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: **Кристиян Стоименов**

Дипломен ръководител: **инж. Любомир Стоянов**

2022

Използвани съкращения

На български

- СУБД - Система за управление на бази данни
- РСУБД - Релационна система за управление на бази данни
- БДТ - Бинарно (двоично) дърво за търсене
- БТ - Бинарно търсене
- ООП - Обектно-ориентирано програмиране
- БД - База данни
- РБД - Релационна база данни
- НБД – Нерелационна база данни
- ИКТ - Информационни и компютърни технологии

На английски

- I/O - Input/Output
- SQL - Sequential Query Language
- ACID - Atomicity, Consistency, Isolation, Durability
- CAP - Consistency, Availability, Partition tolerance
- XML - Extensible Markup Language
- JSON - Javascript Object Notation
- RAM - Random Memory Access
- B - Bytes
- KB - Kilobytes
- RAII - Resource Acquisition Is Initialization
- FOSS - Free and open-source software
- REST - Representational State Transfer
- API - Application Programming Interface
- LRU - Least recently used

Увод

Непрекъснатият прогрес в областта на информационните и комуникационните технологии (ИКТ) открива възможности за съхраняването, обработката, предоставянето на достъп и извличане на данни от информационни масиви. През последните десетилетия нарастването на обема на достъпната информация чрез интернет води до необходимост от съхраняване на големи масиви от данни и еволюция на системите за управление на бази данни.

Често технологичните решения са силно обвързани със съхраняване на информация – детайли за потребителски акаунти, записи от измервания на устройства, запазване на резултати, изискващи голяма изчислителна мощ и много други. В хода на развитие на системите за съхранение на данни, те се превръщат във фундаментална част от системите и приложенията, осигуряващи крайните решения и услуги за потребителите.

Цел на настоящата дипломна работа е създаването на система, предоставяща основните функционалности на базите данни, и, под формата на библиотека, да даде възможност на потребителя лесно да създаде база данни спрямо конкретиката на ситуацията, в която ще бъде използвана.

От поставената цел произтичат следните **задачи**:

- Събиране на информация и изследване на алгоритмите и структурите от данни, които се помещават в реална имплементация на система, управляваща големи обеми от данни.
- Определяне на критичните характеристики и създаване на архитектура, която взима предвид особеностите на системата.
- Изработване на система за управление на база данни.

Дипломната работа е структурирана в четири глави и ... приложения.

В **Глава 1** е направен обзор на функциите, състоянието и тенденциите за развитие на базите данни. Разгледани са основните структури, които се използват като основа. Също така е направен преглед на доказали се системи, които се използват от приложения и разработчици по целия свят.

Във **Глава 2** са изложени функционалните изисквания към дипломния проект, както и е представена теоретичната постановка на основните алгоритми, които се

срещат в имплементацията на база данни. Определена е архитектурата на приложението. Направен е аргументиран избор на технологии.

В **Глава 3** е описана създадената система, отговаряща на поставените изисквания. Разгледани са подробно отделните модули и техните компоненти, които формират отделните функционалности на проекта, в частност подсистемата свързана със съхранение, извличане и обработване на данни.

В **Глава 4** е поместено ръководство за потребителя, като потребител на проекта може да бъде както краен потребител, който използва системата, като краен продукт, така и програмист, който използва разработката като библиотека, върху която може да се надгражда, взимайки предвид конкретната ситуация.

Глава 1. Системи за управление на бази данни

1.1. Основни характеристики на СУБД

Основно ядро на съвременните информационни системи са базите данни. Днес те предоставят бърз и сигурен начин за съхранение на данни.

База данни (БД) *е съвкупност от свързани данни, представящи дадена предметна област*.

Данните са структурирани по начин, който позволява лесното и бързото им извличане, преглеждане, търсене, осигуряващо намаляване дублирането на информация, гарантиране на цялостност и наличност на данните.

Системата за управление на бази данни (СУБД) е предназначена за дефиниране, организация, обработка и съхранение на информацията, структурирана в БД.

Основните операции, които се поддържат от съвременните бази данни са свързани с *въвеждане* (insert, post), *извличане* (select, get) и *премахване* (remove, delete). СУБД трябва да позволява обработката на данни, свързана с извличане и промяна на данни.

Към съвременните СУБД се поставят следните по-важни изисквания:

- да предлага инструменти за дефиниране на обектите на дадена БД, връзките между тях и дефиниционните области (домени) на отделните атрибути.
- да осигурява конкурентен едновременен достъп от потребители и приложения при запазване на валидността и целостта на данните.
- да осигурява логическа независимост на данните от тяхното физическо представяне и прозрачност на процесите за обработка и съхранение в БД.
- да осигурява оптимална структура на различните видове файлове и методи за достъп, гарантиращи бърза обработка на данните.

