



ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ

ДИПЛОМНА РАБОТА

по професия код 481020 „Системен програмист“
специалност код 4810201 „Системно програмиране“

Тема: Система за съхранение, извличане и обработка на големи
графовобазирани данни.

Дипломант:

Стоян Тинчев Тинчев

Дипломен ръководител:

инж. Любомир Стоянов

СОФИЯ

2022

Използвани съкращения

На български:

- АЛУ – аритметико-логически устройство/ва
- СУБД - Система за управление на бази данни
- ИТ - информационни технологии

На английски:

- CRUD - create, read, update and delete operations
- API – application programming interface
- SQL – structured query language
- ACID – Atomicity, Consistency, Isolation, Durability
- CI - continuous integration
- PPM - prediction by partial matching (compression algorithm)
- RAM - random access memory
- ASCII - American Standard Code for Information Interchange

Увод

Един от най-ценните ресурси е времето. Ръка за ръка с него вървят ефикасността и сложността на дадена система. Трябва да се постигне баланс, като всеки ресурс зависи до известна степен от друг.

В света на програмирането и технологиите е много важно съхранението и правилното управлението на паметта. Това са едни от най-ценните ресурси. Съхраняването, извлечането и менажирането на действията изпълнявани от компютрите са задачи, обвързани със сложни изчислителни алгоритми, синхронизация между много асинхронни операции, методи за комуникация между отделните АЛУ и много други обработки, които в повечето случаи изискват голяма изчислителна сила.

Данните в днешно време са много и навсякъде. Те могат да бъдат под формата на показатели, регистрационни файлове, обекти, ключове и стойности. Може да идват от сървъри, вътрешни приложения, уеб страници, от клиенти или от още безброй източници.

Откъдето и да идват, те трябва да се съхраняват, обработват, индексират и след това да се надграждат. Така наречените CRUD операции са нужни на организациите, които следят клиентски данни, сметки, информация за плащане, здравна информация и други записи, изискващи хардуер за съхранение на данни и приложения, които осигуряват постоянно съхранение. Тези данни обикновено са съхранени в база данни, която е просто организирана колекция от данни, които могат да се разглеждат по електронен път. Има много видове бази данни: йерархични бази данни, графични бази данни, обектно-ориентирани бази данни... Най – общо се делят на релационни и нерелационни.

Прилаганият тип база данни най-често е релационна база данни, която се състои от информация в таблици в редове и колони, свързани с други таблици с допълнителна информация, чрез система от ключови думи, която включва първични ключове и външни ключове.

Разглежданата дипломна работа цели реализация на приложно програмен интерфейс (API), който да извлича, обработва и съхранява безопасно и бързо големи графовобазирани данни. Трябва да е достъпна под формата на библиотека и да предоставя възможността на потребителя, по разбираем и удобен начин, да създаде нужната за него база данни.

Сред функционалните изисквания са алгоритъм за компресиране, алгоритъм декомпресиране на данните, имплементация на фрагментно декомпресиране, изграждане на сървърна архитектура (Клиент - Сървър модел), метод за изпълнение на заявките, синхронизация и комуникация между изчислителните стълбове.

Първа глава

Методи при съхраняването на големи обеми от данни (СУБД)

Данните са колекция от отделна малка единица информация. Всичко започва от най-малката единица за съхранение, а именно един бит, който съдържа 2 състояния. След него паметта скалира. Има различни форми съхранявана информация като текст, числа, медии, файлове и др.

В изчисленията, данните са информация, която може да бъде преведена във форма за ефективно движение и обработка.

1.1 СУБД

Базата данни [1] е организирана колекция от данни, така че да може да бъде лесно достъпна и управлявана. Въпросните данни може да се организирани в таблици, редове, колони... Трябва да бъдат лесно достъпни за индексиране от обикновения потребител, за да се улесни намирането на подходяща информация.

Съвременните бази данни се управляват от системата за управление на бази данни (СУБД). Основната цел на СУБД е да оперира с голямо количество информация чрез съхранение, извлечане и управление на данни.

1.1.1 Характеристиките на една СУБД са:

- Използва се цифрово хранилище, установено на сървър, за да се съхранява и управлява информацията
- Ясен и структуриран поглед върху процеса, който управлява данните
- Процедури за възстановяване на данните
- Сигурност на данните
- ACID свойства, които поддържат и се грижат за данните

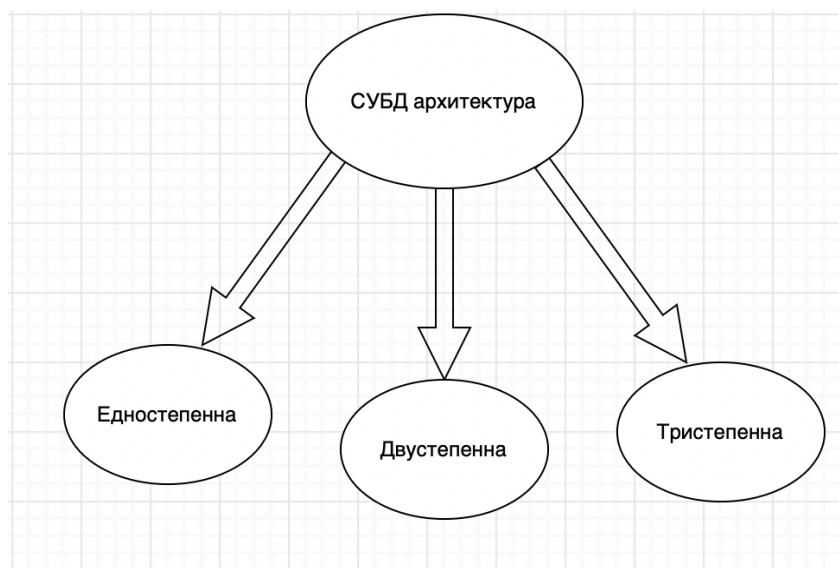
1.1.2 Положителните страни на една СУБД са:

- Споделяне на данни – в СУБД упълномощените потребители на организация могат да споделят данните между множество потребители
- Лесна поддръжка – може лесно да се поддържа поради централизираният характер на системата за база данни
- Архивиране – предоставя подсистеми за архивиране и възстановяване
- Множество потребителски интерфейси – предоставят се широк набор от потребителски интерфейси

1.1.3 Отрицателните страни на една СУБД са:

- Нужда от добър хардуер и софтуер – изисква се висока процесорна мощ за обработката на всички процедури
- Размер – при неправилна конфигурация и неефективна реализация на една СУБД може да се заема голямо пространство
- Сложност – сложна реализация

1.2 Архитектура на СУБД



Фиг. 1.1 Видове архитектури на една СУБД

Дизайнът на СУБД зависи от нейната архитектура. В повечето случаи клиент-сървър се използва за работа с голям брой компютри, уеб сървъри, сървъри на бази данни и други компоненти, които са свързани с мрежи.

Архитектурата клиент-сървър се състои от много компютри и работна станция, които са свързани през мрежата. Архитектурата на СУБД зависи от това как потребителите са свързани към базата данни, за да изпълняват своите заявки.

1.2.1 Едностепенна архитектура

В тази архитектура базата данни е директно достъпна за потребителите. Това означава, че потребителят може директно да достъпи СУБД и да я използва.

Всички промени, направени тук, ще бъдат направени директно в самата база данни. От една страна няма удобен инструмент за крайните потребители, но от друга – всичко става много експедитивно и бързо.

Едностепенната архитектура обикновено се използва за разработване на локално приложение, където програмистите да могат директно да комуникират с базата данни за бързи реакции.

1.2.2 Двустепенна архитектура



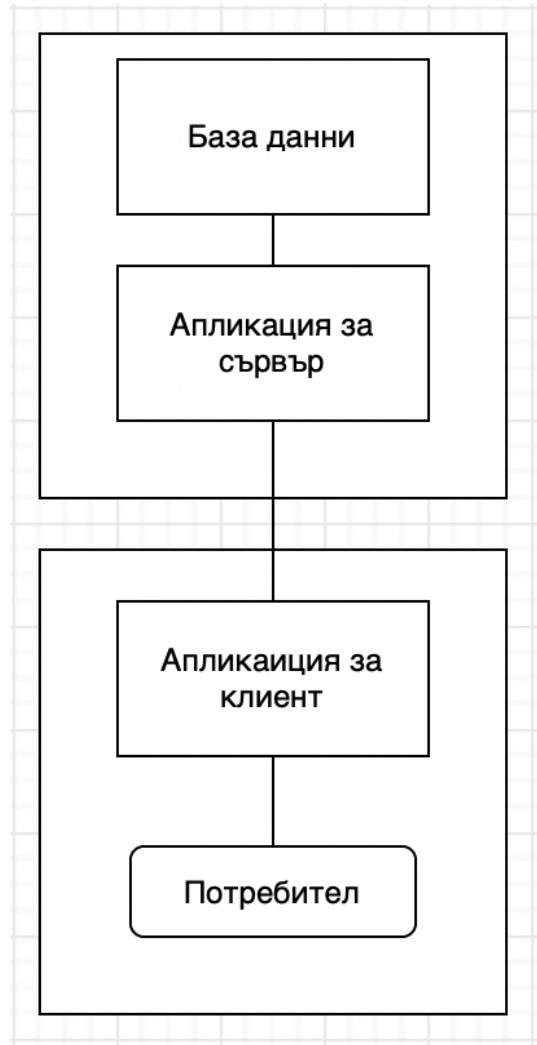
Фиг. 1.2 Двустепенна архитектура на една СУБД

Тази архитектура е базирана на клиент-сървър. В нея приложенията от страна на клиента могат директно да комуникират с базата данни от страна на сървъра.

За това взаимодействие обикновено се използват API. Потребителските интерфейси и приложните програми се изпълняват от страна на клиента, докато сървърната страна е отговорна за предоставянето на функции като обработка на заявки и управление на транзакции.

За успешна комуникация – приложението от страна на клиента установява връзка със сървъра.

1.2.3 Тристепенна архитектура



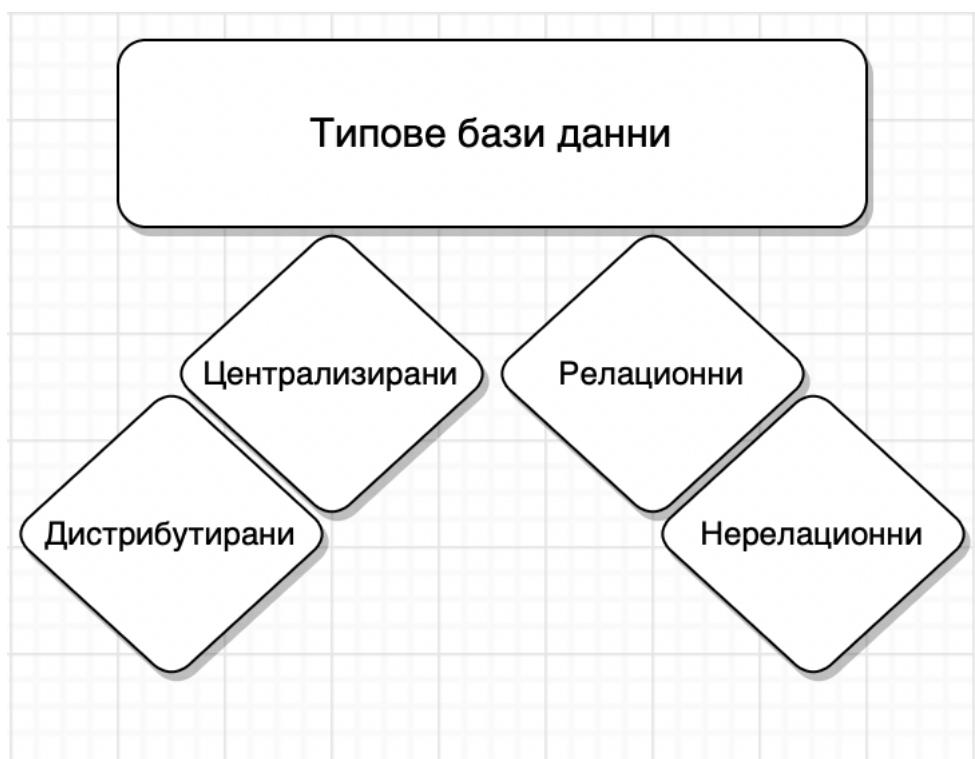
Фиг. 1.3 Тристепенна архитектура на една СУБД

Тристепенната архитектура съдържа друг слой между клиента и сървъра. При нея клиентът не може директно да комуникира със сървъра.

Приложението от страна на клиента взаимодейства със сървърната апликация, като последният допълнително комуникира със системата на базата данни. Така крайният потребител няма представа за съществуването на базата данни извън приложението на сървъра. Базата данни също няма представа за друг потребител извън приложението.

Такава архитектура най-често се използва в случай на голямо уеб приложение.

1.3 Типове СУБД



Фиг. 1.4 Типове СУБД

Има различни видове бази данни, използвани за съхранение на различни видове данни. Всеки вид има своите специфики, които намират употреба при различните нужди и обстоятелства. Някой от по-основните видове са централизирани, дистрибутирани, релационни и нерелационни бази данни.

1.3.1 Централизирани СУБД

Това е тип база данни, която съхранява данни в централизирана система. Така потребителите имат „комфорта“ да имат достъп до съхранените данни от различни места чрез няколко устройства. В устройствата най-вероятно чрез приложение се извършва достъп, който е регулиран чрез процес на удостоверяване. Такъв процес позволява на потребителите да имат сигурен достъп до данните.

Пример за такъв тип база данни може да бъде някоя „Централна библиотека“, която съхранява данните на всяка книга/ списание...

Плюсове от такъв тип съхранение са:

- Намалява се рисъкът от управлението на данни, тоест манипулирането отделни части няма да повлияе на основните данни
- Последователността се поддържа, тъй като всичко е на централно хранилище
- Осигурява се добро качество на информацията, което позволява да се установят определени стандарти
- По-евтино е, тъй като се изискват по-малко доставчици за обработка на информацията

Недостатъците, от друга страна, са:

- Размерът на централизираната база данни е голям, което увеличава времето за реакция за извлечане на информация
- Не е лесно да се извърши актуализация на толкова голямо хранилище
- Ако възникне грешка в сървъра, цялата информация ще бъде загубена

1.3.2 Дистрибутирани СУБД

За разлика от централизираната система за бази данни, в дистрибутиранныте системи данните се разпределят между различни системи (компютри) на една

организация. Тези системи са свързани чрез комуникационни връзки, които помагат на крайните потребители да имат лесен достъп.



Фиг. 1.5 Видове дистрибутирани бази данни

Освен това, дистрибутираната СУБД може да се раздели на хомогенна и хетерогенна.

- Хомогенни са тези системи, които се изпълняват на една и съща операционна система, използват един и същ потребителски интерфейс и носят същите хардуерни характеристики
- Хетерогенни са тези системи, които се изпълняват на различни операционни системи при различни потребителски интерфейси и носят различни хардуерни специфики

Предимства при дистрибутираната СУБД:

- Могат да се добавят допълнително „модули“, тоест системата може да бъде разширена чрез включване на нови компютри и свързването им към разпределена система
- Потенциална грешка в сървъра няма да засегне целия набор от данни

1.3.3 Релационни СУБД

Тази база данни се основава на релационния модел на данни. Това значи, че информацията се съхранява под формата на редове и колони, като заедно образуват таблица (връзка). Релационната база данни използва SQL за съхранение, манипулиране и поддържане на данните. Всяка таблица в базата носи ключ, който прави всеки отделен фрагмент информация уникатен от останалите. Примери за релационни бази данни [2] са MySQL, PostgreSQL, Oracle и други.

Релационният модел има четири основни свойства, известни като ACID, които разграничават SQL от NoSQL.

1.3.4 Нерелационни СУБД

Това са бази данни, които се използват за съхранение на широк спектър от набори от данни. Такива бази данни не са релационни, тъй като не съхраняват информацията само в табличен вид, но и по няколко други различни подхода.



Фиг. 1.6 Видове нерелационни бази данни

Както дистрибутирани бази данни, така и нерелационните се делят на видове.

