился, но никогда не имел строгого определения [4]. В настоящее время не существует ни общепринятого определения этого термина, ни авторитетного органа, который бы предложил такое определение, поэтому есть возможность лишь обсуждать некоторые общие свойства баз данных, относящихся к категории NoSQL. Чаще всего NoSQL базами данных называют базы данных, которые используют определенный ряд подходов, направленных на реализацию хранилищ баз данных, имеющих существенные отличия от моделей, используемых в традиционных реляционных системах управления базами данных.

Целью данной работы является разработка программного обеспечения, позволяющего его пользователю использовать техник высокопроизводительной NoSQL базы данных в собственных проектах, направленных на решение проблем эффективной записи и выборки данных.

Постановка задачи может быть сделана следующим образом: необходимо разработать программу, осуществляющую эффективную запись и выборку данных по заранее заданным спецификацией шаблонам, минимизировав при этом затраты по памяти. Для удовлетворения самых различных нужд такая программа должна генерироваться автоматически из поданного на вход программы спецификации, описывающей параметры данных, их типы, параметры процедур вставки и выборки по условиям.

Для того, чтобы описать задачу с меньшей степенью абстракции в работе присутствуют разделы, посвященные решенным подзадачам, связанным с эффективным хранением данных, описанием их параметров и механизмами их записи. В последующих главах четко формулируется расширенная

постановка задачи, после чего описывается разработка программы и примененных инструментов и техник.

Практическая значимость данной работы очевидна. Можно назвать широкий круг разнообразных областей, в которых задача эффективной записи и хранения данных стоит очень остро. В основном, это службы сбора того или иного потока данных: анализ сетевого трафика, высокочастотная алгоритмическая торговля, службы обработки банковских транзакций и другие. Анализатор трафика должен высокоэффективно осуществлять запись сетевых пакетов для их последующей обработки, торговые роботы обязаны очень быстро принимать решения о покупке или продаже ценных бумаг на основе имеющихся исторических данных, а банки — надежно и быстро обрабатывать колоссальное количество информации о переводах средств и покупках своих клиентов.

1. Обзор предметной области

База данных - это организованная структура, предназначенная для хранения информации. В современных базах данных хранятся не только данные, но и информация.

Это предложение можно пояснить, если, например, рассмотреть базу данных о сетевых пакетах. В ней накапливаются данные о сетевом трафике сети, об адресах отправителей и получателей, о содержимом трафика. Данные собираются и хранятся в бинарном виде, а для извлечения уже информации из них требуются необходимые навыки и набор программных средств для анализа данных.

С понятием базы данных тесно связано понятие СУБД. Система управления базой данных (СУБД) — это комплекс программных средств, предназначенных для создания структуры новой базы, наполнение ее содержимым, редактирование содержимого и визуализации информации. Под визуализацией информации базы понимается отбор отображаемых данных в соответствии с заданным критерием, их упорядочение, оформление и последующая выдача на устройства вывода или передачи по каналам связи.

В мире существует множество систем управления базами данных. Несмотря на то что они могут по-разному работать с разными объектами и предоставляют пользователю различные функции и средства, большинство СУБД опираются на единый устоявшийся комплекс основных понятий. Можно определить базу данных как физическое пространство (место на внешнем носителе компьютера), на котором в строго определенном порядке записываются и хранятся конкретные данные первичных информационных форм, относящиеся к одному роду объектов учета. В базе данных хранятся

данные, предназначенные для совместного использования. Здесь особо выделяется возможность совместного использования информации хранящейся в базе данных многими пользователями одновременно.

Различные СУБД позволяют создавать и обслуживать базы данных различной структуры: иерархические, сетевые, реляционные, объектно-ориентированные и гибридные [5], что между таблицами базы данных могут быть установлены различные отношения. Реляционные СУБД составляют один из крупных сегментов рынка баз данных: они включают все от систем клиент/сервер до настольных систем.

Реляционная модель БД рассматривает все данные как группы таблиц или отношений, которые содержат фиксированные количества рядов и столбцов. Иными словами многие объекты, используемые в реляционной базе данных, аналогичны объектам электронных таблиц. Рассмотрим основные термины и определения связанные с реляционными СУБД:

- поле — базовый элемент любой базы данных, не обязательно реляционной. Поля это элементарные информационные объект базы данных. «Элементарный» означает, что поле не может быть разбито на более мелкие порции информации. Кроме того, в каждом поле может храниться только строго определенный тип информации, например, текстовой или числовой.

- запись — набор данных специфицирующих некоторый объект. Например, в базы данных автотранспортных средств каждая запись содержит сведения о транспортном средстве: государственный номер, марку, год выпуска, номер кузова и прочие сведения. Каждая запись БД содержит отдельный набор информации; в примере, каждая запись представляет данные о конкретном транспортном средстве.

- таблица — это набор полей. Данные, содержащиеся в таблице, хранятся в виде записей. Каждая таблица базы данных представляет некоторый тип хранящихся в ней объектов. В базе данных может быть любое количество таблиц, между которыми могут быть установлены различные отношения. Тот факт, что таблица представляет только один тип объекта, отнюдь не является недостатком, наоборот, это один из ключей к созданию эффективной базы данных.

- первичный ключ — уникально идентифицирует каждую запись в таблице и в большиство реализаций баз данных не имеет повторяющихся значений. Выбор поля в качестве первичного ключа — одно из важнейших решений принимаемых при проектировании базы данных.

2. Расширенная постановка задачи

После краткого описания предметной области, можно четко формализовать задачу данной работы. Её можно сформулировать следующим образом — необходимо реализовать высокоэффективную базу данных, позволяющую записывать и считывать данные по заранее известной схеме данных. При этом, позволительно пренебречь поддержкой удаления и модификации данных, что увеличит эффективность работы целевых операций вставки м выборки. В то же время, полученная база данных должна быть максимально удобной в сопровождении.

Использование существующих продуктов из множества разнообразных систем управления базами данных противоречит идее высокоэффективности. В большинстве технологий, при обращении к базе необходимо сформировать запрос на некотором языке, описывающий операцию вставки или выборки. Такой запрос система должна распознать, то есть интерпретировать, что означает потери в скорости работы. Поэтому необходимо отказаться от такой модели в пользу модели генерации в коде заданных структур и процедур, что позволит сильно ускорить обработку запроса пользователя.

Для достижения цели высокой скорости операций необходимо выбрать максимально приближенный к низкоуровневым язык программирования. В наши дни выбор таких языков крайне мал, поэтому в подавляющем числе случаев при решении подобных задачах выбирается язык Си. Известными преимуществами языка Си являются его компилируемость и статическая типизация, что позволяет достичь высокой скорости.

Пренебрежение поддержкой удаления и модификации данных позволяет отказаться от реализации средств обновления данных. Однажды

подготовленный, набор записей в базе данных должнен использоваться без дополнительных проверок при выборке элементов из него, ожидая что записи всегда актуальны и никогда не обновлялись. Также это условие позволит модифицировать механизмы, обсуживающие работу базы, и структуры данных в сторону экономии памяти и скорости работы.

Для достижения цели удобства в сопровождении база данных должна автоматически генерироваться из описанной пользователем спецификации. Спецификация должна поддерживать указание:

1) таблиц с названиями полей, их типов, указание первичного ключа;

2) процедур на запись и чтение с указание параметров и выполняемых внутри них операций вставки (INSERT) и выборки (SELECT) данных.

В большинстве баз данных для описания запросов к базе данных используется язык SQL. Поэтому в целевой модели необходимо придерживаться идей языка SQL для задания возможного синтаксиса спецификаций.

3. Разработка

3.1. Обзор существующих технологий

Огромные объемы данных побуждают разработчиков и компании использовать в работе альтернативы реляционных баз данных. В совокупности все эти технологии известны как «NoSQL базы данных». Термин «NoSQL» был придуман Эриком Эвансом (Eric Evan) [4].