Други характеристики на СУБД са свързани със защита от неоторизиран достъп, представяне и визуализация на данни, осигуряване на поддържане на БД и свързаните с нея програми по време на целия жизнен цикъл на приложенията и др.

1.2. Видове бази данни

В литературата БД се разделят на два основни типа – **релационни** (РБД) и **нерелационни** (НРБД).

Релационните (SQL) бази съхраняват данните по предварително структуриран начин – в таблици, подредени в редове (записи) и колони. Между отделните данни и таблици могат да се създават връзки (релации) чрез използване на *първичен ключ* (primary key) и *външен ключ* (foreign key). Първичният ключ еднозначно дефинира записите в дадена таблица, а външният ключ представлява първичен ключ на друга таблица. На фиг. 1.1. е илюстрирана тази концепция.

id	name
1	Ivan
2	Petar
3	Georgi

a) Ученици

id	grade	student_id
1	6	2
2	5	1
3	6	3

b) Оценки

```
select name from Student
left join Grade
on Student.id = Grade.student_id
where Grade.grade < 6;
```

c) Заявка

Фиг. 1.1 - Примерна РБД с таблици за ученици и техните оценки

В таблицата „Оценки” (фиг.1.1b), заедно със стойностите на оценките, всеки запис включва и външен ключ към таблицата „Ученици“ (фиг.1.1a). Това предоставя възможност на потребителя да комбинира двете таблици при извършване на заявки. Този тип операция е известен като *join* – например, за да се сдобием с имената на всички ученици, които имат оценка по-ниска от 6 ни трябва не само „Оценки”, но и „Ученици”, тъй като в първата таблица се съхраняват техните имена (фиг.1.1c).

Традиционните РБД трябва да удовлетворяват т.нар. **ACID** правила за всяка транзакция (една логическа операция) [29]:

- **A**tomicity (*атомарност*) на изпълняваните транзакции
- **C**onsistency (*последователност*) – непротиворечивост на промените от дадена транзакция
- **I**solation (*изолация*) на транзакциите – не може да се достъпи информация, която се обработва от незавършила транзакция.
- **D**urability (*дълготрайност*) – когато дадена транзакция е приключила, се запазват направените от нея промени

Основните предимства на РБД се определят от възможността да обработват комплексни заявки, при които е необходимо да се реферират множество таблици. Също така позволяват ясно изразена последователност – операциите свързани с всяка една от заявките имат много точна синхронизация, следвайки ACID характеристиките. Релационните СУБД имат ясно дефиниран модел, който веднъж уточнен не се променя във времето. Тази тяхна характеристика попада както в предимствата, така и в недостатъците – възможността от грешки значително намалява (полетата са типизирани, например не може да се постави число в полето за име на ученик), но при желание за смяна на модела, мигрирането на данните е много сложен процес, понякога невъзможен. Допълнително усложнение е, че при РБД има възможност само за вертикална скалируемост. Нарастването на количеството данни изисква разширяване на капацитета за съхранение и мощността на обработка на съществуващия изчислителен стълб, а повишаването на броя на машините няма да доведе до увеличаване на производителността.

За обработка и управление на данни от нерелационен тип (*текст, изображения и др.*) се използват *нерелационни* (NoSQL) бази данни.

Най-често използваните технологии за съхранение на данните в НРБД са *„ключ-стойност”* и *документни*. При „ключ-стойност” системата наподобява асоциативен масив - подавайки ключа, се извлича стойността. В наредената двойка „ключ-стойност” ключът е уникален идентификатор на записа (сходно на първичния ключ при РСУБД), а стойността - списък от *полета* (може и да е празен). Основно предимство на този подход е малкия обем памет, който се използва спрямо SQL. Съществуват разнообразни имплементации на този тип СУБД, сред които са Cassandra, Redis и традиционния за UNIX-базираните операционните системи - DBM. В документните НРБД записите са организирани в *документ* в XML, JSON или друг подходящ формат. Това наподобява концепцията за обектно-ориентирано програмиране и двете идеи работят добре заедно. Тази технология отново използва „ключ-стойност” двойки, в които стойността е документ, представен също във вида „ключ-стойност.” Основната разлика между двете е, че „ключ-стойност” СУБД не се интересува от вида данни, които се съхраняват, докато при документите СУБД, системата има необходимост да знае какво бива съхранявано вътре, за да може да отвръща подходящо на потребителските заявки. Сред популярните решения е MongoDB.