За разлика от дистрибутирани, обаче, тук видовете са 4.

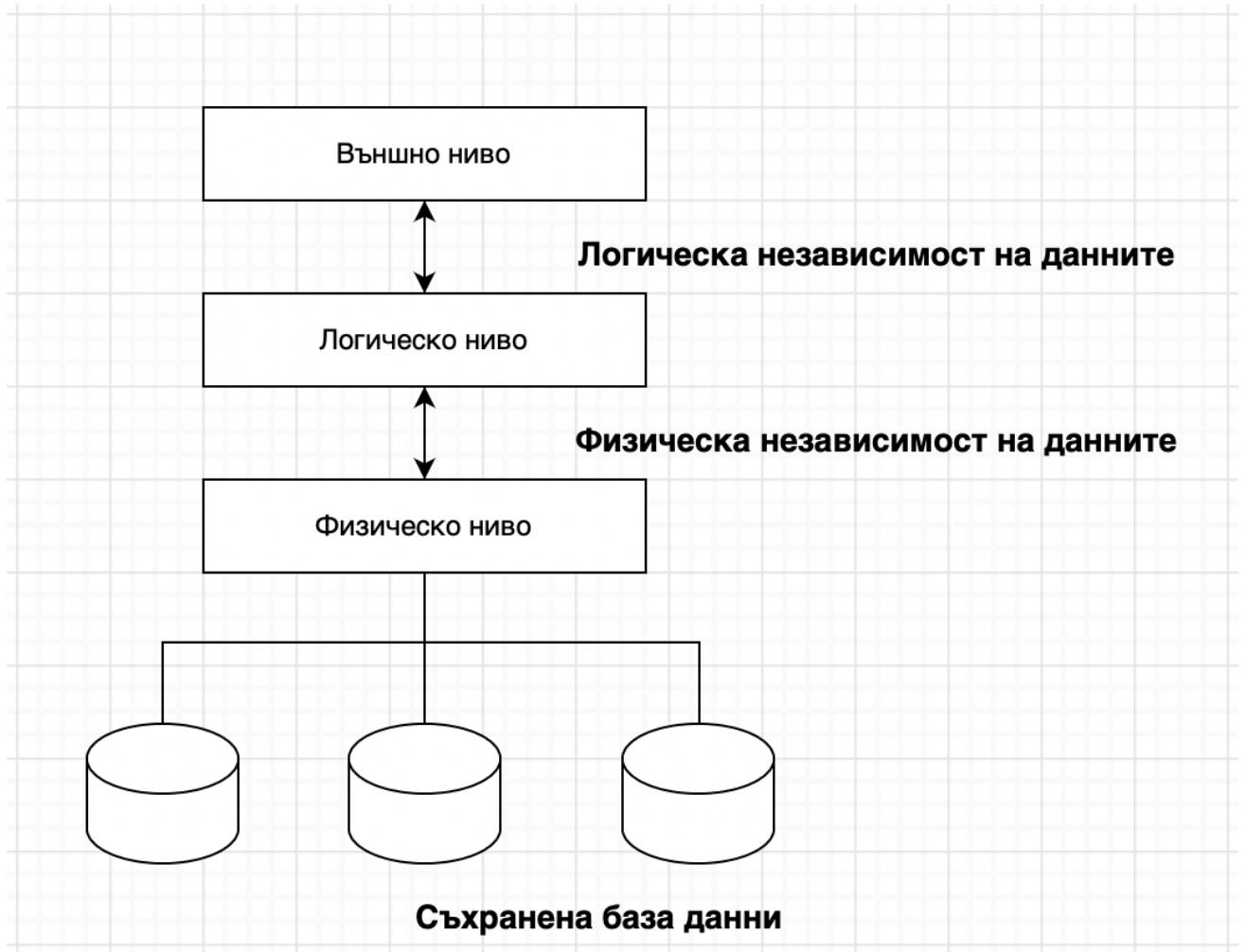
- *Съхранение ключ-стойност* е най-простият тип съхранение на база данни, където всеки отделен елемент е ключ (или име на атрибут), който има невидима връзка със стойността, която е асоциирана за него.
- *Файлово базираната база данни* се използва за съхраняване на данни като JSON документ.
- *Графовата база данни* се използва за съхраняване на огромни количества данни в структура, подобна на граф. Намира приложение най-често при сайтовете за социални мрежи.
- *Базата данни с големи колони* е подобна на данните представени в релационните бази данни. Тук данните се съхраняват заедно в огромни колони, вместо в редове.

Някой от предимствата при тази конфигурация са:

- Добрата производителност при разработването на приложения, тъй като не се изисква съхранение на данните в структуриран формат.
- По-добър вариант е при управление и обработка на големи набори от данни.
- Осигурява се висока скалируемост.
- Потребителите имат бърз достъп до данни чрез връзката ключ-стойност.

1.4 Независимост на данните

Независимостта на данните е тази възможност, която позволява да се модифицира схемата на едно ниво от системата на базата данни, без да се променя схемата на следващото по-високо ниво.



Фиг. 1.7 Независимост на данните

Има два типа независимост на данните - Логическа и Физическа.

1.4.1 Логическа независимост на данните

Независимостта на логическите данни се отнася до възможността да се променя концептуалната схема, без да се налага да се променя външната схема. Такава независимост се използва за отделяне на външното ниво (нива) от цялата концепция/изглед.

Важна характеристика е, че ако се направят някакви вътрешни промени (концептуални) на данните, тогава потребителският изглед няма да бъде засегнат.

1.4.2 Физическа независимост на данните

Физическата независимост на данните може да се дефинира като способност за промяна на вътрешната схема, без да се налага промяна на концептуалната схема. Тоест, ако се направят някакви промени в размера на съхранение на системния сървър на базата данни, тогава концептуалната структура на базата данни няма да бъде засегната.

Втора глава

СУБД и нейните особености

2.1 Технологични изисквания

Технологичните изисквания са от съществено значение. Нужно е много добре съгласуване и обмисляне на наличния проблем, който трябва да бъде решен.

При реализацията на една СУБД има няколко основни фактора, които трябва да бъдат взети под внимание и максимално изпълнени, а именно - ефикасност, бързодействие, оптимална употреба на ресурсите и достъпност. Последните четири са изисквания в ИТ света, които търсят все повече нови разработки, целящи подобряването, усъвършенстването и оптимизацията на основните прилагани принципи.

Технологиите, избрани при разработката трябва да имат следните характеристики:

- Бързина на програмния език
- Възможност за директен достъп до паметта
- Възможност за изпълнение на заявки (клиент-сървър модел)
- Трябва да бъде осигурена стабилна работа

2.2 Избор на език за програмиране

Програмните езици в последно време се развиват все повече. Освен това, се появяват нови такива с пионерски решения на вече съществуващи проблеми.

Изключваме езиците:

- изискващи допълнителни ресурси, като процесорно време и памет
- новите технологии, които все още нямат широка достъпност, поддръжка и не са развити до етап, годен за ползване

Така, основните езици за програмиране, които са използвани при реализация на системи, в частност СУБД, остават С и С++ [3, 4].

Добре реализирана система с езиците С и С++ може да бъде изключително ефикасна и бърза, а при добро менажиране на паметта - се постига оптимална употреба на компютърните ресурси.

С++ е език за програмиране с общо предназначение, който е разработен като подобрение на езика С, за да включва обектно-ориентирано програмиране. Това е една от най-важните характеристики на С++ спрямо С.

Някой от концепциите, които поддържа един обектно-ориентиран език:

- *Класове* - Това е дефиниран от потребителя тип данни, който съдържа свои собствени членове на данни и функции-членове, които могат да бъдат достъпни и използвани чрез създаване на екземпляр на този клас.
- *Обекти* - абстрактен тип данни, създаден от програмиста, който може да включва множество свойства и методи
- *Енкапсуляция* - В нормални условия енкапсулирането се дефинира като обгръщане на данни и информация в една единица. В обектно ориентираното програмиране, капсулирането се дефинира като свързване на данните и функциите, които ги манипулират.
- *Полиморфизъм* - Думата полиморфизъм означава да има много форми. С прости думи, можем да бъде дефиниран като способността на съобщението да бъде показано в повече от една форма. Реален пример за полиморфизъм е например един човек, който в същото време може да има различни характеристики. Както един мъж може в същото време да е баща, съпруг, служител. Така - един и същ човек има различно поведение в различни ситуации. Това се нарича полиморфизъм. Полиморфизмът се счита за една от важните характеристики на обектно-ориентираното програмиране.
- *Наследяване* - Способността на даден клас да извлича свойства и характеристики от друг клас се нарича наследяване. Наследяването е една от най-важните характеристики на обектно-ориентираното програмиране.

- *Абстракция* - Абстракцията на данни е една от най-съществените и важни характеристики на обектно ориентираното програмиране в C++. Абстракцията означава показване само на съществена информация и скриване на детайлите. Абстракцията на данни се отнася до предоставяне само на съществена информация за данните на външния свят, скриване на фоновите подробности или реализация.

Други особености на C++ са:

- *Независим от средата на компилация (crossplatform)* - Изпълним файл на C++ не е независим от платформата (компилираните програми на Linux няма да се изпълняват в Windows), но са независими от средата на компилация, тоест даден код, който може да работи на Linux/Windows/Mac OSx, което прави C++ средата на компилация независима, но изпълнимият файл (създаден при компилация) на C++ не може да работи на различни операционни системи.
- *Език от високо ниво* - C++ е език от високо ниво, за разлика от C, който е език за програмиране на средно ниво.
- *Базиран на компилатор* - C++ е език, базиран на компилатор, за разлика от други програмни езици като Python и Java. Това представлява, че C++ програмите трябва да бъдат компилирани и техният изпълним файл се използва за стартиране. Поради това че е компилируем, C++ е относително по-бърз език от Java и Python.
- *Управление на паметта* - C++ ни позволява да разпределим паметта на променлива или масив по време на изпълнение. Това е известно като динамично разпределение на паметта. В други езици за програмиране, като Java и Python, компилаторът автоматично управлява паметта, разпределена за променливите. Това не е така в C++, където паметта трябва да бъде разпределена ръчно и след това освободена, когато не е от полза. Разпределението и освобождаването на паметта са важни свойства на езика C++.

- *Многозадачност*- функция, която позволява на дадена система да изпълнява две или повече програми едновременно. Като цяло има два вида многозадачност: базирана на процеси и базирана на нишки.
 - Многозадачността, базирана на процеси, се справя с едновременното изпълнение на програми.
 - Многозадачността, базирана на нишки, се занимава с мултипрограмирането на части от еквивалентна програма. Многонишкова програма съдържа две или повече части, които ще се изпълняват едновременно. Всяка част от такава програма се нарича нишка и всяка нишка дефинира отделен път на изпълнение. Важно е да се отбележи, че C++ не съдържа вградена поддръжка за многонишкови приложения. Вместо това той разчита изцяло на операционната система, за да предостави тази функция.

Избран език за програмиране е C++. Неговите нови стандарти (17 и 20) позволяват програмиране от високо ниво, имплементирани са много подобрения и самият език се поддържа активно.

2.3 Използвани библиотеки

При реализацията на проекта са използвани някой бесплатни библиотеки с отворен достъп:

- catch2 [10] (версия - 2.13.7) - за тестване на C++ код. Основното предимство на catch2 е, че използването му е едновременно лесно и естествено. Тестовете се регистрират автоматично и не е необходимо да бъдат именувани с валидни идентификатори, така че твърденията изглеждат като нормален код на C++.
- spdlog [11] (версия - 1.9.2) - Много бърза, header-only/compiled, C++ библиотека за логове.
- fmt [12] (версия - 8.0.1) - библиотека за форматиране с отворен код, предоставяща бърза и безопасна алтернатива на C stdio и C++ iostreams.
 - Използвана е и при форматирането на логовете в spdlog.

- andreasbuhr-cppcoco [13] (версия - cci.20210113) - предоставя множество примитиви за работа със C++ нишки, coroutines
- benchmark [14] (версия - 1.6.0) - библиотека за сравнителен анализ на бързодействието на кодови фрагменти или функционалности
- cprestsdk [15] (версия - 2.10.18) - проект на Microsoft за облачна комуникация клиент-сървър, използвайки модерен асинхронен C++ API дизайн.

2.4 Менажиране на различните библиотеки/ зависимости

Използването на множество библиотеки/софтуер с отворен код води до проблем, състоящ се в това че управлението и тяхна конфигурируемост стават в известна степен сложни и трудни за поддръжка.

При нормални обстоятелства използването на външен софтуер с отворен код, в езика C++, би следвало да се добавя като всеки модул или библиотека се изтегля локално и след това се извършва връзка между него и проекта. За справянето с този проблем е разработен продукта Conan.

Conan е crossplatform мениджър на зависимости и пакети за езиците C и C++.

Има изключително много предимства:

- Той е безплатен и с отворен код, работи във всички платформи (Windows, Linux, OSX, FreeBSD, Solaris и т.н.)
- Може да се използва за разработка за всички цели, включително embedded, мобилни (iOS, Android) и други. Той, също така,
- Интегрира се с всички build системи като CMake, Visual Studio (MSBuild), Makefiles, SCons и др., включително собствени.
- Специално е проектиран и оптимизиран за ускоряване на разработването и непрекъсната интеграция на C и C++ проекти.
- Позволява пълното управление на двоични файлове, може да създава и използва повторно произволен брой различни двоични файлове (за различни конфигурации като архитектури, версии на компилатора и т.н.),

за произволен брой различни версии на пакети, като използва точно същия процес във всички платформи.

- Напълно автоматизира управлението на зависимости
 - преходни зависимости
 - откриването на конфликти
 - отмяната на зависимости
 - условни зависимости
- Понеже е децентрализиран, е лесно да се стартира собствен сървър, за да се хостват собствени пакети и двоични файлове, без да е необходимо да се споделят.
- Conan е бесплатен софтуер с отворен код. Може да се използва, модифицира, разпространява и разширява - дори за търговски цели.

Всяка зависимост, която е отворена за ползване и е предоставена от Conan може да се намери в така нареченият ConanCenter. Това е централното хранилище, където може да се търсят и откриват всички налични пакети с отворен код, създадени от Conan общността. Той включва информация за конфигурация и улеснява откриването на метаданните на текущия пакет в потребителския интерфейс

2.5 Организация на работата, CI

CI [9] (Continuous Integration) или непрекъснатата интеграция е практика за разработка, която се изразява в това че разработчиците трябва да интегрират код в споделено хранилище няколко пъти на ден. Всяка промяна се проверява чрез automated build, което позволява на екипите да откриват проблеми рано. Чрез такава автоматизация може бързо и по-лесно да се следят и откриват грешки.

За споделено хранилище е избран github [26], като е направен конфигурационен файл за CI, който github предоставя.

Automated builds се случват след команда “git push” само към главен клон (branch) или след сливане на друг клон в главния (merge на PR - pull request).

CI build представлява изграждане на приложението от автоматизиран процес в специален сървър. Изпълняват се набор от тестове, за да се потвърди, че най-новият код се интегрира с това, което в момента е в главния клон.

2.6 Структура на системата

Проектът се състои от няколко основни части/модула. Всеки един от тях е от съществено значение за постигането на крайната цел, като комуникацията между отделните модули трябва да се случва безпрепятствено.

Отделните части се състоят от:

- Компресия: Предоставя възможност за по-оптимизирано съхранение на данните
- Pager: Управлява физическото пространство, на което се съхраняват данните. Предоставя абстракция за извършване на I/O операции, на базата на страници - точно оразмерено количество байтове, върху които се извършват операциите на В-дървото. Към Pager-а влиза и кешът за страници, който позволява по-ефикасна работа с наличните страници, оптимизрайки достъпът до твърдия диск.
- Б-дърво: Основната структура, която се грижи за ефикасното съхраняване на данни [7, 10]. Предоставя оформление, гарантиращо операции за извлечане, вмъкване и изтриване на данни (CRUD операции) с висока производителност. Дърводидната структура се основава на примитивите предоставени на pager-а.
- Сървър: Сървърът е подсистемата на СУБД, която инстанцира и управлява едно или повече В-дървета. Ако базата данни е релационна, там се извършват релациите между различните таблици (В-дървета) и се връщат като резултат на клиента.
- Клиент: Клиентът на СУБД, това е част от системата, която комуникира със сървъра. Клиентът изпраща заявки и получава отговори, съдържащи поисканата информация или резултат от операцията, зададена за изпълнение.