Основной проблемой является то, что реляционные базы данных очень плохо справляются с нагрузками, актуальными в наше время. Есть такие известные проблемные области:

- горизонтальное масштабирование при больших объемах данных;
- производительность каждого отдельного сервера;
- не гибкий дизайн логической структуры.

Многие компании нуждаются в нахождении новых путей для хранения и масштабирования огромных массивов данных. Под термином NoSQL скрывается большое количество продуктов с абсолютно разными дизайнами.

Масштабируемость

Под масштабируемостью имеется в виду автоматическое распределение данных между несколькими серверами, а не репликация данных. Такие системы называются распределенными базами данных. В их число входят Cassandra, HBase, Riak, Scalaris и Voldemort.

Нераспределенными базами данных являются, например, CouchDB, MongoDB, Neo4j, Redis и Tokyo Cabinet. Данные системы могут служить

прослойкой для хранения данных для распределенных систем.

Модель данных и запросов

Существует огромное многообразие моделей данных и API запросов в NoSQL базах данных.

Система семейства столбцов используется в Cassandra и HBase, и это идея была привнесена в них из документов, описывающих устройство Google Bigtable. В обеих системах присутствуют строки и столбцы, но количество строк не велико: каждая строка имеет больше или меньше столбцов, в зависимости от необходимости, и столбцы не обязаны быть определены заранее.

Система ключ-значения сама по себе проста и не сложна для реализации, но не эффективна, если присутствует необходимость только в запросе или обновлении части данных. Так же трудно реализовывать сложные структуры поверх распределенных систем.

Документо-ориентированные базы данных — это следующий уровень систем ключ-значение, позволяющих связывать вложенные данные с каждым ключом. Поддержка таких запросов более эффективна, чем просто возвращение всего BLOB каждый раз.

Neo4J обладает другой моделью данных, храня объекты и связи в качестве узлов и ребер графа. Для запросов, которые соответствуют этой модели (например, иерархических данных) они могут быть в тысячу раз быстрее, чем альтернативные варианты. Scalaris уникальна в использовании распределенных транзакций между несколькими ключами.

Система хранения данных

Под системой хранения данных понимается структура хранения данных

внутри системы. Система хранения данных оказывает большое влияние на то, какие нагрузки база может выдерживать.

Базы данных, хранящие данные в оперативной памяти, очень быстрые, но не могут работать с данными, превышающими размер доступной оперативной памяти. Долговечность (сохранение данных в случае сбоя на сервере или отключения питания) также может быть проблемой. Количество данных, которые могут ожидать записи на диск, потенциально велико.

Одна из таких систем с хранением данных в оперативной памяти — Scalaris, решает проблему долговечности с помощью репликации, но она не поддерживает масштабирования на несколько центров данных, так что потеря данных вероятна и в этом случае, например, при отключении питания.

Memtables и SSTables буферизируют запросы на запись в памяти. После накопления достаточного количества записей, они сортируют и записывают их на диск. Это дает производительность, близкую к производительности оперативной памяти, но в то же время система лишена проблем, актуальных при хранении только в оперативной памяти.

Б-деревья используются в базах данных уже очень давно. Они обеспечивают надежную поддержку индексирования, но производительность очень низка при использовании на машинах с жесткими дисками на магнитных дисках (которые по-прежнему наиболее экономически эффективны), так как происходит большое количество позиционирований головки при записи или чтении данных.

Интересным вариантом является использование в CouchDB Б-деревьев только с функцией добавления (дерево, которое не нужно перестраивать

при добавлении элементов), что позволяет получить удовлетворительную производительность при записи данных на диск.

4. Реализация

Целью проекта поставлена задача реализации генератора базы данных, используемого для построения библиотеки высокопроизводительной и компактной базы данных. Получая на входе спецификацию базы данных в виде SQL-подобного описания (содержащего структуру таблиц, индексы, процедуры записи и поиска), он должен порождает исходный код библиотеки на языке C.

Для корректного входного файла, при отсутствии в нем ошибок, программа генерирует заголовочный файл `*.h` и файл с исходным кодом `*.c`.

Для базы данных можно указать название, по умолчанию, им является имя файла. При названии `database` генерируются функции:

- открытия базы данных, в качестве параметра принимающие имя файла для хранения базы данных;
- закрытия базы данных, причем до вызова этой функции все модификации таблиц хранятся в оперативной памяти, при вызове — сохраняются на диск.

При наличии хотя бы одного первичного ключа для таблицы, выполняется генерация структур и методов на C99 для хранения и обслуживания структуры данных «Б-дерево». Иначе, структуры «куча».

Для каждой описанной процедуры выполняется генерация метода, осуществляющего выборку либо запись данных. Если у процедуры присутствует хотя бы один параметр с модификатором OUT, процедура является типа «на чтение». Иначе, «на запись».

Для процедуры типа «на запись» генерируется функция, выполняющая запись данных в соответствии со спецификацией.

Процедуры типа «запись» может включать в себя только один или более операторов `INSERT` вставки новых строк в соответствующие таблицы. При этом значения столбцов в операторе `INSERT` могут представлять собой значения параметров.

Процедуры типа «чтение» должна включать в себя только один или более операторов `SELECT` выборки строк данных.

Для процедуры типа «чтение» с названием генерируется:

- структура, включающие все выходные параметры с модификаторами `OUT` в качестве полей структуры;
- метод, осуществляющий инициализацию выборки, формирование буферов и выгрузку данных;
- метод, возвращающий флаг доступности следующей строки данных и обновляющий структуру-итератор следующей строкой данных.

Спецификация базы данных

Описание структуры таблиц включает в себя название таблицы, описание колонок и их типы.

```
CREATE TABLE table-name ( {column-name data-type} );
```

Описание процедур записи и поиска:

```
CREATE PROCEDURE procedure-name ({parameter-name data-type IN|OUT
   })
BEGIN
        { [INSERT TABLE table-name VALUES ({argument});] |
        [SELECT {column-name SET parameter-name} FROM table-name
           WHERE {condition}; ] }
END;
```

Подготовка к работе

Для работы с программой необходимо скачать исходный с открытого репозитория [1]. Чтобы выгрузить, необходимо осуществить последовательность команд, указанных на листинге 1.

Листинг 1. Загрузка исходного кода

```
$ git clone https://github.com/bmstu-iu9/mill-db
$ cd mill-db
```

Сборка исполняемого файла осуществляется вызовом прилагаемого `Makefile` (листинг 2):

Листинг 2. Команда сборки

```
$ make
```

Запуск

Запуск программмы осуществляется вызовом исполняемого файла с одним аргументом командной строки — путем к файлу спецификации (листинг 3):

Листинг 3. Запуск программы

```
$ milldb <filename>
```

При ошибке в вводе аргументов, программа предложит изучить страницу помощи (листинг 4):

Листинг 4. Вызов программы с неправильными аргументами

```
$ milldb spec1.txt spec2.txt
```

```
milldb: invalid options
Try 'milldb ---help' for more information.
```

При ошибке в подготовке спецификации программа указывает на строку с ошибкой (листинг 5):

Листинг 5. Ошибка при вызове

```
$ milldb spec1.txt
line 4: syntax error
```

4.1. Анализатор входной спецификации

Для удовлетворения самых разнообразных нужд пользователя полученная база данных должна уметь записывать и считывать данные разных типов. В то же время, для ускорения ответов базы на запросы пользователя структуры данных должны быть закодированы напрямую в коде базы данных. Эти пункты приводят к идее создания генератора базы данных, который бы на основе входной спецификации порождал код итоговой базы данных. Для успешного чтения генератором спецификации, необходимо осуществить, как минимум, лексический и синтаксический ее анализ.