NoSQL базите са с вградена в архитектурата способност за автоматично разпределяне на БД върху множество сървъри, облачни инстанции и други, което позволява обработка и обновяване на големи обеми данни в реално време с висока ефикасност. Поради хоризонталното мащабиране при НРБД не могат да се гарантират ACID характеристиките на традиционните РБД. Използва се теоремата **CAP** (Теорема на Брюър), според която трябва да се изберат две от следните свойства [4]:

- **C**onsistency (*последователност*) – всички клиенти на БД виждат една и съща информация.
- **A**vailability (*наличност*) – всички клиенти на БД имат достъп до данните.
- **P**artition tolerance (*възможност за разделяне*) – БД може да се разделя и локализира върху множество сървъри.

Тъй като едновременното удовлетворяване на трите свойства е невъзможно, се използва т.нар. *евентуална последователност* (от англ. eventual consistency), за да се осигурят *наличност* и гаранции за *толерантност на дяловете* с намалено ниво на съгласуваност на данните [5]. Евентуалната последователност гарантира, че след изпълнение на всички актуализации, от даден момент нататък всички прочитания връщат идентичен резултат.

Сред предимствата на НРД е скоростта на изпълнение на заявките. Тъй като най-често в стойността се съдържат всички асоциирани с ключа данни, не се налага да се извършват комбинации с други записи (или множество записи), което прави НРБД подходящи за употреба в продукти и системи, където ниското ниво на латенция е от високо значение. Друга важна характеристика е възможността за хоризонтална скалируемост.

В Таблица 1.1 са посочени основните характеристики на двата основни вида бази данни.

Таблица 1.1 Сравнение между РБД И НРБД

	РБД	*НРБД*
Данни	Структурирани – дати, номера, ЕГН-та, имена, адреси, графични данни (чертежи, диаграми).	Неструктурирани – документи, презентации, мултимедийни данни (графика, анимация, аудио и видео)

Структура	Предварително дефинирани модели.	Предварително неизвестна и динамична
Заявки	Формални, еднозначни	Не са подходящи за сложни заявки
Видове	Базирани на таблици	Документни, „ключ-стойност“, колонни, граф, обектни и др.
Скалируемост	Вертикално мащабируеми	Хоризонтално мащабируеми
Примери	Oracle, MS SQL Server, MySql, PostgreSQL	BigTable, Cassandra, Redis, MongoDB, Hbase

1.3. Структури, използвани при реализацията на СУБД

Двата най-често срещани типа структури, на които се базират БД са асоциативни масиви (*хеш-таблици*) и *дървета* [7,11,13,15].

1.3.1. Хеш-таблици

Хеш-таблицата е структура от данни, за която е характерен директен достъп до данни от произволен тип. Сложността на най-често използваните елементарни операции (вмъкване, изтриване и извличане) е константна $\theta(1)$. Асоциативният масив е оптимален метод за съхранение на данни, когато се намират в RAM паметта на машината, на която функционира СУБД.

Хеш-таблицата съдържа двойки „ключ-стойност“.

Извършва се процеса хеширане, при който се подава аргумент, посредством математическа зависимост (*хеш-функция*), определя *хеш-стойност* за аргумента.

Основава се на математическото понятие *хеш-функция*, която трансформира подаден елемент (ключа) в число от определен краен интервал (интервал на хеш-функцията). То служи като *ключ* (индекс) в асоциирания масив.

Изборът на хеш-функция се определя от две неща – да не отнема много изчислително време и да разпределя елементите относително равномерно в различните хеш-адреси. Често при реализирането на този тип структури от данни възникват проблеми с разпределението на данните от хеш-функцията в свободната част на

интервала и възникването на съвпадения (*колизии*), т.е различни стойности на аргумента, чиято хеш-стойност съвпада. [43].

Съществуват два основни метода за реализиране на хеш-таблици: *отворено* и *затворено* хеширане. Основната разлика между тях е свързана с решението, което използват за справяне с колизии. При първия обикновено се използва външна динамично заделяна памет – елементите със съвпадащи хеш-адреси се записват в *свързан списък (списък на препълванията)*. При затвореното хеширане, повтаряме хешрането добавяйки допълнителен дескриптор към стойността, която хешираме. И двата метода са изследвани в литературата, гарантирайки константна сложност на операциите [11, 12].