Всеки един от горепосочените модули е взаимосвързан и осигурява гладката работа на цялостната архитектура.

2.6.1 Компресия

Компресията или компресирането на данни е процес, при който информацията от един файл се преобразува така, че да заема по-малко място, тоест размерът на файла да е по-малък. За осъществяването на този процес, се прилагат различни математически методи и алгоритми.

В повечето случаи компресирането всъщност е архивиране. Архивирането е вид компресиране, но без загуба на данни. При него даден файл се добавя в „компресирана папка“. Тази „папка“, въпреки че се нарича така, всъщност не е директория, а отделен файл със специално разширение, което обикновено показва какъв е видът на компресиращият алгоритъм, който е бил използван.

Компресирането е изключително полезно, защото позволява да се намалят ресурсите, използвани от потребителите за съхранението/складирането на дадена информация.

Компресията върви ръка за ръка с декомпресията. Компресираната информация трябва да бъде декомпресирана, за да бъде използвана. Всеки компресиращ алгоритъм има специфичен за него декомпресиращ. Един компресиран файл, за да бъде използван трябва да бъде декомпресиран. Тази допълнителна стъпка е процес, който изисква допълнително време. Последното е пропорционално на големината на файла и зависи пряко от ефикасността на алгоритъма.

При създаването на схеми за компресиране на данни се налага да се правят компромиси, включващи степента на компресия и изчислителните ресурси, необходими, за да се изпълнят нужните операции. Трябва да се намери точната закономерност между бързодействие, ниво на компресираност, сложност и приложимост.

Ползите са както следва:

- Понеже файлът заема по-малко място, той може да бъде споделян по-лесно и по-бързо => по-ефикасен пренос на данни.
- Намалява се заетото пространство, и съответно компютъра може да запише далеч повече информация, отколкото, ако ненужните файлове не са архивирани.
- Препоръчително е важните файлове да се архивират, защото много рядко архивираните файлове биват засегнати от вируси.
- Сигурност, тъй като много трудно се правят модификации в компресиран обект

Компресирането може да бъде със или без загуба. Първият вид (компресиране без загуба), компресира данните без загуби на информация и при декомпресия цялата информация е възстановена и непокътната. Обикновено такива компресиращи алгоритми са базирани на представянето на данните в сбит вариант. Това е възможно, тъй като повечето данни от реалния живот са повторяеми или имат статистическо излишество. Обикновено основен принцип е да се заместват или сбият еднаквите блокове с уеднаквени битове.

При компресирането със загуба се идентифицира ненужната информация и се премахва. По този начин се намалява броят на битовете, но за сметка на качеството. Такива алгоритми обикновено намират приложение при изображения, видео, музика... Схемите, на които са базирани тези алгоритми са базирани и придобити спрямо „недостатъците“ на човешките възприятия. Изследвано е как хората възприемат данните, които са под въпрос. Например човешкото око

Разлики и случаи на използване:

- Алгоритмите за компресиране със загуби не запазват всички данни, но могат да постигнат по-малки крайни файлове.
- Алгоритмите за компресиране без загуби са идеални за системни файлове, където загубата на данни е неприемлива.

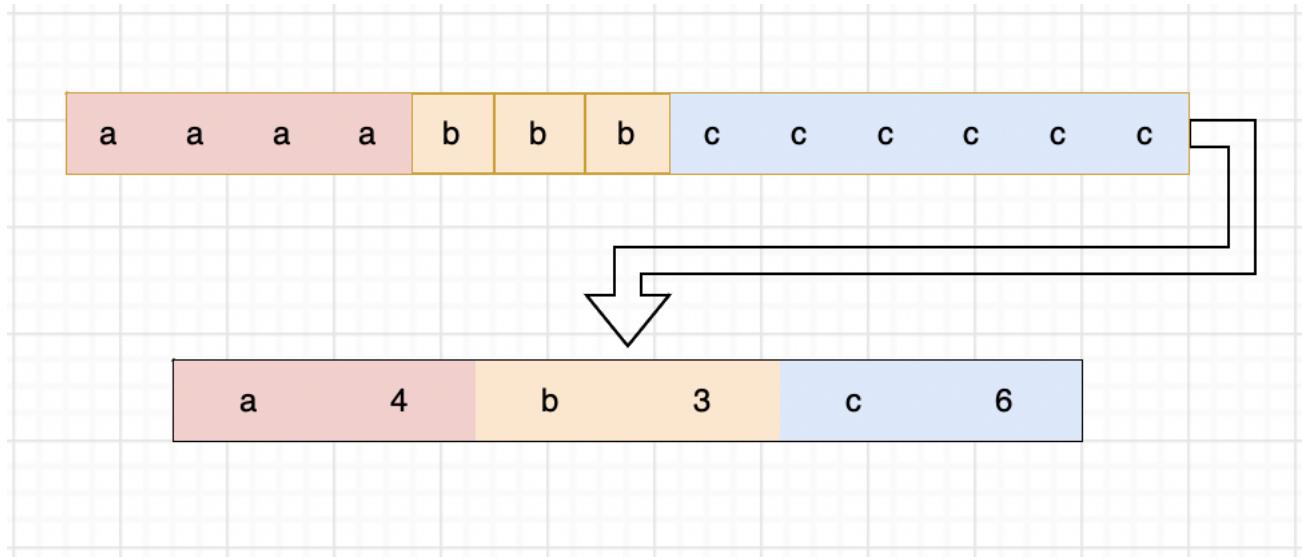
В нашият случай се използва алгоритъм за компресиране без загуба. Алгоритмите за компресиране без загуба обикновено използват методи за премахване на статистическото повторение, за да представят данните в по-сбит вариант, без да се губи информация. Компресирането без загуба е възможно, тъй като в повечето данни от реалния живот има статистическо излишество. Например в едно изображение може да има области, в които цветът не се променя върху няколко пиксела. Вместо да кодираме „червен пиксел, червен пиксел, ...“, информацията може да се представи като „279 червени пиксела“. Това е базов пример за това, как работи кодирането спрямо дълчината (*run-length encoding*), има много схеми, по които може да се намали размер на файл, като се елиминира излишеството в информацията.

Разглеждани видове алгоритми за компресиране без загуба:

- Кодиране по дължина (*run-length encoding*) (2.6.1.1)
- Речниково кодиране (2.6.1.2)
- Кодиране с частично съвпадение (*Prediction by Partial Matching - PPM*) (2.6.1.3)
- Ентропологично кодиране (2.6.1.4)
- Аритметично кодиране (2.6.1.5)
- Кодировка на Хъфман (*Huffman coding*) (2.6.1.6)

2.6.1.1 Кодиране по дължина

Кодирането по дължина (*run-length encoding*) е форма на компресиране на информация, използвана при последователности от данни, където едни и същи елементи се повтарят многократно. След края на кодировката информация е запазена в единична клетка, съдържаща вида данни и броя на неговото срещане. Кодирането по дължина е най-оптимално при файлове, където се срещат много повторения на данни, например простите графични изображения или икони.



Фиг. 2.1 Нагледно изобразяване на алгоритъма за кодиране по дължина

За да намери приложение, алгоритъма за кодиране по дължина, в нашата система за съхранение – то той трябва да „компресира“ стрингове. В един идеален случай стринга би съдържал последователно повтарящи се елементи, както е показано на фиг. 2.1

Алгоритъмът представя оригиналният запис от 13 символа в запис с 6. Последните се имплементират по следния начин: на първо място е символът, а след него броя повторения. Особеностите на този метод са:

- Лесна имплементация
- Кодира данните без загуба
- Неприложим за обикновен текст, понеже много рядко има съответни последователности от символи. Най-често се използва при простите графични изображения и икони.

2.6.1.2 Речниково кодиране

Алгоритмите за компресиране, базирани на речник, поддържат група низове от входния поток, докато се изпълнява процесът на кодиране. Тази група низове се нарича речник и може да се използва за съкращаване на повтарящи се модели в текста. Ако алгоритъмът забележи низ във входния поток, който вече е видял и е съхранил като част от речника, низът може да бъде представен по по-ефективен

начин. Обикновено речниковите записи се обозначават с двубайтов код, който сочи към неговия номер на вписване в речника и неговия размер. По време на процеса на декомпресия рутината създава и поддържа речник със същите правила, които са били използвани в процеса на компресиране. Така че записът n в речника е един и същ по време на компресиране и декомпресия.

Алгоритъмът съхранява всяка уникална дума в паметта и асоциира всеки запис с число, което е позицията на тази дума в речника.. Това елиминира съхранението на дублиращи се стойности в текст, намалявайки общата памет и дисково пространство, необходими за съхранение на данните.

Речниковото кодиране е най-ефективно за информация с “ниска мощност” - колкото по-малък е броят на уникалните стойности, толкова е по-голямо намаляването на използваната памет. Съответно разкодирането на отделни фрагменти или цялостното разкодиране ще бъде по-бързо.

Някои речникови кодирания използват „статичен речник“, при който пълният набор от низове е определен преди да започне кодирането и не се променя в процеса на кодиране. Този подход е най-често използван, когато съобщението или наборът от съобщения, които трябва да бъдат енкодирани, е голям и устойчив

Често използвани методи са такива, при които речникът започва при някое предопределено състояние, но съдържанието се променя в продължение на процеса на енкодиране, основано на данните, които вече са били енкодирани. На този принцип работят алгоритмите LZ77 и LZ78 [17], където структурата от данни се нарича “пълзгащ прозорец” (sliding window). Съвременна имплементация на този алгоритъм е GraphChi [18].

2.6.1.3 Кодиране с частично съвпадение

Кодирането с частично съвпадение (prediction by partial matching) е адаптивна техника за компресиране на статистически данни, базирана на контекстно

моделиране и прогнозиране. PPM моделите използват набор от предишни символи в некомпресирания символен поток, за да предскажат следващия символ в потока. PPM алгоритмите могат също да се използват за кълстериране на данни в предвидени групировки в кълстерния анализ.

Прогнозите обикновено се свеждат до класиране на символи. Всеки символ (буква, бит или друго количество данни) се класира преди да бъде компресиран и системата за класиране определя съответната кодова дума (и следователно степента на компресия). В много алгоритми за компресиране, класирането е еквивалентно на оценката на функцията на вероятностната маса. Като се имат предвид предишните букви (или даден контекст), всеки символ се присвоява с вероятност. Например, при аритметичното кодиране (2.6.1.5) символите се подреждат по техните вероятности да се появят след предишни символи и цялата последователност се компресира в една фракция, която се изчислява според тези вероятности.

Броят на предишните символи, n , определя реда на PPM модела, който се обозначава като $PPM(n)$. Неограничени варианти, при които контекстът няма ограничения за дължина, също съществуват и се обозначават като PPM^* . Ако не може да се направи прогноза въз основа на всички n контекстни символа, се прави опит за прогнозиране с $n - 1$ символа. Този процес се повтаря, докато се намери съвпадение или в контекста не останат повече символи. В този момент се прави фиксирана прогноза.

Имплементациите на PPM компресия се различават значително. Действителният избор на символ обикновено се записва с помошта на аритметично кодиране (2.6.1.5), въпреки че е възможно също да се използва кодиране на Хъфман (2.6.1.6) или дори някакъв вид техника за речниково кодиране. Основният модел, използван в повечето PPM алгоритми, също може да бъде разширен, за да се предвидят множество символи. Размерът на символа обикновено е статичен, обикновено един байт, което улеснява общата работа с всеки файлов формат.

Голям недостатък на РРМ алгоритмите е че изискват значително количество RAM памет, поради което не е избран в текущата разработка.

2.6.1.4 Ентропологично кодиране [19]

Един от основните типове на ентропологично кодиране създава и определя уникален префиксен код към всеки уникален символ. Тези ентропологични кодировки след това компресират данните чрез заместването на всеки символ с неговия уникален префиксен код. Дължината на всяка дума от кода е приблизително пропорционална на негативния логаритъм от вероятността. Следователно най-честите символи използват най-късите кодове.

2.6.1.5 Аритметично кодиране [20]

Аритметичното кодиране е метод за компресиране на данни без загуба. То е форма на ентропологичното кодиране. За разлика от ентропологичното кодиране, което разделя входните данни на отделни символи и заменя всеки символ с даден код, аритметичното кодиране кодира цялото входно съобщение и го превръща на едно число между 0.0 и 1.0.

2.6.1.6 Кодировка на Хъфман (Huffman coding)

Кодировката на Хъфман [7] е техника за компресиране на данни, за да се намали размерът им, като представлява алгоритъм за ентропологично кодиране. За първи път е разработен от Дейвид Хъфман.

Алгоритъмът използва таблица с кодове, притежаващи променлива дължина, използвани за кодирането на сорс символи (например писмен знак от текстов файл). Тази таблица произлиза по начин, базиран на оценената вероятност на проявление на всяка възможна стойност на сорс символа.

За реализация е избрана тази компресия, поради следните причини:

- доказана ефикасност

- средно ниво на трудност при реализация
- лесни и логични стъпки при компресия
- бързодействие
- малък спектър от потенциални проблеми

2.6.2 Pager

Основната роля на този модул е да управлява пространството, върху което се разполага базата данни. Той предоставя примитиви за достъп, които вътрешно осигуряват максимална ефикасност на изпълняваните I/O операции.

Това е подсистемата на СУБД, която управлява физическото пространство, на което се съхраняват данните. Предоставя абстракция за извършване на I/O операции, на базата на страници - точно оразмерено количество байтове, върху които се извършват операциите на В-дървото. Към Pager-а влиза и кешът за страници, който позволява по-ефикасна работа с наличните страници, оптимизирайки достъпът до твърдия диск.

2.6.3 В-дърво

В-дървото е специален тип самобалансиращо се дърво за търсене, в което всеки възел може да съдържа повече от един ключ и може да има повече от две деца. Това е обобщена форма на дървото за двоично търсене.

Известно е още като “m-way” дърво с балансирана височина.

Необходимостта от В-дърво възниква с нарастването на необходимостта от по-малко време за достъп до физически носител за съхранение като твърд диск. Вторичните устройства за съхранение са по-бавни с по-голям капацитет. Има нужда от такива типове структури от данни, които минимизират достъпа до диска.

Други структури от данни като двоично дърво за търсене, “avl” дърво, червено-черно дърво и т.н. могат да съхраняват само един ключ в един възел.

Ако трябва да съхранявате голям брой ключове, тогава височината на такива дървета става много голяма и времето за достъп се увеличава.

Въпреки това, В-дървото може да съхранява много ключове в един възел и може да има множество дъщерни възли. Това значително намалява височината, позволявайки по-бърз достъп до диска.

2.6.4 Комуникация, сървърна архитектура

Сървърната архитектура е от съществено значение за реализацията на системата, тъй като е нужно някакво взаимодействие между обикновените потребители и предоставеният API.

Сървърът е подсистема на СУБД, която позволява асинхронни CRUD операции от страна на даден потребител. След това се грижи за ефикасното им предаване към Б-дървото или Б-дърветата, където се извършва съхранението на информацията, предоставена от клиента.

Трета Глава

Реализация на СУБД

Няма еднозначна методология за избиране и създаване на архитектурите на различните системи. При реализацията на СУБД търсеният метод трябва да е максимално ефикасен и стабилен, възможно най-достъпен и скалируем, придържащ се към основните, доказани във времето принципи, използвани при създаване на база данни.