Существует два класса синтаксических анализаторов — восходящие и нисходящие. Восходящие осуществляют разбор, начиная с листьев, которые являются входными лексемами, нисходящие, наоборот, начинают с корня дерева [6]. Если анализатор читает поток слева направо, он называется LR (left-rigth, слева направо). LR-анализаторы являются восходящими анализаторами, то есть строят дерево снизу вверх. Используется также термин LR(k)-анализатор, где k выражает количество непрочитанных символов предпросмотра во входном потоке, на основании которых принимаются решения при анализе. Обычно k равно 1 и часто опускается.

Преимущества LR-анализаторов:

- быстрый;
- мгновенное отображение и обработка ошибок.

Основым недостатком LR-анализатора является сложность его построения вручную.

Важными понятиями являются понятия терминального и

нетерминального символов. Терминальный символ (терминал) — символ, который может быть дан на вход анализатору пользователем. Нетерминальный символ (нетерминал) — символ, который обозначает объект языка.

Синтаксис многих языков программирования может быть определён грамматикой, которая является LR или близкой к этому, и по этой причине LR-анализаторы часто используются компиляторами для выполнения синтаксического анализа исходных кодов. LR-анализатор может быть создан из контекстно-свободной грамматики программой, называемой генератор синтаксических анализаторов, или же написан вручную программистом. Детерминированный контекстно-свободный язык — это язык, для которого существует какая-либо LR-грамматика.

LR-анализатор основан на алгоритме, приводимом в действие таблицей анализа, структурой данных, которая содержит синтаксис анализируемого языка. Таким образом, термин LR-анализатор на самом деле относится к классу анализаторов, которые могут разобрать почти любой язык программирования, для которого предоставлена таблица анализа. Таблица анализа создаётся генератором синтаксических анализаторов.

LR-анализаторы сложно создавать вручную и обычно они создаются генератором синтаксических анализаторов или компилятором компиляторов. В зависимости от того, как была создана таблица анализа, эти анализаторы могут быть названы простыми LR-анализаторами (SLR), LR-анализаторами с предпросмотром (LALR) или каноническими LR-анализаторами. LALR-анализатор имеют значительно большую распознавательную способность, чем SLR-анализаторы. При этом таблицы для SLR-анализа имеют такой же объём, что и для LALR-анализа, поэтому

SLR-анализ уже не используется. Канонические LR-анализаторы имееют незначительно большую распознавательную способность, чем LALR-анализаторы, однако требуют намного больше памяти для таблиц, поэтому их используют очень редко.

В настоящее время одним из самых популярных средств для реализации всевозможножных текстовых анализаторов является связка программных комплексов flex + bison, которая и была использована в проекте.

GNU Bison — программа, используемая для автоматического создания синтаксических анализаторов (парсеров) по входному описанию грамматики [2]. Программа bison относится к свободному программному обеспечению, она разработана в рамках проекта GNU и портирована под все традиционные операционные системы. Обычно bison используется в комплексе с лексическим анализатором flex.

Flex используется для описания базовых токенов и создания программы (кода на языке C++), обрабатывающей поток символов и выдающей поток токенов [3]. Bison используется для описания грамматики, построенной на базе алфавита токенов, и применяется для генерации программы (кода на языке C, C++ или Java), которая получает на вход поток токенов и находит в этом потоке структурные элементы (нетерминальные токены) согласно заданной грамматике.

На рис. 1 представлена схема работы всего анализатора входной спецификации. Демонстрационный отрывок исходного кода спецификации упрощен, опущены определения колонок и их типов.

Сначала входной файл подается на вход лексическому анализатору (лексеру), реализованному с помощью программы flex. Используя заданные программистом паттерны, лексер разбивает входной файл на поток токенов.

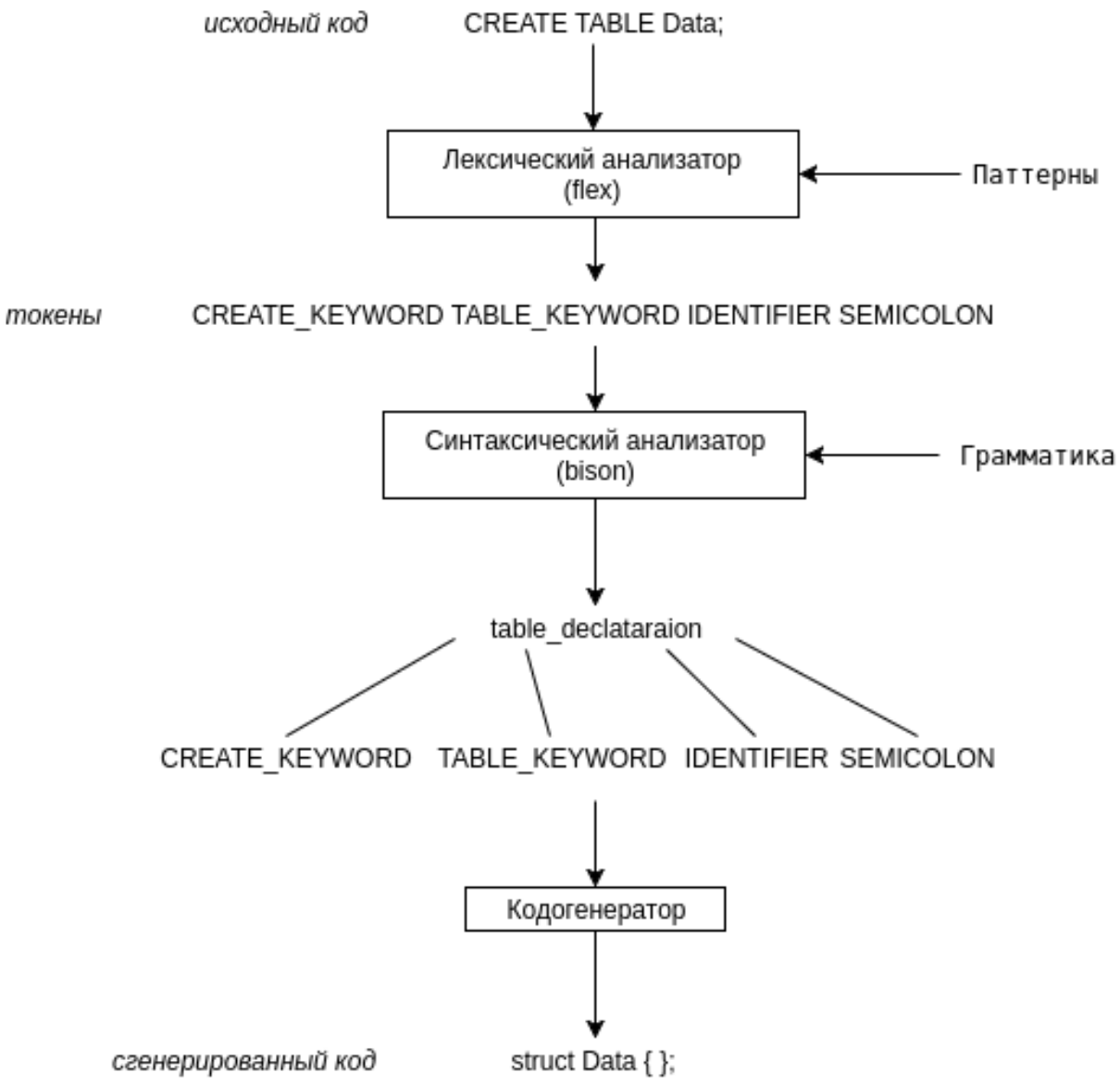

Рис. 1. Схема работы анализатора спецификации

Рассмотрим на листинге 6 установленный список правил, который использует flex при генерации лексический анализатора. Первую часть правил составляют простые правила для выделения парных скобок, двоеточия, запятой, точки, символа «@» (at) и знака равенства.

Далее следует большой блок выделения ключевых слов. Опция `?i:` указывает генератору лексического анализатора о том, что паттерн не должен учитывает регистр. Таким образом, для выделения токена `TABLE` подходит указание в спецификации как `table`, так и `TaBlE`.

Затем указан блок правил выделения идентификаторов и параметров. В качестве идентификаторов разрешается использовать любые непустые последовательности латинских букв любого регистра A-Z и a-z, цифры 0-9 и символ нижнего подчеркивания «_». Первым символом идентификатора обязана быть буква. Правила для параметров идентичны правилам для идентификаторов, но они предваряются символом «@».

Целые числа и числа с плавающей запятой определяются стандартно. Целое число — это непустая последовательность цифр из набора 0-9, а число с плавающей запятой определяется как целое число, знак точки и затем целое число.

Для поддержки комментариев введен паттерн «», обозначающий начало комментария. Начиная от этого терминала, весь остальной поток до достижения символа перевода строки игнорируется.

Листинг 6. Паттерны лексического анализатора

```
LPAREN                  "("
RPAREN                  ")"
SEMICOLON               ";"
COMMA                   ","
POINT                   "."
```

```
AT                      "@"
EQ                      "="

TABLE_KEYWORD           ?i:"table"
CREATE_KEYWORD          ?i:"create"
PK_KEYWORD              ?i:"pk"

SELECT_KEYWORD          ?i:"select"
FROM_KEYWORD            ?i:"from"
WHERE_KEYWORD           ?i:"where"
INSERT_KEYWORD          ?i:"insert"
VALUES_KEYWORD          ?i:"values"

PROCEDURE_KEYWORD       ?i:"procedure"
BEGIN_KEYWORD           ?i:"begin"
END_KEYWORD             ?i:"end"
IN_KEYWORD              ?i:"in"
OUT_KEYWORD             ?i:"out"
SET_KEYWORD             ?i:"set"

ON_KEYWORD              ?i:"on"
AND_KEYWORD             ?i:"and"
INT_KEYWORD             ?i:"int"
FLOAT_KEYWORD           ?i:"float"
DOUBLE_KEYWORD          ?i:"double"
CHAR_KEYWORD            ?i:"char"

IDENTIFIER              {IDENTIFIER_START}{IDENTIFIER_PART}*
IDENTIFIER_START        [[:alpha:]]
IDENTIFIER_PART         [[:alnum:]_]
```

```
PARAMETER               {AT}{IDENTIFIER}

INTEGER                 {DIGIT}+
FLOAT                   {DIGIT}+{POINT}{DIGIT}+
DIGIT                   [[:alnum:]]

COMMENT_START           "---"

NEWLINE                 \n
WHITESPACE              [ \t\n]+
```

Далее полученный поток токенов передается на вход синтаксическому анализатору, который в данном случае был сгенерирован при помощи программы bison. Bison преобразовывает предоставленную программистом грамматику языка, указанную на листинге 7, в синтаксической анализатор.

Листинг 7. Грамматика

```
program: program_element_list;

program_element_list: program_element
        | program_element_list program_element;
program_element: table_declaration
        | procedure_declaration;

table_declaration: CREATE_KEYWORD TABLE_KEYWORD table_name LPAREN
    column_declaration_list RPAREN SEMICOLON;
procedure_declaration: CREATE_KEYWORD PROCEDURE_KEYWORD
   procedure_name LPAREN parameter_declaration_list RPAREN
   BEGIN_KEYWORD statement_list END_KEYWORD SEMICOLON;

statement_list: statement
        | statement_list statement;
```

```
statement: insert_statement
        | select_statement;

insert_statement: INSERT_KEYWORD TABLE_KEYWORD table_name
   VALUES_KEYWORD LPAREN argument_list RPAREN SEMICOLON;
select_statement: SELECT_KEYWORD selection_list FROM_KEYWORD
   table_name WHERE_KEYWORD condition_list SEMICOLON;

parameter_declaration_list: parameter_declaration
        | parameter_declaration_list COMMA parameter_declaration;
parameter_declaration: parameter_name data_type parameter_mode;

selection_list: selection
        | selection_list COMMA selection;
selection: column_name SET_KEYWORD parameter_name;

condition_list: condition
        | condition_list AND_KEYWORD condition;
condition: column_name EQ parameter_name;

column_declaration_list: column_declaration
        | column_declaration_list COMMA column_declaration;
column_declaration: column_name data_type
        | column_name data_type PK_KEYWORD;

argument_list: argument
        | argument_list COMMA argument;
argument: parameter_name;

table_name: IDENTIFIER;
column_name: IDENTIFIER;
```

```
procedure_name: IDENTIFIER;
parameter_name: PARAMETER;

data_type: INT_KEYWORD
        | FLOAT_KEYWORD
        | DOUBLE_KEYWORD
        | CHAR_KEYWORD LPAREN INTEGER RPAREN;

parameter_mode: IN_KEYWORD
        | OUT_KEYWORD;
```

Грамматика языка достаточно проста. Весь входной текст с точки зрения генератора синтаксического анализа называется «программой» и состоит из последовательности «элементов программы».

«Элемент программы» — это либо «объявление таблицы», либо «объявление процедуры». «Объявление таблицы» состоит из последовательного набора ключевых слов `CREATE` и `TABLE`, затем следует название таблицы, которое должно являться «идентификатором», затем, обрамленный скобками, список колонок и точка с запятой.

Каждая колонка должна иметь:

- имя колонки — по паттерну «идентификатор»;
- тип колонки — `int`, `float`, `double` или `char(int)`;
- модификатор первичного ключа (необязательно) — `PK`.

«Объявление процедуры» состоит из последовательного набора ключевых слов `CREATE` и `PROCEDURE`, затем, как и в случае с таблицами, название процедуры — «идентификатор». После — список параметров процедуры, где каждый параметр представляет собой:

- имя параметра — по паттерну является «параметром», то есть идентификатором с префиксом «@»;
- тип параметра — аналогичны типам колонок;
- модификатор `IN` или `OUT` — признак, указывающий является параметр входным или выходным.

Далее, по грамматике, в «объявлении процедуры» следует «список выражений», обрамленный ключевыми словами `BEGIN` и `END` и завершающийся точкой с запятой. «Список выражений» — это последовательность INSERT- или SELECT-операций, определяющих, соответственно, операции вставки данных и выборки.

Процедура может быть одного из двух типов: «на чтение» и «на запись». При наличии в списке параметров хотя бы одного параметра с модификатором `OUT`, процедура считается «на чтение», в остальных случаях, то есть при наличии у параметром только модификаторов `IN`, —- «на запись».

У процедуры «на запись» в теле допускаются только операции вставки данных (INSERT). Если таких операций больше одной, они будут произведены по порядку перечисления.

Процедура «на чтение» допускает только операции выборки данных (SELECT). При указании нескольких операций, их результаты будут скомбинированы в одну выборку.

Пример входной спецификации указан на листинге 8.

Листинг 8. Пример спецификации

```
CREATE TABLE Person (
        id int,
```

```
        name char(4)
)

CREATE PROCEDURE add_person(@id int in, @name char(4) in)
BEGIN
        INSERT TABLE Person VALUES (@id, @name);
END;

CREATE PROCEDURE get_person(@id int in, @name char(4) out)
BEGIN
        SELECT name SET @name FROM Person WHERE id = @id;
END;
```

4.2. Хранение данных

Важным вопросом является проблема эффективного хранения данных с учетом поставленных целей о быстрой вставке и выборке данных.