Последователността от действия в текущата архитектура е както следва:

1. Поставяне на конфигурация за текущия база данни
2. Сървърен модул
 - 2.1. Автентикация чрез POST заявка
 - 2.2. Запис/редакция на данни чрез PUT заявки
 - 2.3. Извличане на данни чрез GET заявки
3. В-дърво
 - 3.1. Едностъпкови операции върху съхраняваните данни
 - 3.1.1. Създаване на записи (наредена двойка „ключ-стойност“)
 - 3.1.2. Извличане на стойности на база подадени ключове
 - 3.1.3. Премахване на стойности, асоциирани с подадени ключове
 - 3.2. Многостъпкови операции
 - 3.2.1. Групов запис
 - 3.2.2. Филтриране
 - 3.2.3. Групово премахване
 - 3.3. Операции свързани със постоянно запазване на данните
 - 3.3.1. Зареждане
 - 3.4. Запазване
4. Pager
 - 4.1. Управление на паметта под формата на страници
 - 4.2. Оптимизиране на I/O операциите

5. Компресиращ/Декомпресиращ модул
 - 5.1. Компресиране на данните под формата на файл
 - 5.1.1. Може да бъде съвкупност от файлове и директории (дърводидна структура)
 - 5.2. Декомпресиране на данните
 - 5.2.1. Ако компресираният файл е съвкупност от файлове и директории е възможно частично декомпресиране (само на отделни файлове)

3.1 Сървърен модул

Клиент - сървър комуникацията е реализирана чрез `cpprestsdk` [15]. Цели се възможно най-голяма абстракция, така че потребителя на текущото API да може да постигне висока степен на конфигурируемост.

Предоставени са логики за:

- Авторизация на потребител чрез “POST” заявка
- Запис или актуализация на информацията към В-дървото чрез “PUT” заявка
- Изтриване на информация чрез “DELETE” заявка
- Извличане на информация чрез “GET” заявка

3.1.1 Авторизация

Потребител може да бъде регистриран чрез “basic authentication” принципа, а това се случва чрез POST заявка на адрес “/eugene/register”.

Нужно е данните за потребителя да бъдат изпратени във формат “base64” стринг, за да могат да бъдат защитени и декодирани правилно. Клиентът трябва да изпраща HTTP заявки с “header” за авторизация, която съдържа търсените име и парола.

След като даден потребител е регистриран, неговите данни се съхраняват в В-дървото под формата на ключ и стойност стрингове.

Всяка останала заявка проверява дали вече има регистриран потребител със съответното потребителско име и парола - ако има, заявката се изпълнява, а ако ли не - връща грешка за неоторизиран потребител (статус код 401) и не може да се извърши зададената операция (фигура 3.1).

```
if (!authenticate( headers: request.headers())) {  
    request.reply( status: status_codes::Unauthorized);  
    return;  
}
```

Фиг. 3.1 Отговор при неуспешна оторизация

3.1.2 Запис на информация

С “PUT” заявка на адрес “/eugene” може да бъде извършен запис на информация в В-дървото.

Потребителя може сам да си настрои какво иска да записва като данни с конфигурационния файл на В-дървото. Така могат да бъдат реализирани релационни и нерелационни бази данни, които да съдържат специфични за клиента данни.

При пръв запис на информация се създава ново В-дърво, което може да бъде използвано с един конфигурационен файл, като за да бъдат направени много В-дървета трябва да се направят нужните за инстанциране конфигурационни файлове. Когато има опит за актуализация на вече съществуваща информация се случват следните стъпки:

- Обработване на “PUT” заявката
 - Проверка за правилни потребителско име и парола
 - Проверка за правилен адрес

- Ако се обърка нещо друго се връща статус код “Not found” (цифрен код - 404)
- Индексиране на В-дървото
- Балансиране на В-дървото
- Обновяване на компресираните файлове

При успешна операция - всяка заявка връща статус код “OK” (цифрен код - 200), както фигура 3.2.

```
108
109
request.reply( status: status_codes::OK, body_data: answer);
return;
```

Фиг. 3.2 Отговор при успешна заявка

3.1.3 Изтриване на информация

Изтриването на информация става чрез “DELETE” заявка на адрес “/eugene”. То има същите спецификации като при “PUT” заявката и с последната може да се извърши същото, което може да се извърши с текущата заявка.

“DELETE” заявката е имплементирана за по-ясно възприеман интерфейс от страна на потребителя, като при триене на информация от В-дървото се случват стъпките описани в точка [3.2.2].

3.1.4 Извличане на информация

Извличането на информация от потребителя може да се случи на адрес “/eugene” чрез “GET” заявка. Последователността от действия тук е:

- Обработване на “GET” заявката
 - Проверка за правилни потребителско име и парола
 - Проверка за правилен адрес
 - Ако се обърка нещо друго се връща статус код “Not found” (цифрен код - 404)
- Индексиране на В-дървото

- Извличане на търсените данни от В-дървото
- Връщане на търсените данни от потребителя под формата на “json” обект със статус код “OK” (цифрен код - 200)

3.2 В-дърво

В-дървото е вид идеално балансирано дърво, което съхранява данни в подреден ред и позволява операции за извлечане, вмъкване и премахване на записи. Всеки възел от дървото съдържа в себе си част от цялостните данни, които биват подредени според сравняваща функция.

Според оригиналната си дефиниция [23] всеки възел на В-дърво съдържа множество записи (ключ и стойност), както и връзки към наследниците на възела. Максималният обем на всеки от тях е статично определен при дефиницията на дървото и не се изменя в хода на изпълнението. Основните операции, които се прилагат върху дървото се изпълняват с гарантирана логаритмична сложност - $\log(n)$, където n е текущия брой на възли в структурата.

Съществуват различни модификации на В-дървото като структура от данни. През годините вариацията B+ се е доказала като фундаментална в основната на реализацията на СУБД [24]. За разлика от дефиницията на [25], Тази структура е конструирана от два вида възли - вътрешни (разклонения) и листа. Записите с данни се съхраняват изцяло в листата, докато вътрешните възли служат изцяло за насочване на търсенето на операцията по извлечане.

Структурата се базира на алгоритми, които се изпълняват абстрактно от типа данни, които се изпълняват. Това позволява записите, върху които се оперира да бъдат от различни типове. Най-често това са наредени двойки „ключ-стойност”, където „ключът” е уникален идентификатор на съхраняваните данни, които биват представени като „стойност”.

3.2.1 Свойства на В-дървото

1. За всеки възел х ключовете се съхраняват в нарастващ ред.
2. Ако “n” е реда на дървото, всеки вътрешен възел може да съдържа най-много “n - 1” ключа заедно с указател към всяко дете.
3. Всеки възел с изключение на “root” може да има най-много “n” деца и най-малко “n/2” деца.
4. Всички листа имат еднаква дълбочина (т.е. височина-h на дървото).
5. Коренът има поне 2 деца и съдържа минимум 1 ключ.
6. Ако “ $n \geq 1$ ”, тогава за всяко В-дърво с “n” ключа и височина “h” има минимална степен “ $t \geq 2$ ”, “ $h \geq \log t (n+1)/2$ ”.

3.2.2 Операции с В-дървото

Търсенето на елемент в В-дърво е обобщената форма на търсене на елемент в двоично дърво за търсене. Следват се следните стъпки:

1. Корена на дървото се означава като текущ възел.
2. Ако текущия възел е листо, се сканират ключовете във възела, търсейки посочения. Ако той бъде намерен, се връщат асоциираните данни, в противен случай търсенето приключва без да се намерят желания ключ.
3. Ако текущия възел е разклоняващ, се търси най-малката стойност, която надвишава търсения ключ. Ако такава не съществува, операцията приключва. В противен случай за текущ възел се означава възелът, към който сочи намереното разклонение.
4. Повтаря се стъпка 2.
5. Повтарят се стъпки от 1 до 4, докато се стигне до листото.

3.4 Компресионен модул

Компресирайт модул е базиран на алгоритъма за кодиране на Хъфман. Предоставя възможност за по-оптимизирано и сигурно съхранение на потребителските данни.

Компресията може да бъде разделена на две основни части.

Първата част обработва файла в сувор вид, като извлича и съхранява логически информационни блокове, които ще послужат впоследствие при компресирания нов файл. Състои се от следните стъпки, които са съхранени под формата на променливи:

- Информация за размера на файла
- Отчитане на честотата на използваните уникални байтове и броя на уникалните байтове
- Създаване на основното дърво (trie), което ще послужи по-късно за транслация на уникалните байтове
 - Създаване на `trie_root` (главния корен на транслационната карта) вътре в транслационната карта чрез разпределение на тежестта, определяна от честотата на определен байт
 - Добавяне/създаване на нови възли, които представляват преведените версии (в компресиран формат) на всеки уникален байт

Втората част пише в новосъздадения файл с разширение “`.huff`”. Записва следните параметри, които ще са нужни при декомпресирането, за да се получи завършен алгоритъм без загуба на данни [6]:

1. Броя на всички уникални символи/байтове
2. Бит групи
 - 2.1. 8 бита за текущия уникален символ
 - 2.2. 8 бита за дължината на транслацията
 - 2.3. Динамичен брой битове - за транслационният код на текущия уникален символ
3. 2 байта, които индикират колко е броя на файловете, които ще бъдат компресирани
4. 1 бит - показва дали ще бъде компресиран файл или ще бъде компресирана папка
5. 8 байта, които представляват размера на текущия входен файл
6. Бит групи

- 6.1. 8 бита за дължината на името на текущия/текущата файл/папка
- 6.2. Динамичен брой битове, които представляват трансформираната версия на името на текущия/текущата файл/папка
7. Динамичен брой битове, които представляват трансформираната версия на съдържанието на текущия/текущата файл/папка

3.4.1 Компресионен модул първа част

Целия компресионен модул е разделен на вътрешни модули (namespace), които позволяват скалируемост, възможност за по-оптимално тестване на отделните фрагменти код и една по-добра подялба спрямо обикновения потребител. Последният ще използва “namespace compression”, който предоставя класа “Compressor”.

При създаване на инстанция от класа “Compressor”, потребителят трябва да подаде два аргумента:

1. релативен или абсолютен път до файловете/папките за бъдеща компресия като първи аргумент
2. името, което иска да бъде зададено на бъдещия компресиран файл, като втори аргумент.

За стартиране на вътрешния модул на компресията се извиква името на първоначалната инстанция на “Compressor” класа + оператора “()”, който е предефиниран да изпълнява конструктора на “CompressorInternal” класа.

```
compression::Compressor compress{
    args: std::vector<std::string>(
        n: 1, value: params["test_dir_name"]),
    compressed_name: params["compressed_name"]
};
compress();
```

Фиг. 3.3 Примерна инициализация от обикновения потребител за класа “Compressor”

Използвана е специална C++ програмна техника с “PImpl” [22], която премахва подробностите за реализациите на клас от неговото обектно представяне, като ги поставя в отделен клас, достъпен чрез указател - фигура 3.4.

```

502 class Compressor {
503 private:
504     using pimpl = storage::detail::CompressorInternal;
505     std::unique_ptr<pimpl> compressor_internal;
506
507 public:
508     /// \brief Constructor of the compression class with which you
509     /// can compress provided m_files
510     ///
511     /// \param args - arguments that are provided to the program
512     /// \param compressed_name - name of the future compressed file
513     explicit Compressor(const std::vector<std::string> &args,
514                          std::string_view compressed_name = "") {...}
515
516     /// \brief The main function of compression class that do all
517     /// the magic with provided m_files.
518     void operator()()
519     {
520         (*compressor_internal)();
521     }
522 };
523 // namespace compression

```

Фиг. 3.4 Класът “Compressor”, предназначен за обикновения потребител

Трябва да се има предвид че всяка една от гореизброените стъпки, в “**3.5 Компресионен модул**” от първа и втора част на алгоритъма, се извършват във вътрешен модул (namespace), наименован “storage::detail”, съдържащ “CompressorInternal” класа (фиг. 3.5).

```

62 namespace compression {
63     namespace storage::detail {
64         class CompressorInternal {...};
65     } // namespace storage::detail
66     class Compressor {...};
67 } // namespace compression

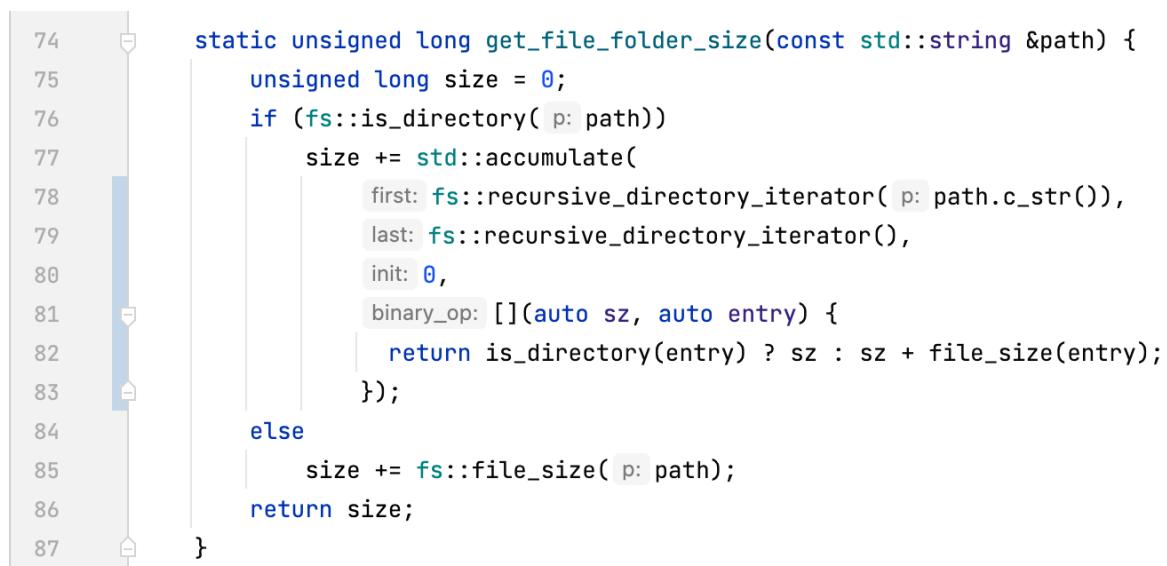
```

Фиг. 3.5 Структура на компресионния модул

3.4.1.1 Информация за размера на файла

Анализирането на размера на файла е интересен процес, поради използването на някой особености на новите стандарти на езика C++.

В случая, както може да се види на фигура 3.6, е използвана функцията “accumulate”, идваща от новата стандартна библиотека, която спомага за интуитивното итериране по файловете, ако е предоставена като входен аргумент папка. Получава се акумулиране на размера на всеки един файл.



```
74     static unsigned long get_file_folder_size(const std::string &path) {
75         unsigned long size = 0;
76         if (fs::is_directory( p: path))
77             size += std::accumulate(
78                 first: fs::recursive_directory_iterator( p: path.c_str()),
79                 last: fs::recursive_directory_iterator(),
80                 init: 0,
81                 binary_op: [](auto sz, auto entry) {
82                     return is_directory(entry) ? sz : sz + file_size(entry);
83                 });
84         else
85             size += fs::file_size( p: path);
86         return size;
87     }
```

Фиг. 3.6 Тяло на функция за анализиране на размера на папка/файл

Важно е, че “fs” е псевдоним за друга стандартна библиотека, а именно “filesystem” (фиг. 3.7).

13

```
namespace fs = std::filesystem;
```

Фиг. 3.7 Използвана е стандартната библиотека “std::filesystem”

3.4.1.2 Отчитане на честотата на използваните уникални байтове и броя на уникалните байтове

Използван е един от най-бързите начини за съхранение и справяне с този проблем. Прегледани са много възможни решения, като “`std::map`”, който да пази ключ-стойност релация (например символ - честота на срещане) или “`std::unordered_map`”, който се води още по-бърз понеже е хеширан, но всяко едно от тези решения е по-бавно от текущото.

```
163     std::array<long int, 256> m_occurrence_symbol;//!< long integer array that
164 //!< will contain the number of occurrences of each symbol in the m_files
```

Фиг. 3.8 Целочисленият масивът, който съдържа честотата на срещане на всеки символ

На фигура 3.8 е показан масив от 256 целочислени стойности. Понеже всеки символ има уникален ASCII код, а символите в ASCII таблицата са 256 - съответно общия брой кодове е 256.

Това позволява индексите на масива да отговарят на уникалния ASCII код на всеки един символ, а целочислените стойности - на това колко често е срещан във съответния/съответната файл/папка.