Очевидно, что во всех распространенных системах управления базами данных физическое упорядочение записей индекса на диске имеет те же недостатки, что и физическое упорядочение самих записей данных: необходимость реорганизации файла базы при операциях вставки, удаления, модификации. Возможность пренебречь поддержкой операций удаления и модификации данных позволяет поменять при реализации традиционные подходы к хранению данных и формированию индексов.

В оперативной памяти упорядоченные структуры, допускающие эффективную вставку, удаление и модификацию данных обычно организуются в виде сбалансированных деревьев (АВЛ-деревьев). Сбалансированные деревья - это двоичные деревья, для которых постоянно

поддерживается равномерное распределение вершин между поддеревьями. При этом достигается близкая к теоретической эффективность поиска $\mathcal{O}(\log n)$.

Однако использование подобных структур данных для организации индексов на диске оказывается малоэффективным из-за больших затрат на ввод-вывод. Поскольку вершины располагаются не в оперативной памяти, а на внешнем устройстве, то каждое обращение к вершине при поиске требует отдельной операции чтения или записи. Таким образом, если дерево содержит, например 1 млн вершин, то в среднем может потребоваться около 20 операций обращения к внешней памяти, что является достаточно больших числом.

Логичным подходом к преодолению этой проблемы является группировка нескольких вершин дерева в один блок (страницу) ввода-вывода. При этом в древовидную структуру организуются блоки, а не отдельные вершины. Несмотря на то, что возникает необходимость производить поиск внутри блока, количество обращений к устройству внешней памяти сокращается до минимума.

Эффективной структурой хранения данных при таких ограничениях является Б-дерево, пример устройства которого изображен на рис. 2. В данном проекте была применена его модификация, отличающаяся от традиционной реализации. Так как данные никогда не модифицируются, позволительно сформировать индексное дерево лишь единожды при собственно записи его на диск.

Основным отличием Б-дерева от бинарного дерева поиска является количество потомков. В случае бинарного дерева оно ограничено двумя, для Б-дерева число может быть различным.

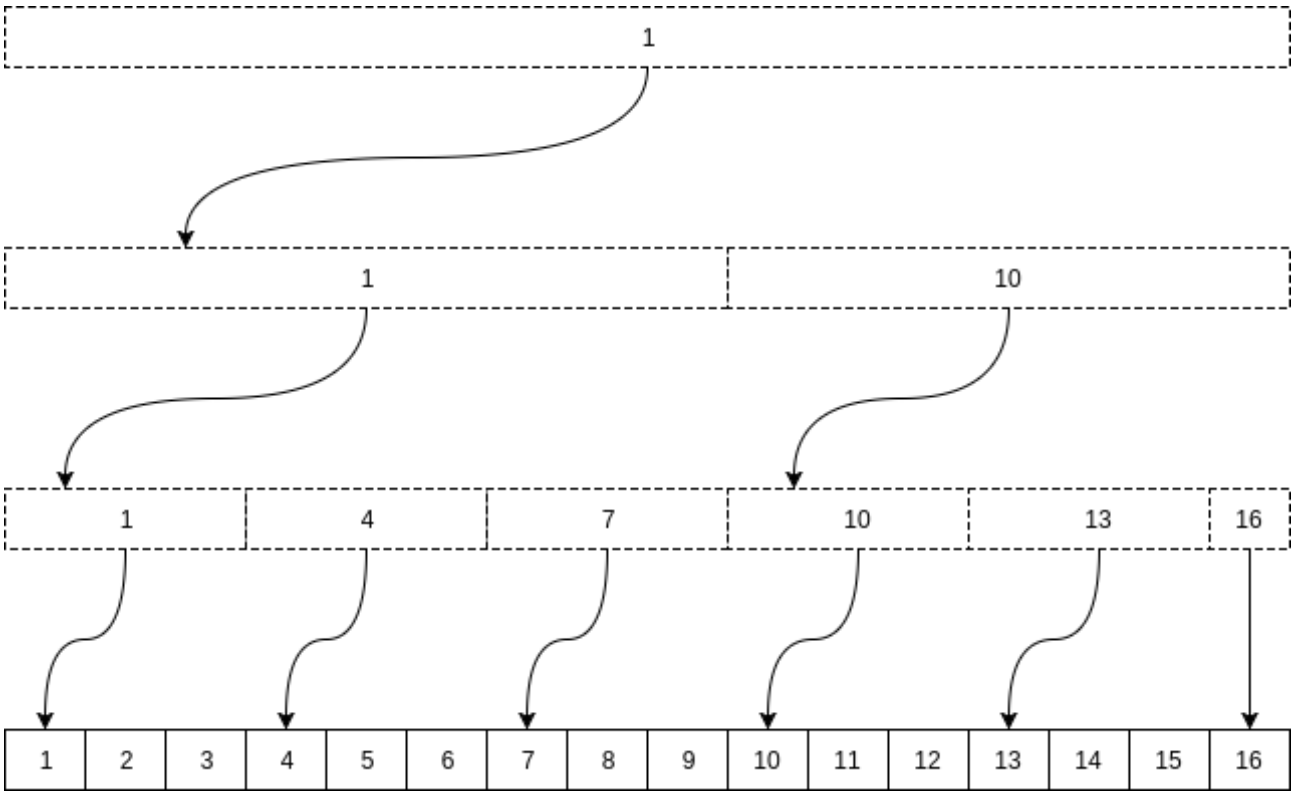

Рис. 2. Устройство Б-дерева

Во время записи данные сохраняются во временный буфер в порядке записи, не поддерживая порядок и структуру дерева. Это позволяет избегать задержек при собственно записи.

Только при закрытии базы данных на запись происходит формирование индексного дерева. При вызове процедуры закрытия база сначала сортирует весь буфер данных «быстрой» сортировкой, формируя дерево «снизу», а не «сверху», как в случае традиционной реализации. Затем становится возможным генерация узлов индексного дерева.

Количество потомков Б-дерева может быть выбрано различными способами. Эффективным способом его выбора является его динамическое вычисления на основе размера хранящихся типов данных.

Можно жестко зафиксировать желаемый размер страницы данных.

Область данных содержит файлы и каталоги, подчиненные корневому.

Вся область данных разбивается операционной системой на кластеры. Кластер — это один или несколько смежных секторов области данных. С другой стороны, кластер - это минимальная адресуемая единица дисковой памяти, выделяемая файлу. То есть, файл или каталог всегда занимает целое число кластеров. Для создания и записи на диск нового файла операционная система отводит для него несколько свободных кластеров диска. Эти кластеры не обязательно должны следовать друг за другом. Для каждого файла хранится список всех номеров кластеров, которые предоставлены данному файлу.

В современных файловых системах эта проблема решается за счет ограничения размера кластера (максимум 4 Кбайта).

Если жестко зафиксировать желаемый размер страницы данных размером кластера, то есть 4 Кбайтами, тогда количество записей на одной странице данных будет равно

$$N = 4 \text{ Кбайт}/size_{struct},$$

где $size_{struct}$ — размер одной записи.

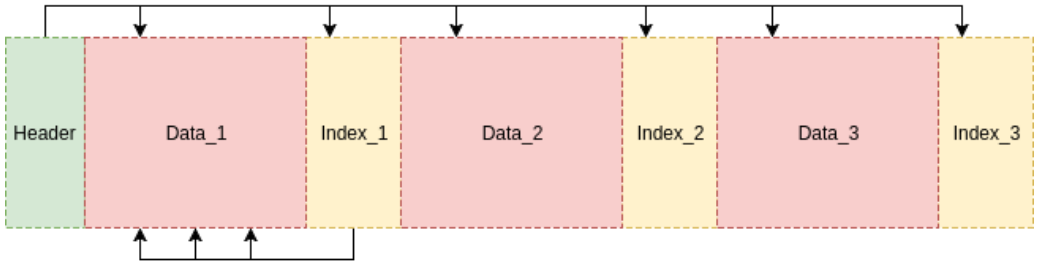

Рис. 3. Файл базы данных

Формируемый файл, пишущийся на диск, имеет вид, указанный на рисунке 3. Пример приведен для случая с тремя таблицами. В начале

файла располагается заголовок фиксированного размера, который содержит информацию о количестве записей в каждой таблице и смещениях, его строение указано на листинге 14. Далее идут блоки, относящиеся к каждой из таблиц — сначала данные, потом индекс.