Реализацията на това може да се види на фигура 3.9:

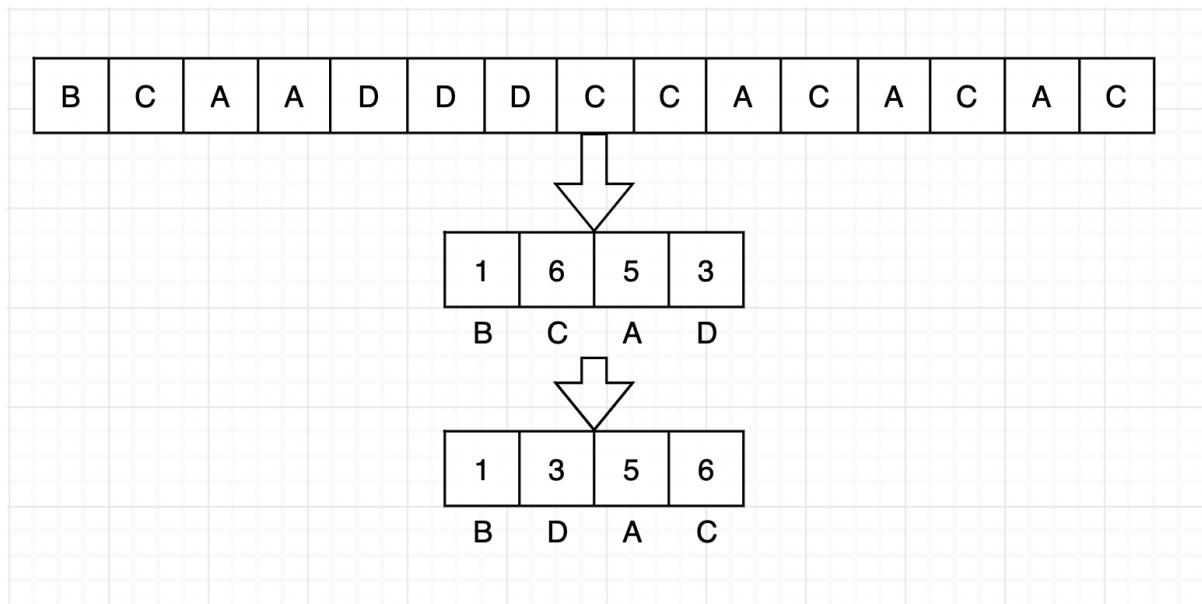
- Обхождат се всички файлове
- Обхожда се съдържанието на текущ файл
- За запис на съответен индекс типа се преобразува към целочислен
- На индекс, където не съответства символ е записана 0

```
107
108
109      for (const auto &item : const string & : m_files) {
          for (const char *c = item.c_str(); *c; c++)
              m_occurrence_symbol[(uint8_t) *c]++;
```

Фиг. 3.9 Запис в масива за уникалните байтове и тяхната честота

3.4.1.3 Създаване на основното дърво, което ще послужи по-късно за транслация на уникалните байтове

Както В-дървото, така и Хъфман алгоритъма за компресия е базиран на дърво (trie структура [21]).

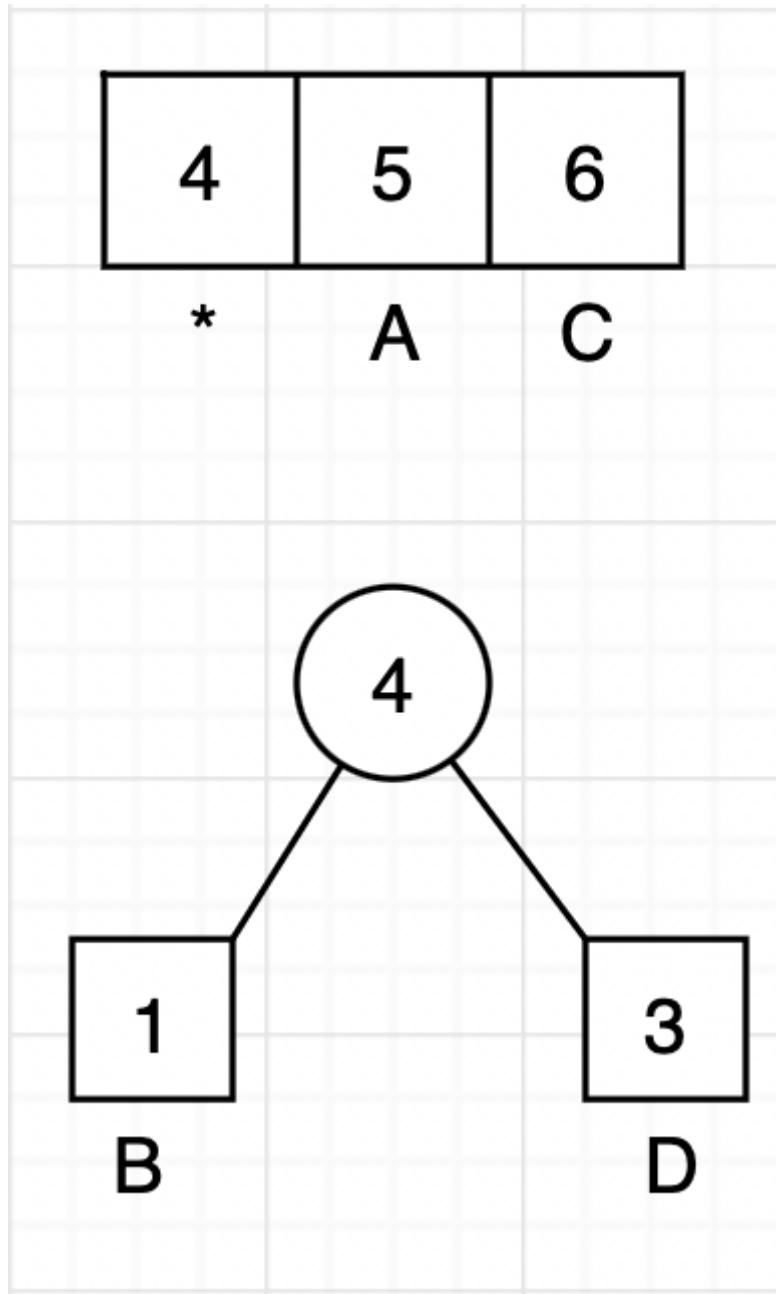


Фиг. 3.10 Визуализация на стъпките при компресия

На фигура 3.10 е представено нагледно текущото положение, постигнато от предишните 2 стъпки на алгоритъма. Тоест ако има даден низ от символи, първо ще се намери честота на всеки символ и след това ще се извърши сортиране във възходящ ред.

Следващите стъпки целят създаването на Хъфман дървото, което по-късно ще се използва за транслация на отделните символи (фиг. 3.11).

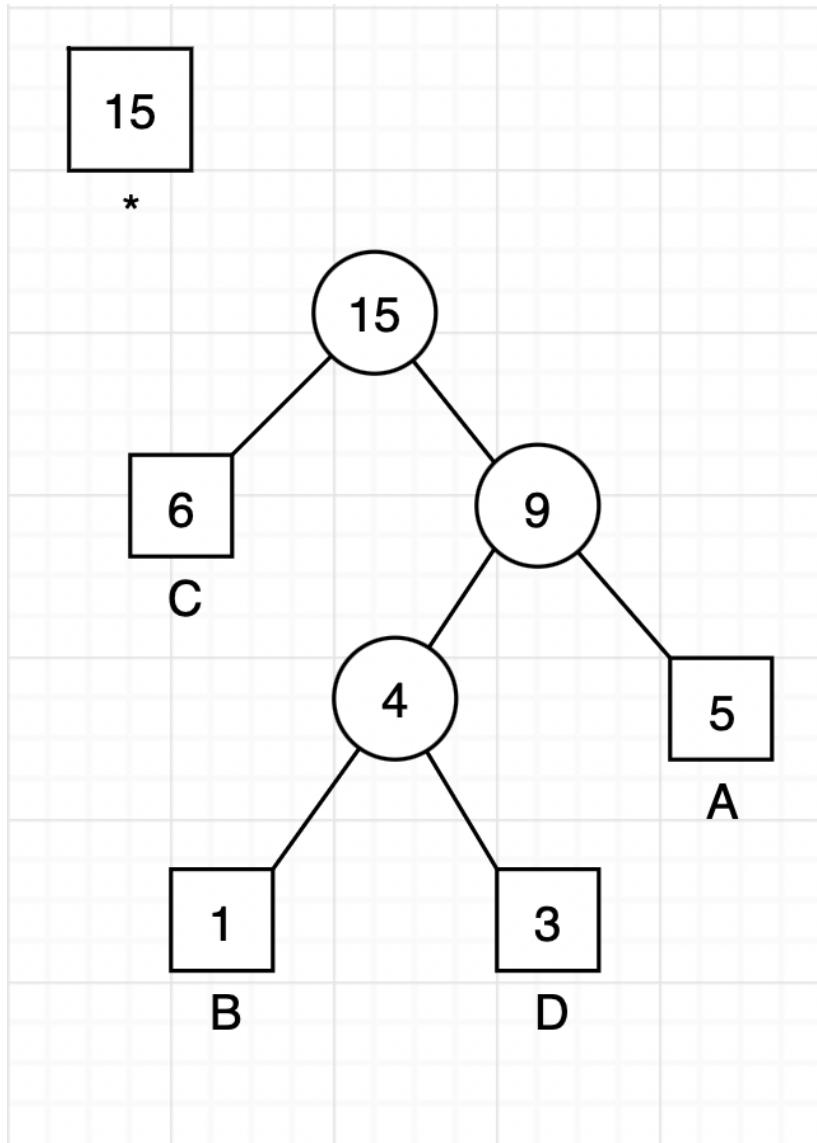
Започва се с празен възел, на който левия дъщерен възел представлява минималната честота на гореизброените символи, а десния - втората минимална честота. Стойността на главния възел трябва да бъде сбор от честотите на дъщерните му възли.



Фиг. 3.11 Първа стъпка при създаване на Хъфман дървото

След това се премахват тези две минимални честоти от списъка с всички честоти и се добавя нова сума в списъка с честоти. Тази операция представлява “*” от фиг. 3.11.

След това се повтарят тези стъпки за всички знаци - може да се види нагледно във фиг. 3.12.



Фиг. 3.12 Финален стадий на Хъфман дървото (наричано още Trie) спрямо примера

Като код тези стъпки се извършват от функцията “initialize_trie” в “CompressorInternal” модула, а на фигура 3.13 се вижда структурата, в която се съхраняват отделните разклонения (възли/листа) и каква информация съдържат те.

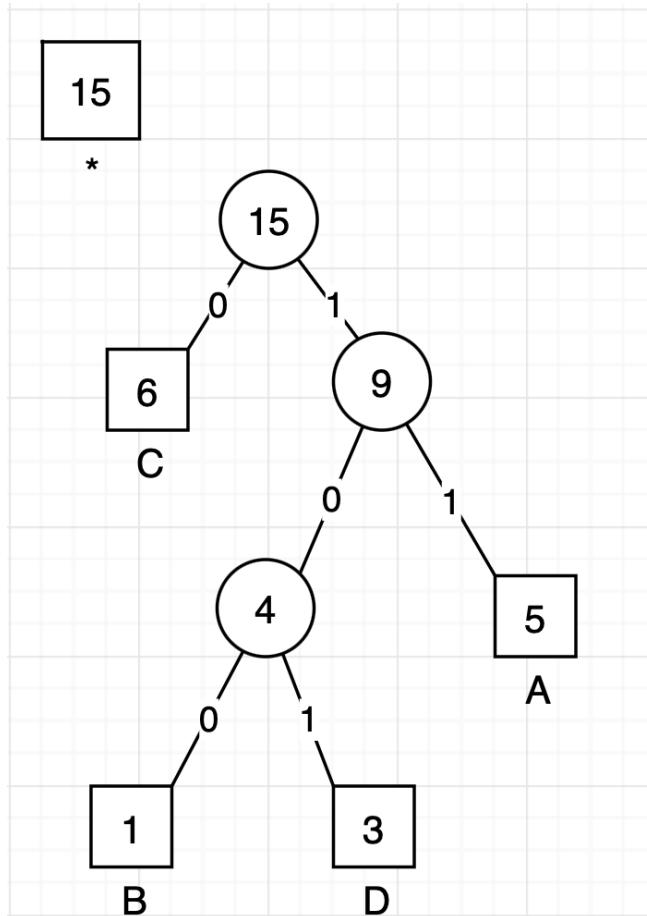
```

149 // brief This structure will be used to create the trie
150 struct huff_trie {
151     huff_trie *left=nullptr, *right=nullptr; //!< left and right nodes of the trie_root
152     uint8_t character; //!< associated character in the trie_root node
153     long int number; //!< occurrences of the respective character
154     std::string bit; //!< bit that represents Huffman code of current character
155
156     huff_trie() = default;
157
158     huff_trie(long num, uint8_t c) : character(c), number(num) {}
159
160     bool operator<(const huff_trie &second) const {
161         return this->number < second.number;
162     }
163 };
164

```

Фиг. 3.13 Структура на Хъфман дървото

Параметъра, наименован “bit”, в тази структура е от тип стринг и представлява хъфман кода на съответния символ. Впоследствие този бит ще бъде записан в компресирания файл под формата на единици и нули, съхранявани в променлива чрез побитови операции.



Фиг. 3.14 Асоциирани 0 и 1, репрезентиращи хъфман код на символ

Последната стъпка при инициализацията на хъфман дървото (“trie” структурата) е да се запишат съответстващите 0 и 1 към всеки възел. За целта на всяко разклонение на ляво (такова, със символ с по-висока честота) трябва да се зададе 0, докато при всяко разклонение на дясно (такова, със символ с по-ниска честота респективно) трябва да се зададе 1. Графично това е изобразено на фигура 3.14, докато като код може да се види при последната операция от функцията “initialize_trie” - фигура 3.15, където всеки един бит от даден възел се акумулира, за да се получи цялостния хъфман код за съответен символ.

```
for (huff_trie *huff = m_trie.data() + m_symbols * 2 - 2; huff > m_trie.data() - 1; huff--) {
    if (huff->left)
        huff->left->bit.insert(huff->left->bit.begin(), huff->bit.begin(), huff->bit.end());
    if (huff->right)
        huff->right->bit.insert(huff->right->bit.begin(), huff->bit.begin(), huff->bit.end());
}
```

Фиг. 3.15 Изграждане на хъфман кодовете

3.4.2 Компресионен модул втора част

Следващата стъпка при текущата реализация на хъфман компресията е създаване на нов файл, в който трябва да бъде съхранена информацията от гореизброените стъпки (първа част). Тук трябва да се отбележи, че при текущия подход се четат входните файлове два пъти.

При първото преминаване програмата отчита честотата на използване на всеки уникален байт и създава претеглено дърво за превод на всеки използвани уникален байт, обратно пропорционално на неговата честота на използване. След това тази информация за трансформацията се записва в компресирания файл за целите на декомпресията.

При второто преминаване през файла - програмата превежда входните файлове спрямо вече създаденото дърво и записва в новосъздадения компресиран файл.

На този етап цялата нужна информация за създаване и запис в компресирания файл се знае, а данните именно, се състоят от:

- Хъфман дървото (trie) е създадено
- В променливи са записани:
 - Размера на входните файлове
 - Броя на уникалните символи
 - Честота на срещане на всеки символ

Паралелно с преминаването на стъпките от първа част на компресията е създадена променлива, в която, спрямо определени условия, е установена големината на компресирания файл.

Условията за определяне на големината на компресирания файл са тези описани в [3.5] (втора част на компресионния модул).

След като се имат налични всички тези параметри, трябва да бъдат възможно най-оптимално записани, като не се пропуска нищо и се следва строга последователност, за да може по-късно декомпресиращият алгоритъм да възпроизведе данните без загуби.

178	<code>std::array<std::string, 256> m_char_huffbits; //!< transformation string</code>																
179	<code>//!< is put to m_str_huffbits array to make the compression process more time efficient</code>																
<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: center;">0</td> <td style="text-align: center;">null</td> </tr> <tr> <td style="text-align: center;">1</td> <td style="text-align: center;">null</td> </tr> <tr> <td style="text-align: center;">2</td> <td style="text-align: center;">null</td> </tr> <tr> <td style="text-align: center;">...</td> <td style="text-align: center;">...</td> </tr> <tr> <td style="text-align: center;">49</td> <td style="text-align: center;">101</td> </tr> <tr> <td style="text-align: center;">50</td> <td style="text-align: center;">null</td> </tr> <tr> <td style="text-align: center;">...</td> <td style="text-align: center;">...</td> </tr> <tr> <td style="text-align: center;">97</td> <td style="text-align: center;">1</td> </tr> </table>		0	null	1	null	2	null	49	101	50	null	97	1
0	null																
1	null																
2	null																
...	...																
49	101																
50	null																
...	...																
97	1																

Фиг. 3.16 Време ефикасният подход за справяне с релацията символ - хъфман код

На фигура 3.16 е представен един от най-удачните начини за индексиране на информацията - чрез вече споменатия подход с масив и ASCII индекси. Идеята е същата - ASCII кодът на символа служи като индекс в масива и масивът е от 256 елемента, понеже ASCII кодовете са 256. Единствената разлика тук е че се съхраняват 256 уникални стринга, понеже се търси на съответната буква какъв е хъфман кодът. Така може да има символи, които не се срещат във файла/файловете - съответно там е записана празна (null) стойност, а на индекс, който отговаря на реално буква е записан найният хъфман код във вида показан на фиг. 3.16.

```

438     /// \brief Writes provided string bytes to the new compressed file
439     ///
440     /// \param for_write - string that will be written
441     void write_bytes(std::string_view for_write) {
442         for (const auto &item : const char & : for_write) {
443             if (m_current_bit_count == CHAR_BIT) {
444                 fwrite(ptr: &m_current_byte, size: 1, n: 1, s: m_compressed_fp);
445                 m_current_bit_count = 0;
446             }
447             switch (item) {
448                 case '1': m_current_byte <= 1;
449                     m_current_byte |= 1;
450                     m_current_bit_count++;
451                     break;
452                 case '0': m_current_byte <= 1;
453                     m_current_bit_count++;
454                     break;
455
456                 default:
457                     Logger::the().log( lvl: spdlog::level::err,
458                                     msg: "Compressor: Function write_bytes incorrect configuration!");
459             }
460         }
461     }

```

Фиг. 3.17 Функцията, която прави записите във файла

Следващата стъпка е да се обходят всички файлове (ако те са няколко) и последователно да се заменят символите в наименованията им и в съдържанията им със съответния за всеки един символ, предварително генериран, уникален хъфман код.

На фигура 3.17 може да се види как всъщност се записва информация в новосъздадения файл, предназначен за съхранение на компресирани данни.

Така на функцията “write_bytes” се подава константен стринг, съдържащ “1” и/или “0” като символи. Този стринг всъщност е съответният хъфман код, отговарящ за даден символ от съдържанието на файла. Всеки такъв символ трябва да бъде заменен с хъфман код посредством вече създадения масив за бързо индексиране спрямо ASCII кодовете на символите.

За правилното записване на хъфман код стринг в един байт са използвани побитови операции. Символите се обхождат, като чрез побитови операции се репрезентират в един единствен байт. Този байт след това се записва във файла за компресиране.

Това се повтаря многоократно, докато не се изчерпа информацията от файла. След това се продължава със следващ файл, ако има такъв.

```
418     /// \brief This function translates and writes bytes from current input file to the compressed file.
419     /// (Manages seventh of part 2)
420     ///
421     /// \param path - string that represents the path to the file/folder
422     void write_file_content(const std::string &path) {
423         const std::string buff = return_file_info(path);
424
425         for (const auto &item : const char & : buff)
426             write_bytes( for_write: m_char_huffbits[(uint8_t) item]);
427     }
```

Фиг. 3.18 Замяна на символите от съдържанието на файл със съответните им хъфман кодове

Използвана е функцията “write_file_content” (фигура 3.18), която записва съдържанието на всеки файл, като заменя символ с код и в тялото си извиква функция “write_bytes” (фигура 3.17).

Има строга последователност от действия, без които компресията, а още повече след това декомпресията - не биха били възможни. През цялото време трябва да се следи за състоянието на текущия байт, който трябва да бъде записан в

компресирания файл, за броя на записаните битове в байта и размера на текущия код.

3.5 Декомпресионен модул

Декомпресията, за разлика от компресията, прочита информацията от файла само веднъж. Декомпресорът първо чете информация за “превода” - trie структурата, която съдържа данни за това на кой символ, какъв хъфман код отговаря, и създава двоично дърво от нея. След като този процес приключи, дървото се използва, за да преведе останалата част от файла.

Стъпките, които изпълнява са:

1. Първо се записва информация в променлива за това колко на брой са всички символи
2. След това, спрямо компресията (втора част, втора стъпка) трябва да се извлече информация за:
 - 2.1. Текущия уникален байт, която отговаря за даден компресиран символ
 - 2.2. Дълчината на хъфман кода
 - 2.3. Самият хъфман код
3. Ако трябва да се декомпресира директория:
 - 3.1. Извличат се броя на файловете в текущата директория
4. Следва бит, който показва дали се декомпресира папка или файл (“1” за файл и “0” за папка, както при компресията)
5. Отново в променлива се записва оригиналният размер на текущия файл, ако е файл
6. След това алгоритъма продължава със същинската част - запис на истинския файл. За да се случи това, трябва да се обработи следната информация:
 - 6.1. 8 бита, показващи дълчината на наименованието на текущата файл/папка

- 6.2. Динамичен брой битове, които се трансформират, за да се получи името на текущия входен файл или папка
7. Записва се трансформираната версия на съдържанието на текущия входен файл

3.5.1 Структура на потребителския клас “Decompressor”

Логиката при инициализация на декомпресията наподобява тази при компресията. Има един “namespace decompression”, който вътре в себе си съдържа класа “Decompressor”.

```

323 class Decompressor {
324     private:
325         using pimpl = storage::detail::DecompressorInternal;
326         std::unique_ptr<pimpl> decompressor_impl;
327
328     public:
329         /// \brief Constructor of the decompression class with which you can detail provided file
330         ///
331         /// \param path - path to the file for detail
332         explicit Decompressor(std::string_view path) {...}
341
342         /// \brief The main function of decompression class that do all the magic with provided m_files.
343         void operator()(std::string_view folder_name = "") {
344             (*decompressor_impl)(folder_name);
345         }
346     };
347 } // namespace decompression

```

Фиг. 3.19 Класът “Decompressor”

Инициализацията и стартирането на декомпресията са много интуитивни процеси. Същите са както при компресията.

При създаване на инстанция от класа “Decompressor”, потребителят трябва да подаде само един аргумент - релативен или абсолютен път до файла за декомпресиране. Може да се види на фигура 3.19.

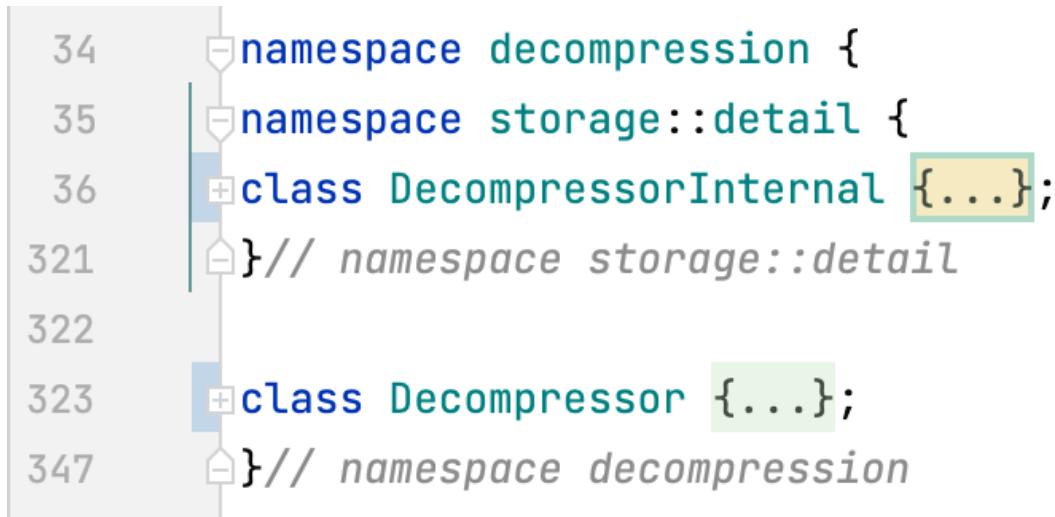
```

decompression::Decompressor decompress{ path: params["compressed_name"]};
decompress( folder_name: partial_decompress_name);

```

Фиг. 3.20 Демонстрация на примерна употреба на декомпресията

Има предефиниран оператор “()”, който стартира процеса по декомпресиране и позволява на вече създадената инстанция да достъпи вътрешния модул (“namespace”) на декомпресията, а именно “namespace storage::detail” - фигури 3.20 и 3.21.



Фиг. 3.21 Вътрешна архитектура на декомпресирана модул

3.5.2 Логика на декомпресията

Декомпресиращият алгоритъм е изцяло обвързан с предварително компресираният файл. Всяка една от стъпките, изброени в [3.5], трябва да бъде изпълнена “на обратно”.

Най-важни са записаните битове от втора част на компресионния модул [3.5.2].

Следва се първа стъпка при компресирането и тя е запис на броя на всички уникални символите на подадения файл. Тогава това трябва да се извлече първо като член променлива чрез функцията “fread”, която чете байтове от подаден файл.

```
fread( ptr: &m_symbols, size: 1, n: 1, stream: m_compressed);
```

Създава се нова хъфман структура (“huff_trie”), която трябва да бъде извлечена от компресириания файл. Тя ще послужи за получаване на оригиналния файл

преди компресията. При декомпресията структурата за “trie” дървото няма нужда от други параметри, освен референции към ляв и десен възел, както и асоциираният символ, за който отговарят (фигура 3.22).

```

69     /// \brief This structure will be used to represent the trie
70     struct huff_trie {
71         huff_trie *zero{nullptr};
72         huff_trie *one{nullptr};      //!< zero and one bit representation nodes of the m_trie_root
73         char character{'\0'};//!< associated character in the trie node
74     };

```

Фиг. 3.22 Хъфман структурата в декомпресионния модул

За извлечане на “huff_trie” структурата от компресирания файл се извиква функцията “trie_initialization” (фигура 3.23) толкова пъти, колкото е броя на уникалните символи, които са предварително прочетени и записани в член променливата “m_symbols”. Тази функция прочита един байт от компресирания файл и след това създава референциите “huff_trie” възлите спрямо него.

```

125     /// \brief trie_initialization function reads n successive bits from the compressed file
126     /// and stores it in a leaf of the translation trie,
127     /// after creating that leaf and sometimes after creating nodes that are binding that leaf to the trie.
128     ///
129     /// \param node - pointer to the trie node that is going to be created
130     void trie_initialization(huff_trie *node) {
131         char curr_char = (char) process_byte_number();
132         long len = process_byte_number();
133         if (len == 0)
134             len = Symbols;
135
136         for (int i = 0; i < len; i++) {
137             if (m_current_bit_count == 0) {
138                 fread(&m_current_byte, 1, 1, m_compressed);
139                 m_current_bit_count = CHAR_BIT;
140             }
141
142             if (m_current_byte & Check) {
143                 if (!(node->one))
144                     node->one = new huff_trie;
145
146                 node = node->one;
147             } else {
148                 if (!(node->zero))
149                     node->zero = new huff_trie;
150
151                 node = node->zero;
152             }
153             m_current_byte <= 1;
154             m_current_bit_count--;
155         }
156         node->character = curr_char;
157     }

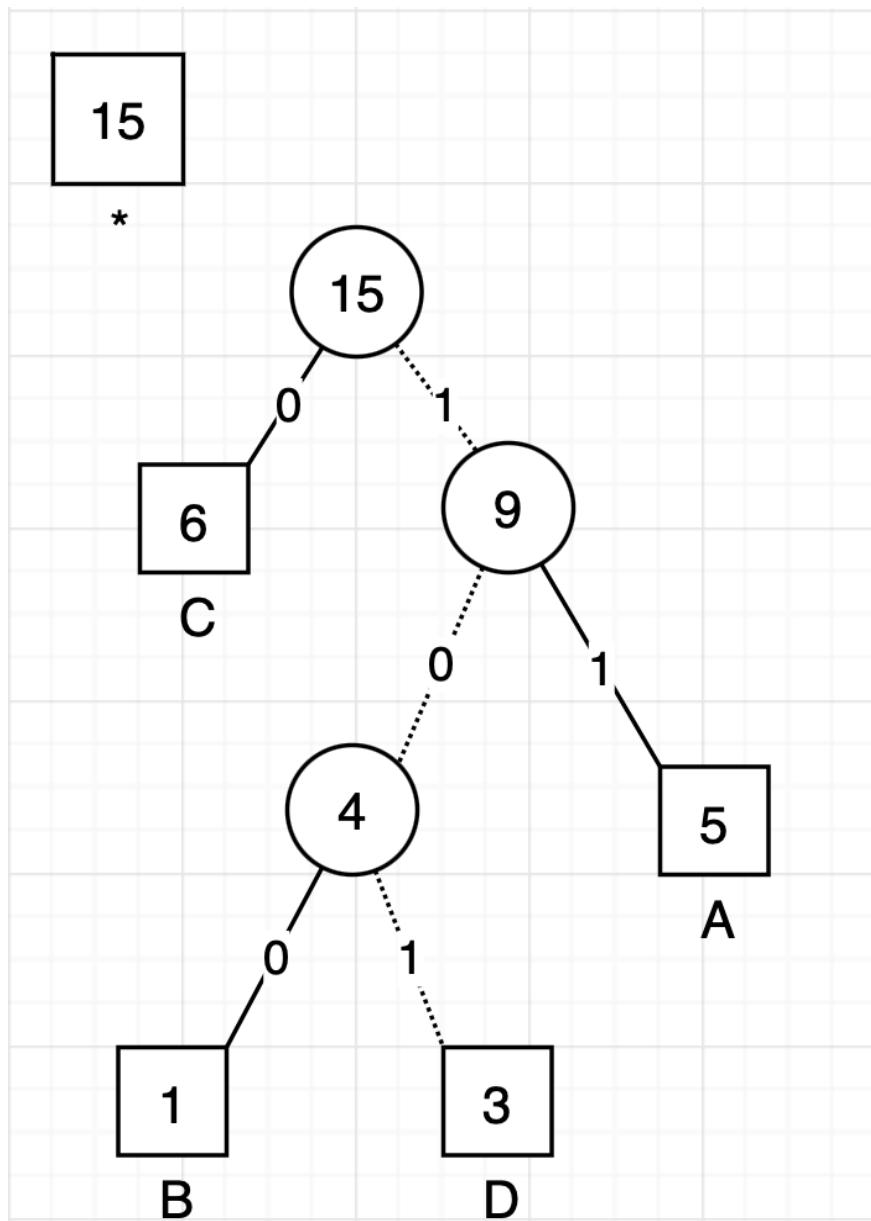
```

Фиг. 3.23 Създаване на хъфман дървото

Така при декомпресия се итерира спрямо прочетения бит. При прочетена нула се навлиза навътре в “huff_trie” дървото с референция наименована “zero”, докато при прочетена единица се навлиза с референция наименована “one”. Това продължава, докато не се достигне края на кода и съответния символ, отговарящ за него.

Например на фигура 3.24 е прочетен код 101. Започва да се итерира “huff_trie” структурата, докато бъде достигнат края.

Когато е достигнат последният елемент - се гледа символът на него.



Фиг. 3.24 Итериране през хъфман кодовете с цел декомпресиране

Обобщено - при декодиране на хъфман код се четат единиците и нулите от компресирания файл и се итерира през дървото, за да бъде достигнат съответният символ.

3.5.3 Декомпресиране на директория

Компресията поддържа компресиране на множество файлове и директории, съответно декомпресията трябва да може да декомпресира такъв компресиран файл.