Заголовок файла содержит данные о количестве записей в каждой из таблиц и смещениях относительно начала файла данных и индексного дерева каждой таблицы. Данные пишутся вплотную друг к другу, используя сгенерированные структуры. Узлы индекса записываются в обратном порядке, то есть сначала происходит запись листовых узлов, а корень дерева распологается самым последним.

4.3. Генерация целевого кода

Анализируя входную спецификацию, генерируется целевой код на языке Си. Рассмотрим пример полученного кода, используя спецификацию из листинга 8, включающую в себя объявление одной таблицы с двумя колонками и двух процедур, одна на запись и одна на чтение по ключу.

Описание таблицы практически напрямую транслируется в описание структуры на языке Си (листинг 9). При записи экземпляры таких структур пишутся на диск.

Листинг 9. Структура по описанию таблицы

```
struct Person {
        int32_t id;
        char name[32];
};
```

Для последующей сортировки структур генерируется функция-компаратор (листинг 10). Функция возвращает 0 в случае равенства функций, 1 — если первый аргумент «больше» и -1 — если «больше» второй аргумент.

Листинг 10. Функция-компаратор

```
int Person_compare(struct Person* s1, struct Person* s2) {
        if (s1->id > s2->id)
                return 1;
        else if (s1->id < s2->id)
                return -1;

        if (strncmp(s1->name, s1->name, 32) == 1)
                return 1;
        else if (strncmp(s1->name, s1->name, 32) == -1)
                return -1;
```

```
        return 0;
}
```

Индекс данных, как было описано, представляет собой Б-дерево. Узлом дерева является экземпляр структуры, указанной на листинге 11. Поле `key` — это значение ключа записи, на которую указывает узел, а поле `offset` – смещение в байтах это записи относительно начала. Такая структура пишется на диск.

Листинг 11. Узел индексного дерева на диске

```
struct Person_tree_item {
        int32_t key;
        uint64_t offset;
};
```

При открытии файла на чтение весь индекс копируется в оперативную память. Для хранения в оперативной памяти генерируются экземпляры другой структуры (листинг 12), которая включает в себя как часть структуры индексного дерева, прочитанные с диска. Здесь `childs` — это указатель на список дочерних узлов, а `n` — их количество.

Листинг 12. Узел индексного дерева в ОЗУ

```
struct Person_Node {
        struct Person_tree_item data;
        struct Person_Node** childs;
        uint64_t n;
};
```

Для работы с данными (записи или чтения) должен быть открыт некоторый файл данных. База данных по условию должна поддерживать одного «писателя» и много «читателей», поэтому было введено понятие

дескриптора, отличающего одного читателя от другого. Здесь под дескриптором нужно понимать экземпляр структуры данных, имеющий ссылку на файл, описание модификатора открытого файла (чтение или запись), ссылку на заголовок файла и набор ссылок на корень индексных деревьев всех имеющихся таблиц (листинг 13).

Листинг 13. Дескриптор

```
struct sample_handle {
        uint32_t id;
        FILE* file;
        int mode;
        struct MILLDB_header* header;
        struct Person_Node* Person_root;
};
```

Заголовок, в свою очередь, содержит данные о количестве записей в каждой из таблиц и смещениях относительно начала файла данных и индексного дерева каждой таблицы (листинг 14).

Листинг 14. Заголовок файла

```
struct MILLDB_header {
        uint64_t count[1];
        uint64_t data_offset[1];
        uint64_t index_offset[1];
};
```

Описания процедур транслируются в функции языка Си. Из процедур «на запись» генерируются функции, не возвращающие значений и принимающие в качестве аргументов перечисленные параметры процедуры (листинг 15).

Листинг 15. Операция вставки данных

```
void add_person_1(int32_t id, const char* name) {
```

```
        struct Person* inserted = Person_new();
        inserted->id = id;
        memcpy(inserted->name, name, 4);
        Person_buffer_add(inserted);
}

void add_person(int32_t id, const char* name) {
        add_person_1(id, name);
}
```

Из процедур «на чтение» в целевом коде образовывается пара функций: `PROCEDURENAME_init` и `PROCEDURENAME_next` (листинг 16). Для корректного вызова процедуры сначала вызывается функция с суффиксом `_init`, принимающая в качестве аргументов указатели на итератор (листинг 17) и дескриптор и входные параметры, а далее данные просматриваются с помощью конструкции while и вызова функции с суффиксом `_next`. Функция с суффиксом `_next` возвращает `1`, когда итератор содержит текущую запись из выборки, и `0`, когда выборка завершилась.

Листинг 16. Операция выборки данных

```
void get_person_init(struct get_person_out* iter, struct
   sample_handle* handle, int32_t id) {
        memset(iter, 0, sizeof(*iter));
        iter->handle = handle;
        iter->set = NULL;
        iter->size = 0;
        iter->count = 0;

        get_person_1(iter, id);
}
```

```
int get_person_next(struct get_person_out* iter) {
        struct sample_handle* handle = iter->handle;
        if (iter->set != NULL && iter->count < iter->size) {
                memcpy(&iter->data, iter->set, sizeof(struct
                   get_person_out_data));
                iter->count++;
                return 1;
        } else
                free(iter->set);
        return 0;
}
```

Листинг 17. Итератор

```
struct get_person_out {
        struct sample_handle* handle;
        struct get_person_out_data* set;
        int size;
        int count;
        struct get_person_out_data data;
};
```

Как было указано, для вставки и выборки данных файл должен быть открыт на запись или чтение соответственно.

Для открытия и закрытия файла для записи используется пара функций `void sample_open_write(const char* filename)` и `void sample_close_write(void)` (листинг 18).

Во время открытия формируется глобальный дескриптор для «читателя», сохраняется информация о файле, инициализируются буферы для вставки данных всех таблиц. При закрытии происходит сохранение данных, формирование индексов, запись в файл и очистка памяти.

Листинг 18. Открытие и закрытие файла на запись

```
void sample_open_write(const char* filename) {
        FILE* file;
        if (!(file = fopen(filename, "wb")))
                return;
        /* fill sample_write_handle */
        Person_buffer_init();
}

void sample_close_write(void) {
        sample_save(sample_write_handle);
        Person_buffer_free();
        fclose(sample_write_handle->file);
        free(sample_write_handle);
}
```

Для того, чтобы открыть и закрыть файл для чтения используется так же пара функций `struct sample_handle* sample_open_read(const char* filename)` и `void sample_close_read(struct sample_handle* handle)` (листинг 19). Во время открытия на чтение формируется дескриптор и загружаются в оперативную память индексы, из функции возвращается полученный дескриптор.