За справянето с този проблем е използвана функцията “translation”, която създава файлове и директории в предварително записания, компресиран абсолютен или релативен път.

```
229     /// \brief translation function is used for creating files and folders inside given path
230     /// by using information from the compressed file.
231     /// whenever it creates another file it will recursively call itself with path of the newly created file
232     /// and in this way translates the compressed file.
233     ///
234     /// \param path - the file will be created here
235     /// \param change_path - if there are compressed folders - this flag allows recursion
236     void translation(const std::string &path, bool change_path) {
237         unsigned long file_count = get_file_count();
238         for (unsigned long current_file = 0; current_file < file_count; current_file++) {
239             long int size = 0;
240             bool file = is_file();
241             if (file)
242                 size = read_file_size();
243
244             std::string new_path = get_name();
245             if (change_path)
246                 new_path.insert(0, path + "/");
247
248             if (file) {
249                 if (size == 0) {
250                     Logger::the().log(spdlog::level::err, "Size cannot be "
251                                     "fetched from compressed file");
252                     return;
253                 }
254                 translate_file(new_path, size);
255             } else {
256                 fs::create_directory(new_path);
257                 translation(new_path, true);
258             }
259         }
260     }
```

Фиг. 3.24 Функцията “translation”, която служи за декомпресиране на директория (дървовидна структура)

Функцията “translation” (фигура 3.24) извършва декомпресирането на дървовидна структура от файлове и директории.

Ако се съдържа само файл - декомпресиран е само файл, ако е подадена директория, обаче, се създава директория и функцията извиква себе си наново, така че да се получи рекурсия.

Това са стъпки 3 до 7 от компресията (трета глава, точка [3.5]) или както се вижда на фигура 3.25.

3. 2 байта, които индицират колко е броя на файловете, които ще бъдат компресирани
4. 1 бит - показва дали ще бъде компресиран файл или ще бъде компресирана папка
5. 8 байта, които представляват размера на текущия входен файл
6. Бит групи
 - 6.1. 8 бита за дължината на името на текущия/текущата файл/папка
 - 6.2. Динамичен брой битове, които представляват трансформираната версия на името на текущия/текущата файл/папка
7. Динамичен брой битове, които представляват трансформираната версия на съдържанието на текущия/текущата файл/папка

Фиг 3.25 Стъпки при декомпресирането на дървовидна структура от файлове и директории

3.5.4 Частично декомпресиране

Декомпресията предоставя и възможността за частично декомпресиране. То представлява декомпресиране на файловете в дадена директория или на единичен файл. Реализирано е чрез функция, много подобна на тази за обикновеното декомпресиране, но с тази разлика че пропуска файловете, които не са подадени предварително.

За да бъде използвана тази способност - трябва да бъде подаден аргумент на предефинирания оператор “()” (фигура 3.25).

```
342     /// \brief The main function of decompression class that do all the magic with provided m_files.  
343     ///  
344     /// \param for_decompress - if folder name is different from empty string - then partial  
345     /// decompression is enabled at only the folder with name "for_decompress" will be decompressed  
346     void operator()(std::string_view for_decompress = "") {  
347         (*decompressor_impl)(for_decompress);  
348     }
```

Фиг. 3.25 Предефинираният оператор “()” в класа “Decompressor”

При различните стойности на аргумента се извършват различни операции:

- може да бъде стринг, представляващ името на папката или файла. Тогава ще се извърши частично декомпресиране и ще бъде декомпресирана само тази папка/файл.
- ако не е подадено нищо, тази променлива има стойност по подразбиране - празен стринг и ще се извърши обикновено декомпресиране на целия файл/файлове/дърворидна структура.

```
262     /// \brief translation function is used for creating files and folders inside given path  
263     /// by using information from the compressed file.  
264     /// whenever it creates another file it will recursively call itself with path of the newly created file  
265     /// and in this way translates the compressed file.  
266     ///  
267     /// \param for_decompress - folder to decompress  
268     /// \param change_path - if there are compressed folders - this flag allows recursion  
269     void translation_search(const std::string &path, std::string_view for_decompress,  
270                             bool change_path) {...}  
308
```

Фиг. 3.26 Функцията “translation_search”

Важно е да се отбележи, че частичното декомпресиране е възможно, само когато преди това е компресиран повече от един файл или е компресирана дърворидна структура, съдържаща файлове и директории.

3.6 Документация с времеви печат на събития

В проекта е използван и модул за документация с времеви печат на събитията (логове).

За реализацията на този модул е използвана библиотеката “spdlog” [11] и класът “Logger” (фигура 3.27), който е структуриран спрямо модела за единствена инстанция (Singleton).

```
1 #pragma once
2
3 #include <memory>
4 #include <spdlog/sinks/rotating_file_sink.h>
5 #include <spdlog/spdlog.h>
6
7     /// @brief Singleton and thread-safe
8 class Logger {
9 public:
10     /// @brief Singleton instance
11     /// @return Logger instance
12     static spdlog::logger &the() {
13         static Logger instance;
14         return *instance._logger;
15     }
16 private:
17     long max_size = 1048576 * 5; //!< 5MB max size of the log file
18     long max_files = 1; //!< 2 files to keep in rotation
19
20     std::shared_ptr<spdlog::logger> _logger =
21         spdlog::rotating_logger_mt(
22             "logger", "logs.txt", max_size, max_files
23         ); //!< logger instance
24
25     /// @brief Constructor
26     Logger() = default;
27 public:
28     Logger(Logger const &) = delete;
29     void operator=(Logger const &) = delete;
30 };
```

Фиг. 3.27 Класът използван за логовете

За да се направи дадена документация с времеви печат (фигура 3.28) се подават следните аргументи на метода “the”:

- Нивото, на което е документацията
 - “info” - за информация или текущ статус

- “err” - при възникнала грешка
- “warn” - за нещо, което трябва да се вземе под внимание

```
Logger::the().log(spdlog::level::info,
    R"(Compressor: started for "{0}" file/files/folder/folders with size: "{1}" bytes)",
    m_files.begin()->c_str(), m_all_size);
```

Фиг. 3.28 Примерна документация с времеви печат

Накрая документацията с времеви печат (логовете) се записват във файл, който се намира при изпълнимите файлове за проекта. Файла с логовете изглежда като на фигура 3.29.

```
[2022-03-28 12:08:50.717] [logger] [info] Compressor: started for "ForTesting" file/files/folder/folders with size: "10200000" bytes
[2022-03-28 12:08:51.041] [logger] [info] Compressor: initialized the trie with "21" symbols and "41" nodes
[2022-03-28 12:08:51.041] [logger] [info] Compressor: The size of the sum of ORIGINAL m_files is: "10200000" bytes
[2022-03-28 12:08:51.041] [logger] [info] Compressor: The size of the COMPRESSED file will be: "4462620" bytes
[2022-03-28 12:08:51.041] [logger] [info] Compressor: Compressed file's size will be [%43.75118] of the original file
[2022-03-28 12:08:52.040] [logger] [info] Compressor: created compressed file: "Test"
[2022-03-28 12:08:52.040] [logger] [info] Compressor: compression is completed

[2022-03-28 12:08:52.040] [logger] [info] Decompressor: Decompressing all files...
[2022-03-28 12:08:52.687] [logger] [info] Decompressor: Decompression is completed
```

Фиг. 3.29 Примерни лог съобщения

3.7 Тестове на отделните фрагменти код

Отделните модули на програмата са тествани за коректност при изпълнение. Направени са тестове на компресионния и декомпресионния модули, както и на В-дървото.

При тестовете е използвана библиотеката “catch2” [10], която позволява тестове на отделни фрагменти C++ код.

```

1 #include <core/storage/compression/tests/Shared.h>
2
3 // brief In this test - in the file are written 16 chars
4 // so if you want specific number of bytes(N)
5 // you give value of file_size with the formula N / 16
6 TEST_CASE("Compressor specific_file_size_test", "[specific_file_size_test]")
7 {
8     const int file_size = 32;
9     const std::string &file_name = "test.txt";
10    const std::string &compressed_file_name = "compressed";
11
12    const std::string_view text_in_file = "some text here.\n";
13
14    std::ofstream ofs(file_name);
15    for (int j = 0; j < file_size; ++j)
16        ofs << text_in_file;
17    ofs.close();
18
19    REQUIRE(exists(file_name));
20
21    compression::Compressor compress{
22        std::vector<std::string>(1, file_name),
23        compressed_file_name
24    };
25    compress();
26    REQUIRE(exists(compressed_file_name));
27
28    check_initial_compressed_size(file_name, compressed_file_name);
29    REQUIRE(clean({{"file_name", file_name},
30                  {"compressed_file_name", compressed_file_name}}));
31}

```

Фиг. 3.30 Примерен тест на компресията

Например на фигура 3.30 е тестван компресионния модул с различни по-големина файлове. На текущата снимка конфигурацията е с размер $32 \times 16 = 512$ байта, които са записани във файл “test.txt” и се очаква компресираният файл да е неименован “compressed”, като неговата големина трябва да е по-малко от оригиналния файл, което се проверява с функцията “check_initial_compressed_size”.

3.8 Сравнителен анализ на бързодействието на кодови фрагменти или функционалности

Освен че за отделните модули са направени тестове за коректност, също така е направен сравнителен анализ на бързодействието.

Използвана е библиотека “benchmark” [14] за оценка на всеки модул.

```
1 #include <core/storage/compression/benchmarks/Shared.h>
2
3     /// @brief Compression benchmark
4     ///
5     /// \param st - google benchmark state object that contains the
6     /// benchmark parameters
7     static void BM_Compression(benchmark::State &st) {
8         for (auto _ : st) {
9             st.PauseTiming();
10            std::ofstream ofs(file_name.begin());
11            for (int j = 0; j < st.range(0); ++j)
12                ofs << "some text here \n";
13            ofs.close();
14            st.ResumeTiming();
15
16            compression::Compressor{
17                std::vector<std::string>(1, file_name.begin()),
18                compressed_file_name
19            }();
20
21            st.PauseTiming();
22            clean({file_name.begin(), compressed_file_name.begin()});
23            st.ResumeTiming();
24        }
25    }
26
27    // 67108864 for 1GB ".txt" file -> ~105 seconds
28    // 65536 for 1MB ".txt" file -> ~110ms
29    // 64 for 1KB ".txt" file
30    BENCHMARK(BM_Compression)
31        ->Unit(benchmark::kMillisecond)
32        ->Iterations(10)
33        ->Arg(65536);
```

Фиг. 3.31 Сравнителен анализ за бързодействие на компресията

На фигура 3.31 е представен процеса за сравнителен анализ на бързодействието на компресиране модул. Изходът при изпълнението му може да се види на фигура 3.32.

```
2022-03-31T11:26:09+03:00
Running /home/stoyan/tmp/OriginalEugene/build/bin/CompBench
Run on (8 X 2600 MHz CPU s)
CPU Caches:
  L1 Data 32 KiB (x8)
  L1 Instruction 32 KiB (x8)
  L2 Unified 256 KiB (x8)
  L3 Unified 20480 KiB (x8)
Load Average: 0.28, 0.11, 0.07
-----
Benchmark           Time      CPU Iterations
-----
BM_Compression/65536/iterations:10    131 ms     131 ms       10
```

Process finished with exit code 0

Фиг. 3.32 Изход от сравнителен анализ на компресията

Четвърта Глава

Ръководство на потребителя

В тази глава се прави преглед на това какви допълнителни инструменти са необходими, как се придобива реализацията на проекта, как се изпълняват тестове и примерни програми. Също така е направен обзор на основните стъпки, които трябва да бъдат взети под внимание, при реализацията на нови СУБД, използвайки настоящата имплементация. В следващите секции биват използвани следните програми без да се разглежда инсталационния им процес: `git`, `make`, `gcc`.

4.1 Придобиване на проекта

За да може да бъде стартиран, изпробван и модифициран, проектът първо трябва да бъде изтеглен локално на потребителската машина. Това се осъществява посредством системата за управление на версии `git` и командата `git clone`:

```
$ git clone git@github.com:boki1/eugene.git
```

Фиг. 4.1 Команда за клониране на хранилището

Освен с имплементацията на проекта, потребителят се нуждае и от използваните библиотеки. За целта е необходимо да инсталира програмата `conan`, която се използва за управление на зависимостите на проекта. Най-лесният начин за това е да се използва `pip` посредством командата на фиг. 4.2:

```
$ pip install conan
```

Фиг. 4.2 Команда за инсталиране на `conan` dependency manager

Следващата стъпка е да бъдат инсталирани необходимите зависимости. Това може да бъде осъществено посредством предоставения `Makefile` и правилото `dep - setup`, посредством командата изложена на (фиг. 4.3).

```
$ make dep-setup
```

Фиг. 4.3 Команда за инсталиране зависимостите

При успешно компилиране и инсталиране на библиотеките може да се премине и към компилиране на проекта. Препоръчително е да се изпълнят и всички unit-тестове, за да се потвърди коректността на компилирания код. Това може да бъде направено чрез друго правило от предоставения Makefile (фиг. 4.4.).

```
$ make clean-test
```

Фиг. 4.4 Команда за изпълняване на unit-тестовете

Инсталацията се приема за успешна, ако изпълнението на горната команда завърши с резултат сходен на изложенияя на фиг. 4.4.

```
2022-03-28T09:32:41.7025124Z      Start 1: TestBtree
2022-03-28T09:32:42.1529916Z 1/7 Test #1: TestBtree
.....                           Passed   0.45 sec
2022-03-28T09:32:42.1562004Z      Start 2: TestNode
2022-03-28T09:32:42.1658761Z 2/7 Test #2: TestNode
.....                           Passed   0.01 sec
2022-03-28T09:32:42.1659459Z      Start 3: TestPager
2022-03-28T09:32:42.1727698Z 3/7 Test #3: TestPager
.....                           Passed   0.01 sec
2022-03-28T09:32:42.1728158Z      Start 4: TestUtil
2022-03-28T09:32:42.1762703Z 4/7 Test #4: TestUtil
.....                           Passed   0.00 sec
2022-03-28T09:32:42.1763165Z      Start 5: TestCompDecompTests
2022-03-28T09:32:43.7751183Z 5/7 Test #5: TestCompDecompTests
.....                           Passed   1.60 sec
2022-03-28T09:32:43.7752040Z      Start 6: TestCompTests
2022-03-28T09:32:43.7796749Z 6/7 Test #6: TestCompTests
.....                           Passed   0.00 sec
2022-03-28T09:32:43.7797525Z      Start 7: TestDecompTests
2022-03-28T09:32:43.7856920Z 7/7 Test #7: TestDecompTests
.....                           Passed   0.01 sec
2022-03-28T09:32:43.7859384Z
2022-03-28T09:32:43.7859811Z 100% tests passed, 0 tests failed out of 7
2022-03-28T09:32:43.7860071Z
2022-03-28T09:32:43.7860250Z Total Test time (real) =    2.13 sec
```

Фиг. 4.5 Примерни резултати от изпълнение на unit-тестовете на системата

4.2 Ръководство за имплементиране на бази данни

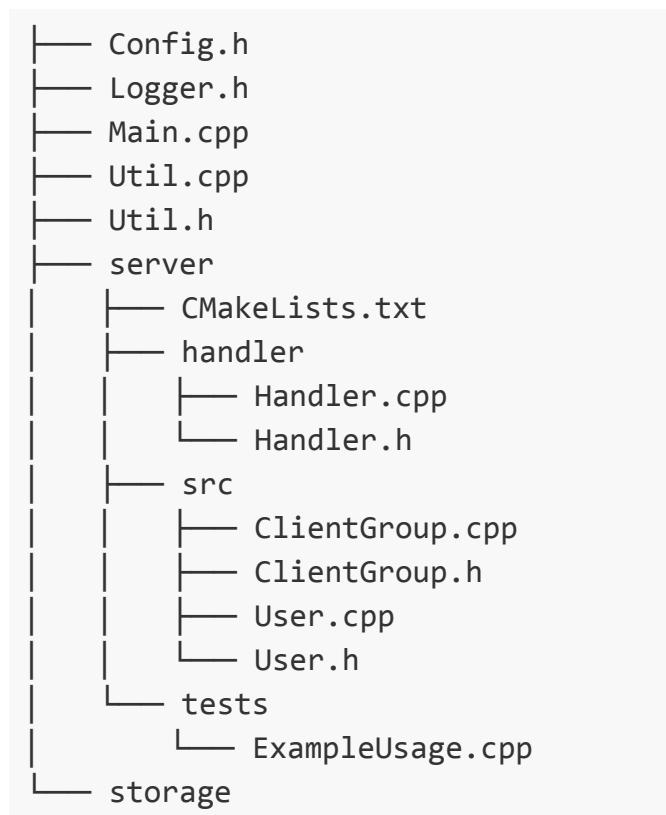
Конфигурация

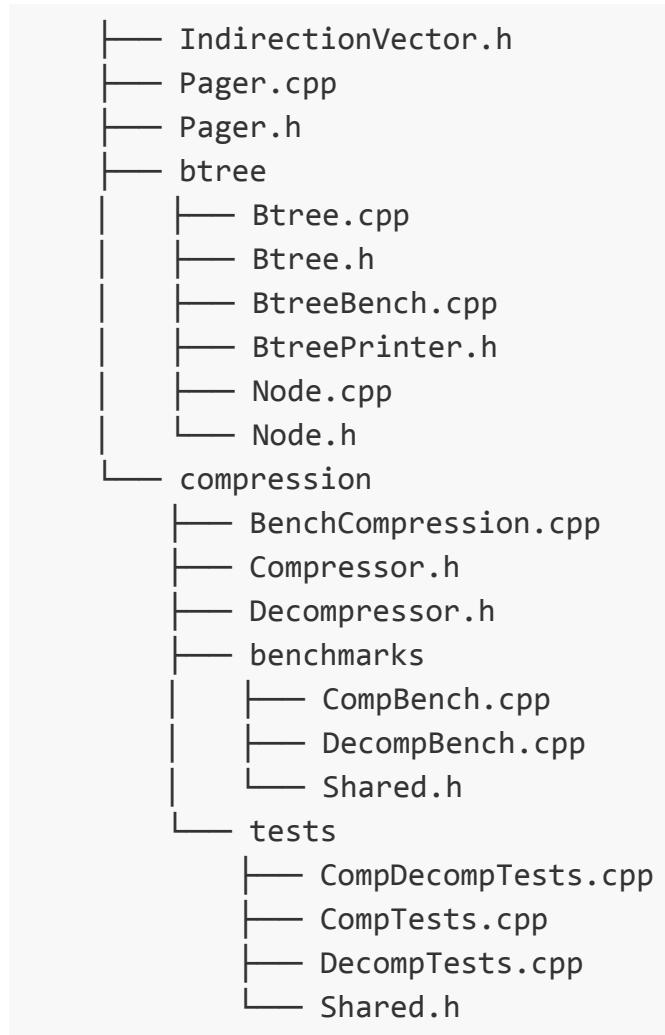
Първата стъпка при създаване на нова база данни, използвайки библиотеката, е да се дефинира конфигурация. Това се случва, като се наследи класа Config, намиращ се в Config.h. Съществуват различни видове конфигурации според желаната функционалност.

Пример: Ако желаните да се запишат данни са относно Човек, който бива характеризиран от име (динамично оразмерен std::string, години (std::uint32_t) и телефонен номер (статично оразмерен Phonenumbers), то тогава конфигурацията може да бъде сходна на представената във фиг. 4.5.

4.2.1 Структура на директориите

Имплементацията на библиотеката се намира в `src/core/`, където файловете биват разпределени според функционалности, които имплементират.





Фиг. 4.6 Структура на директориите и разпределени на файловете в библиотеката

4.2.2 Интерфейс към модулите

Модул Pager	
Идентификатор	Предназначение
клас BadAlloc	Изключение, използвано при възникнала грешка свързана с алокация.
клас BadPosition	Изключение, използвано при възникнала грешка свързана с грешна позиция.
клас BadWrite	Изключение, използвано при възникнала

	грешка свързана със записване на данни.
клас BadRead	Изключение, използвано при възникнала грешка свързана с прочитане на данни.
абстрактен клас GenericPager	Дефинира основния интерфейс към Pager модула.
интерфейс IPersistentPager	Дефинира функции за запазване на състоянието на Pager инстанция.
интерфейс ISupportingInnerOperations	Дефинира функции за опериране със съдържанието вътре в страница.
клас Pager	Конкретна имплементация на GenericPager, която имплементира и ISupportingInnerOperations и IPersistentPager.
клас InMemoryPager	Конкретна имплементация на GenericPager, която имплементира съдържа изцяло в паметта. Използва PageCache с NeverEvictCache стратегия.
клас StackSpaceAllocator	Алокатор на страници базиран на стак.
клас FreeListAllocator	Алокатор на страници базиран на списък със свободните страници.
клас PageCache	Кеш за страници.
клас LRU Cache	Стратегия за премахване на страници базирана на least-recently used метода.
клас NeverEvictCache	Стратегия за премахване на страници,

	която никога не гони принудително.
--	------------------------------------

Модул Btree	
Идентификатор	Предназначение
клас BadTreeRemove	Изключение, използвано при възникнала грешка свързана с триене на запис.
клас BadTreeSearch	Изключение, използвано при възникнала грешка свързана с търсене на запис.
клас BadTreeInsert	Изключение, използвано при възникнала грешка свързана със записване на данни.
клас BtreePrinter	Изключение, използвано при възникнала грешка свързана с прочитане на данни.
енумератор клас ActionOnConstruction	Означава действието, което да бъде предприето при инициализация.
енумератор клас ActionOnKeyPresent	Означава действието, което да бъде предприето при създаване на запис, който вече е наличен.
клас Btree	Основният клас, имплементиращ API към модула В-дърво.
клас Btree::Header	Заглавната част на В-дърво, която съдържа метаданните на инстанцията, без потребителските данни.
клас Btree::MemConfig	Еквивалентна конфигурация, която да съхранява дървото в паметта, вместо на диска.

<code>клас Btree::PosNod</code>	Възел заедно с информация за неговата позиция на диска и в дървото.
<code>клас Btree::TreePath</code>	Описва път в дървото.
<code>клас Btree::InsertionReturnMark</code>	Краен резултат от извършено вмъкване.
<code>клас Btree::InsertionTree</code>	Дърво създадено по време на групово вмъкване.
<code>клас Btree::SearchResultMark</code>	Краен резултат от извършено търсене.
<code>енумератор клас Btree::CornerDetail</code>	Дескриптор за операцията, която намира ъгъл в дървото.
<code>енумератор клас MakeRootAction</code>	Означава действието, което да извърши при създаване на нов корен.
<code>клас Btree::RemovalReturnMark</code>	Краен резултат от извършено премахване.
<code>клас BadIndVector</code>	Изключение, използвано при възникнала грешка свързана с достъпа до вектора за индиректност.
<code>клас Slot</code>	Дескриптор на използвани слот, намиращ се във вектора за индиректност.
<code>клас BadTreeAccess</code>	Изключение, използвано при възникнала грешка свързана с достъп до дървото.
<code>енумератор клас LinkStatus</code>	Изброяване описваща различните състояния на връзки от възел към друг.

енумератор клас SplitBias	Описва различните метод, според които да бъде изчислена точка на пречупване във възлите по време на разделяне.
енумератор клас SplitType	Описва как операцията по разделяне се отнася към междинния елемент.
клас Node	Възел от дървото.
клас Node::Branch	Метаданни на разклоняващ възел в дървото.
клас Node::Leaf	Метаданни на листо в дървото.
клас Node::Entry	Запис, съхраняван в дървото - ключ и стойност.
енумератор клас Node::RootStatus	Състояние на възела спрямо корена.

Компресионен модул	
Идентификатор	Предназначение
клас compression::Compressor	Компресира дадена директория, файл, файлове, дървовидна структура
клас compression::storage::detail::CompressorInternal	Вътрешен клас, скрит от потребителя, който реализира логиката за компресиране. Използван е при тестовете.
структура compression::storage::detail::CompressorInternal ::huff_trie	trie структура, в която са записани хъфман кодовете за съответните символи

Декомпресионен модул	
Идентификатор	Предназначение
клас decompression::Decompressor	Декомпресира дадена директория, файл, файлове, дървовидна структура
клас decompression::storage::detail::DecompressorInternal	Вътрешен клас, скрит от потребителя, който реализира логиката за декомпресиране. Използван е при тестовете.
структура decompression::storage::detail::DecompressorInternal::huff_trie	trie структура, която извлича от компресирания файл хъфман кодовете за съответните символи

Logger	
Идентификатор	Предназначение
клас Logger	Клас, структуриран спрямо модела за единствена, който позволява документация с времеви печат на събитията

Сървърен модул	
Идентификатор	Предназначение
клас Handler	Клас, който позволява “http” заявки

клас CredentialsStorage	Абстрактен клас, който трябва да се имплементира от потребителя, за да се записват данните за username и парола. Поддържа “basic authentication”.
клас Storage	Абстрактен клас, който трябва да се имплементира от потребителя, за да служи като база данни.

Заключение

Днешният свят се характеризира с повсеместно навлизане на ИКТ във всички аспекти на живота. Чрез различни технологични средства днес почти всичко може да бъде измерено, документирано и трансформирано в данни. Огромни по обем, разнообразие и темп на растеж и промяна данни се събират, съхраняват и натрупват в компютърните системи. Във връзка с полезното използване на натрупаните данни извлечането на информация (от англ. information retrieval) е една от най-активно развиващите се области на информатиката.

Във връзка с целта на настоящата дипломна работа и поставените изследователски задачи, в заключение могат да бъдат формулирани следните по-важни резултати:

- Бе направено подробно проучване за избор на технологии за реализация на поставените задачи.
- След направените проучвания успешно беше изградена система за съхранение, извлечане и обработка на големи графове базирани данни, съдържаща модулите Pager, В-дърво, Compression и Server.
- Pager подсистемата управлява физическото пространство, върху което се разполага базата данни. В-дървото структурира наредените данни и предоставя възможност за тяхната обработка. Compression модулът реализира функционалност за минимизиране на заетия обем, а Server предоставя възможност за достъп до данните от отдалечени машини.

Изискванията наложени в заданието на дипломната работа бяха изпълнени.

Съдържание

Използвани съкращения	2
Увод	3
Първа глава: Методи при съхраняването на големи обеми от данни (СУБД)	5
 1.1 СУБД	5
 1.1.1 Характеристиките на една СУБД са:	5
 1.1.2 Положителните страни на една СУБД са:	6
 1.1.3 Отрицателните страни на една СУБД са:	6
 1.2 Архитектура на СУБД	6
 1.2.1 Едностепенна архитектура	7
 1.2.2 Двустепенна архитектура	7
 1.2.3 Тристепенна архитектура	8
 1.3 Типове СУБД	9
 1.3.1 Централизирани СУБД	10
 1.3.2 Дистрибутирани СУБД	10
 1.3.3 Релационни СУБД	12
 1.3.4 Нерелационни СУБД	12
 1.4 Независимост на данните	13
 1.4.1 Логическа независимост на данните	14
 1.4.2 Физическа независимост на данните	15
Втора глава: СУБД и нейните особености	16
 2.1 Технологични изисквания	16
 2.2 Избор на език за програмиране	16
 2.3 Използвани библиотеки	19
 2.4 Менажиране на различните библиотеки/зависимости	20

2.5 Организация на работата, СІ	21
2.6 Структура на системата	22
2.6.1 Компресия	23
2.6.1.1 Кодиране по дължина	25
2.6.1.2 Речниково кодиране	26
2.6.1.3 Кодиране с частично съвпадение	27
2.6.1.4 Ентропологично кодиране [19]	29
2.6.1.5 Аритметично кодиране [20]	29
2.6.1.6 Кодировка на Хъфман (Huffman coding)	29
2.6.2 Pager	30
2.6.3 В-дърво	30
2.6.4 Комуникация, сървърна архитектура	31
Трета Глава: Реализация на СУБД	32
3.1 Сървърен модул	33
3.1.1 Авторизация	33
3.1.2 Запис на информация	34
3.1.3 Изтриване на информация	35
3.1.4 Извличане на информация	35
3.2 В-дърво	36
3.2.1 Свойства на В-дървото	37
3.4 Компресионен модул	37
3.4.1 Компресионен модул първа част	39
3.4.1.1 Информация за размера на файла	41
3.4.1.2 Отчитане на честотата на използваните уникални байтове и броя на уникалните байтове	42
3.4.1.3 Създаване на основното дърво, което ще послужи по-късно за транслация на уникалните байтове	43
3.4.2 Компресионен модул втора част	47

3.5 Декомпресионен модул	51
3.5.1 Структура на потребителския клас “Decompressor”	52
3.5.2 Логика на декомпресията	53
3.5.3 Декомпресиране на директория	56
3.5.4 Частично декомпресиране	57
3.6 Документация с времеви печат на събития	58
3.7 Тестове на отделните фрагменти код	60
3.8 Сравнителен анализ на бързодействието на кодови фрагменти или функционалности	61
Четвърта Глава: Ръководство на потребителя	64
4.1 Придобиване на проекта	64
4.2 Ръководство за имплементиране на бази данни	66
4.2.1 Структура на директориите	66
4.2.2 Интерфейс към модулите	67

Използвана литература

- [1] "Structured Query Language (SQL)". International Business Machines. October 27, 2006
- [2] db2-big-sql, <https://www.ibm.com/docs/en/db2-big-sql/7.1>, 2020 - 2021
- [3] Alexandrescu A., "Modern C++ Design: Generic Programming and Design Patterns Applied", ISBN 978-0-201-70431-0, 2001
- [4] C++ Reference, en.cppreference.com/w/cpp, 2022
- [5] Git Source Control Management, <https://git-scm.com>, 2022
- [6] Memory Management, <https://www.memorymanagement.org>, 2022
- [7] Huffman, D., "A Method for the Construction of Minimum Redundancy Codes", 1952
- [8] CMakeList ref, <https://cmake.org>, 2022
- [9] CI, <https://docs.github.com/en/actions/automating-builds-and-tests/about-continuous-integration>, 2021
- [10] catch2, <https://github.com/catchorg/Catch2/blob-devel/docs>, 2022
- [11] spdlog, <https://github.com/gabime/spdlog>, 2022
- [12] fmt, <https://github.com/fmtlib/fmt>, 2022
- [13] andreasbuhr-cppcoro, <https://github.com/andreasbuhr/cppcoro>, 2022
- [14] benchmark, <https://github.com/google/benchmark>, 2022
- [15] cpprestsdk, <https://github.com/Microsoft/cpprestsdk>, 2022
- [16] Conan, <https://conan.io>, 2022
- [17] LZ78, <https://math.mit.edu/~goemans/18310S15/lempel-ziv-notes.pdf>, 2015
- [18] GraphChi, <https://github.com/GraphChi>, 2022
- [19] Entropy compression from Terence Tao, <https://terrytao.wordpress.com/2009/08/05/mosers-entropy-compression-argument/>, 2009

[20] Howard, Paul G.; Vitter, Jeffrey S. - Arithmetic coding for data compression,

[https://web.archive.org/web/20131018025150/http://home.tiscali.nl/rajduim/Vid eo%20Compression%20Docs/Arithmetic%20Coding%20For%20Data%20Com pression%20\(2\).pdf](https://web.archive.org/web/20131018025150/http://home.tiscali.nl/rajduim/Vid eo%20Compression%20Docs/Arithmetic%20Coding%20For%20Data%20Com pression%20(2).pdf), 1994

[21] Thenmozhi, M.; Srimathi, H. (February 2015). "An Analysis on the Performance of Tree and Trie based Dictionary Implementations with Different Data Usage Models",

<https://sciresol.s3.us-east-2.amazonaws.com/IJST/Articles/2015/Issue-4/Article 2.pdf>, 2015

[22] PImpl метод, <https://en.cppreference.com/w/cpp/language/pimpl>

[23] В-дърво дефиниция, Bayer, R.; McCreight, E. - "Organization and maintenance of large ordered indices"

[24] B+ дърво и СУБД, Comer, D., "The Ubiquitous B-Tree", 1979

[25] Bayer R. , "Organization and maintenance of large ordered indices", 1970

[26] Github, <https://github.com>