Листинг 19. Открытие и закрытие файла на чтение

```
struct sample_handle* sample_open_read(const char* filename) {
        FILE* file;
        if (!(file = fopen(filename, "rb")))
                return NULL;

        struct sample_handle* handle = malloc(sizeof(struct
           sample_handle));
```

```
        /* fill sample_write_handle */

        fseek(handle->file, 0, SEEK_SET);
        struct MILLDB_header* header = malloc(MILLDB_HEADER_SIZE)
           ;
        fread(header, MILLDB_HEADER_SIZE, 1, handle->file);
        handle->header = header;

        Person_index_load();

        return handle;
}

void sample_close_read(struct sample_handle* handle) {
        fclose(handle->file);
        if (handle->Person_root)
                Person_index_clean(handle->Person_root);
        free(handle->header);
        free(handle);
}
```

5. Тестирование

Тестирование полученной базы данных проводилось в два этапа. На первом этапе проводилось сравнение характеристик с экземпляром базы данных, применяемой в реальных условиях для сбора информации о сетевом трафике и разработанной специально для этих целей. На втором этапе база данных, разработанная в рамках проекта, была сравнена с популярной и распространенной базой данных MySQL.

5.1. Сравнение с аналогичной разработкой

Как было указано выше, аналогичной разработкой в данном пункте является экземпляр базы данных, который был специально разработан для мониторинга трафика и в данный момент успешно применяется.

Тестирование проводилось с помощью рабочей машины, характеристики которой приведены на листинге 20.

Листинг 20. Характеристики производственной машины

```
Intel(R) Xeon(R) CPU E5-2690 0 @ 2.90GHz x 32, x86_64, OS Fedora
   16, compiler gcc (GCC) 4.6.3 20120306 (Red Hat 4.6.3-2), flags
    -O3, -std=gnu99
```

Для тестирования была создана таблица, имеющая 8 целочисленных полей, одно из них являющееся первичным ключом. В таблице 1 указаны результаты замеров скорости операции вставки и размер данных для 8 725 695 записей. Замеры скорости выборки не проводились, так как тестирование выявилось ошибки в реализации, устраненные после проведения контрольных замеров.

Таблица 1. Сравнение с аналогичной разработкой

Характеристика	**Корпоративная БД**	**Разработанная БД**
Скорость операции вставки, с	18.52	11.27
Размер данных, Мбайт	95.12	350.12

Как видно из результатов тестирования, разработанная БД осуществляет запись данных более чем в полтора раза быстрее, но размер записанных на диск данных превышает более чем в 3 раза. В большей степени, и первый, и второй факт являются следствием применения корпоративной базой данных техник сжатия данных. Это, с одной стороны, позволяет экономить память при хранении, но, с другой стороны, занимает дополнительное время при обработке.

5.2. Сравнение с MySQL

В данном пункте выполнялось сравнение с популярной и широко распространенной базой данных MySQL. Использовалась стандартная конфигурация MySQL из репозитория Ubuntu.

Тестирование проводилось на рабочей машины, характеристики которой приведены на листинге 21.

Листинг 21. Характеристики рабочей машины

```
Intel(R) Core(TM) i3-4010U CPU @ 1.70GHz x 4, x86_64, OS Ubuntu
   16.04, compiler gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6
   ubuntu1~16.04.4), flags -O3
```

Для тестирования была создана таблица, имеющая 2 поля `int` и `char(4)`. На рисунках указаны результаты замеров:

- скорость вставки данных (рис. 4);

- скорость выборки всех данных (рис. 5);
- скорость выборки данных по ключу (рис. 6);
- размер заполненной базы данных на диске (рис. 7).

Из результатов тестирования очевидно, что разработанная БД осуществляет по всем протестированным параметрам превосходит MySQL, по некоторым даже на порядки. Это объясняется гораздо более узкими задачами, которые решает разработанная база данных.

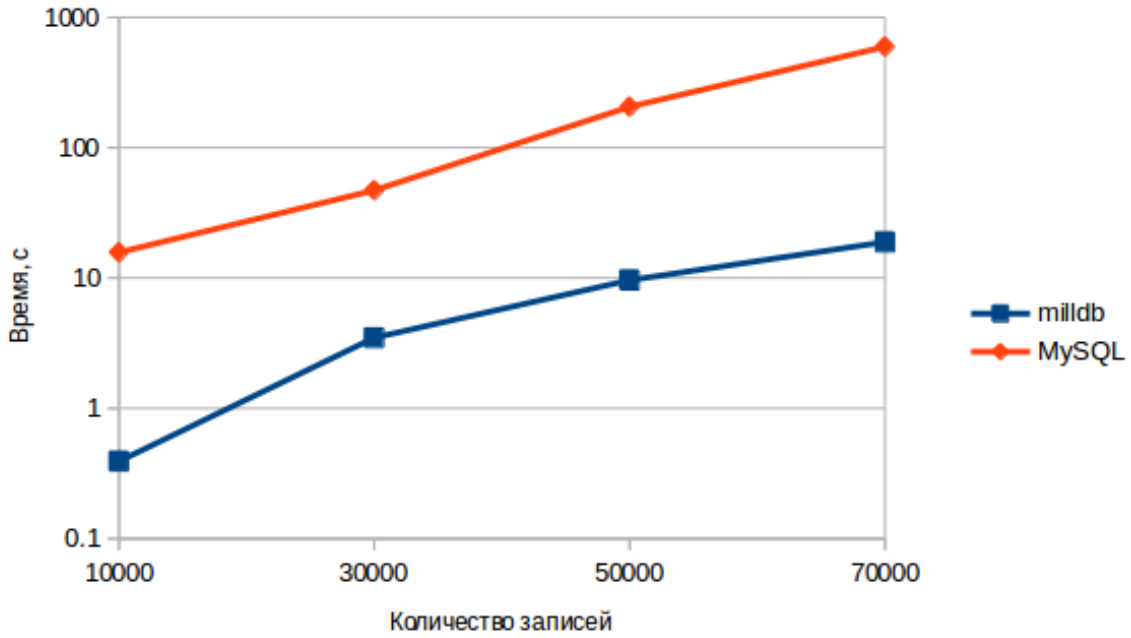

Рис. 4. Тест на вставку данных

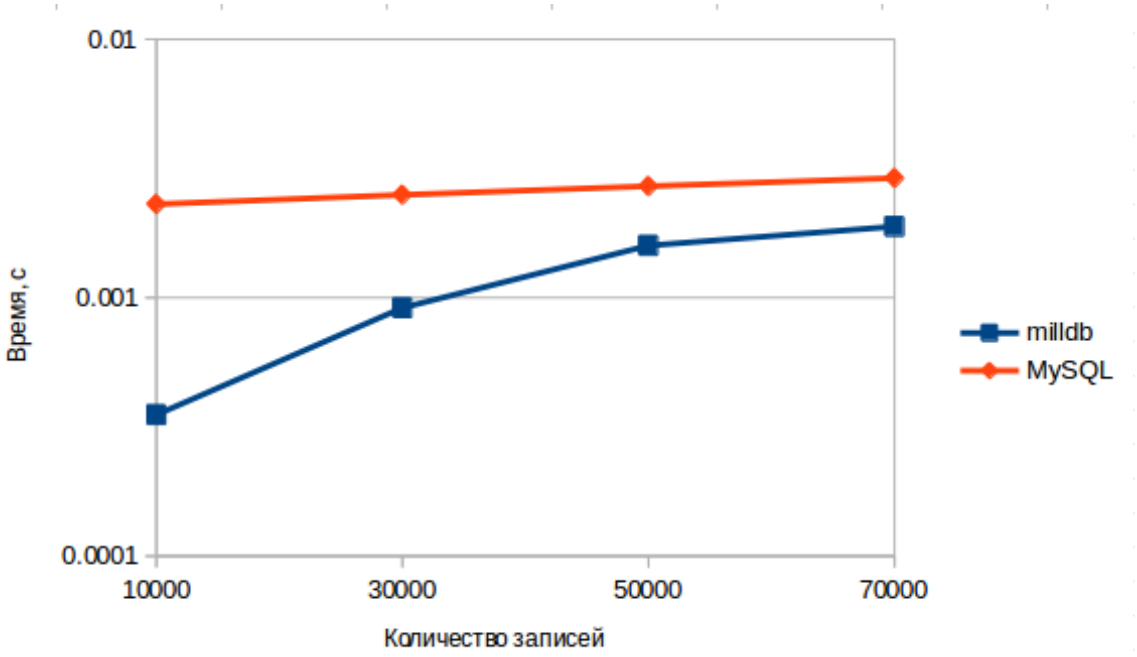

Рис. 5. Тест на выборку всех данных

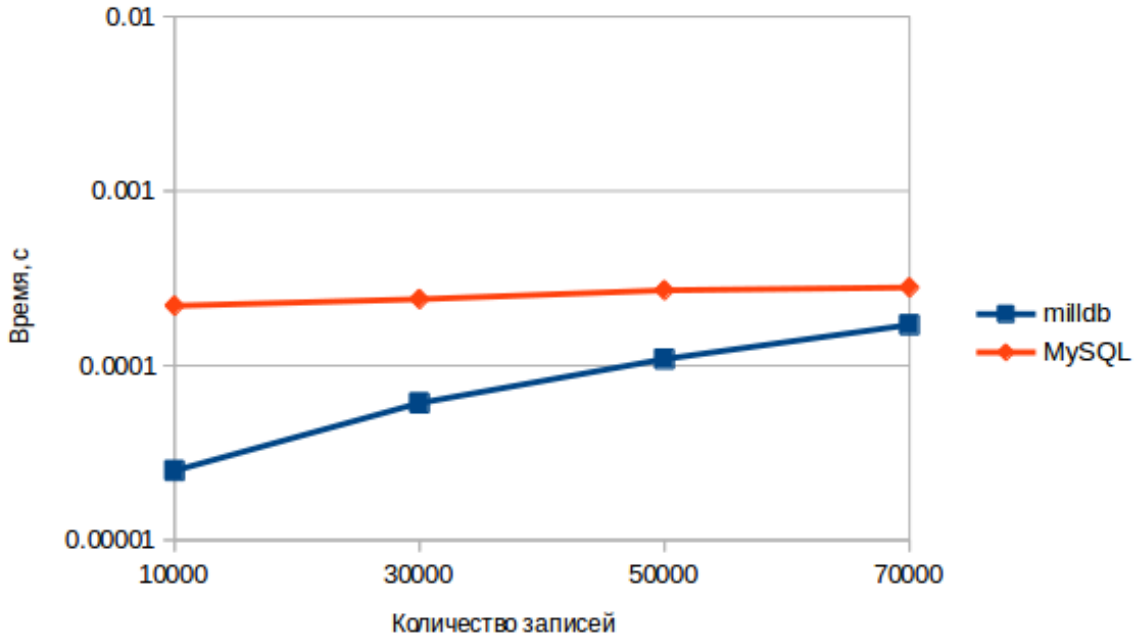

Рис. 6. Тест на выборку данных по ключу

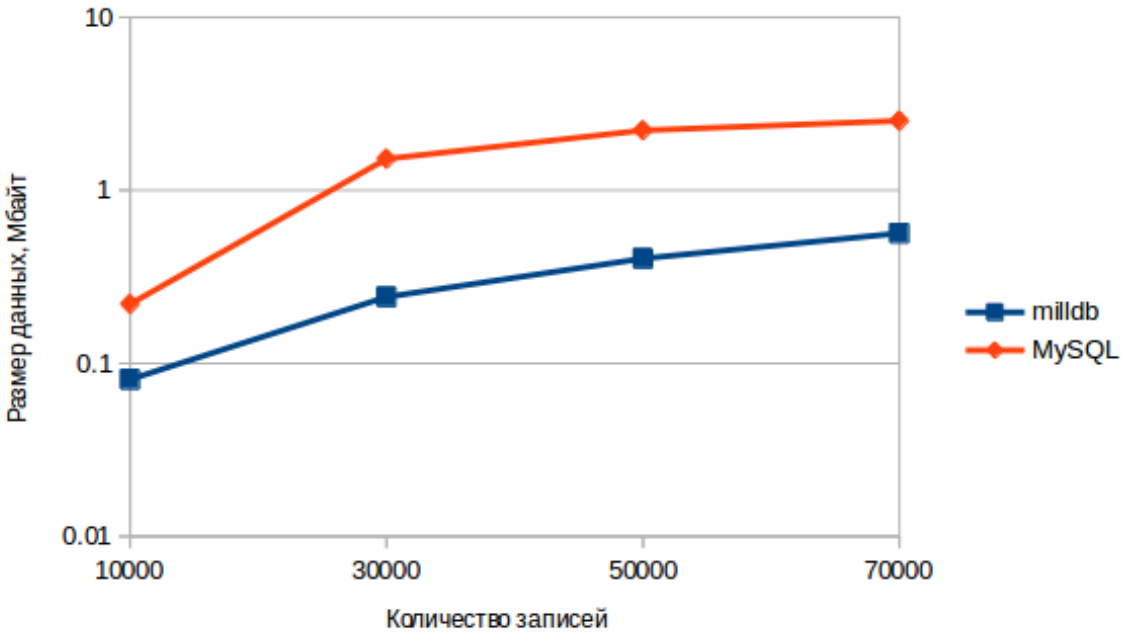

Рис. 7. Тест на размер базы данных из таблицы

Заключение

Подводя итоги, можно сказать, что в данной работе был разработан механизм высокоэффективной записи, хранения и поиска данных. Такой опыт, безусловно, будет полезным для приложения во многих сферах.

Задачи, поставленные ранее, были успешно решены, а именно: была разработана база данных, позволяющая записывать и считывать данные по заранее известной схеме данных. Вдобавок к этому, был разработан генератор библиотеки базы данных, благодаря которому пользователь может сам опеределять формат и тип обрабатываемых данных, что способствует удобству поддержки кода.

В процессе разработки базы данных и ее генератора были рассмотрены различные структуры данных, изучены и реализованы алгоритмы и применены техники. Совместимость языка Си, на котором производится код генератора, со многими другими высокоуровневыми языка позволяет говорить о множестве проектов его приложения.

Результат разработки можно оценивать как положительный, так как тесты показывают большое преимущество по различным характеристикам перед другими популярными программными комплексами. В разы отличающиеся в лучшую сторону показатели скорости работы и затрат памяти позволяют утверждать, что полученная реализация базы данных может быть применимой на практике.

Дальнейшее развитие поднятой в данной работе темы может иметь несколько различных направлений:

- поддержка других возможностей, доступных в других базах данных и не противоречащих идеям базы данных, созданной в рамках данного

проекта, таких как, например, создание дополнительных индексов;

- усовершенствование применяемых алгоритмов и структур данных с учетом специфик применения базы данных;

Как видно, проект может быть усовершенствован в дальнейшем, используя свои основные идеи как конкурентное преимущество перед аналогичными продуктами в индустрии.

Список литературы

[1] Генератор высокопроизводительной NoSQL базы данных [Электронный ресурс] —- Режим доступа: https://github.com/bmstu-iu9/mill-db

[2] GNU Bison [Электронный ресурс] —- Режим доступа: https://www.gnu.org/software/bison/

[3] Flex [Электронный ресурс] —- Режим доступа: https://www.gnu.org/software/flex/

[4] Прамодкумар Дж. Садаладж, Мартин Фаулер. NoSQL. Новая методология разработки нереляционных баз данных. Пер. с англ. — М: Вильямс, 2017. — 192 с., ил.

[5] Джеффри Д. Ульман, Дженнифер Уидом. Реляционные базы данных. Пер. с англ. — М: Лори, 2014. — 384 с., ил.

[6] Альфред Ахо, Рави Сети, Джеффри Ульман, Моника Лам. Компиляторы. Принципы, технологии и инструментарий. Пер. с англ. — М: Вильямс, 2014. — 1184 с., ил.