1.3.2. Дървовидни структури за търсене

Бинарното дърво за търсене (БДТ) е често използван механизъм за съхраняване на данни, които имат определена наредба [13].

a) Общ вид на БДТ b) БДТ със сложност $O(n)$

Фиг. 1.2 - Схематично представяне на БДТ

На фиг.1.2 а) е показан най-общ вид на бинарно дърво. Всеки възел има по двама наследници, които индуцират *ляво* и *дясно* поддърво. Съхранява единствена стойност – ключ, който притежава по-голяма стойност от тези в лявото поддърво, и по-малка от тези в дясното. Представянето на данните на базата на тази структура гарантира операции със средна сложност $O(logn)$, където n е текущият брой на възли в бинарното дърво. Въпреки предимствата, съществуват случаи, при които използването на БДТ не може да предостави логаритмична сложност на операции, а се държи като свързан списък.

На фиг. 2.2b) е илюстрирано дърво, в което всеки нов добавен възел е по-голям от предишните. Нови записи се добавят единствено като десни наследници на възлите, довеждащи до $\theta(n)$ операции. Този проблем е известен от над 50 години. Първото решение, което успява да гарантира $\theta(logn)$ сложност, независимо от поредността на въвеждане на възлите, е структурата AVL дърво [14]. То е базирано на БДТ, добавяйки изискване – височината на всички поддървета, индуцирани от възли с общ родител, да се различават с не повече от 1.

$$|h_L - h_R| \leq 1 \tag{1.1}$$

Където h_L е височината на лявото поддърво, а h_R - на дясното.

Основният подход, който се използва, за да се осигури тази характеристика са ротации. Ротацията представлява операция върху БДТ, която размества елементи, запазвайки свойствата му. Сложността ѝ е $O(1)$, тъй като само константен брой възли трябва да бъдат достъпени. Методът е показан на фиг. 1.3.

AVL дървото е представител на категорията балансирани бинарни дървета. Основната им цел е да поддържат максималната си височина възможно най-малка при извършване на промяна върху съхраняваните възли. Друга често срещана модификация на БДТ, която има характеристики на балансирано дърво, е *черно-червеното дърво*. При нея алгоритмите за операциите “вмъкване”, “търсене” и “изтриване” се различават от тези на AVL, но основно са базирани на ротации и гарантират горна граница $O(logn)$ и при трите операции.

Фиг. 1.3 - Ротации върху БДТ

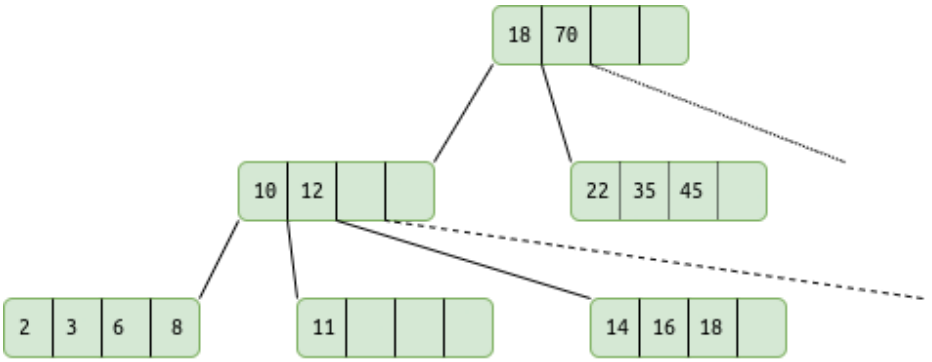
1.3.3. B-дърво като основа на СУБД

B-дървото се е установила като стандартен подход при реализацията на СУБД. През годините дефиницията на структурата се адаптира към усъвършенстването на технологиите - по-голям размер на възлите с увеличаване на размера памет, въвеждане на нови алгоритми и промени върху структурата с цел оптимизация на дадени операции. B-дърво от ред m, наричаме дърво за търсене със следните свойства [3, 7, 10, 15]:

- Всички възли имат не повече от m разклонения.
- Всички възли с изключение на корена и листата имат поне $\frac{m}{2}$ разклонения.
- Коренът има поне две разклонения, освен ако не е листо.
- Всички листа се намират на една и съща дълбочина и не носят информация.
- Всички възли, които не са листа имат с едно повече разклонения, отколкото ключове.

Пример за B-дърво[1] е показана на фиг. 2.4.

[1] Илюстрирани единствено ключовете в дървото без асоциираните към тях данни.



Фиг. 1.4 - B-дърво при $m = 2$

Дърветата от този вид по дефиниция са идеално балансирани [15] - разликата между всеки две поддървета е не повече от 1. При B-дърво тази разлика е 0 по определение. Можем да изчислим броят на листа спрямо броя на записи (чифтове „ключ-стойност”), по следната формула:

$$\log_{\frac{m}{2}} \left(\frac{n+1}{2} \right) \tag{1.2}$$

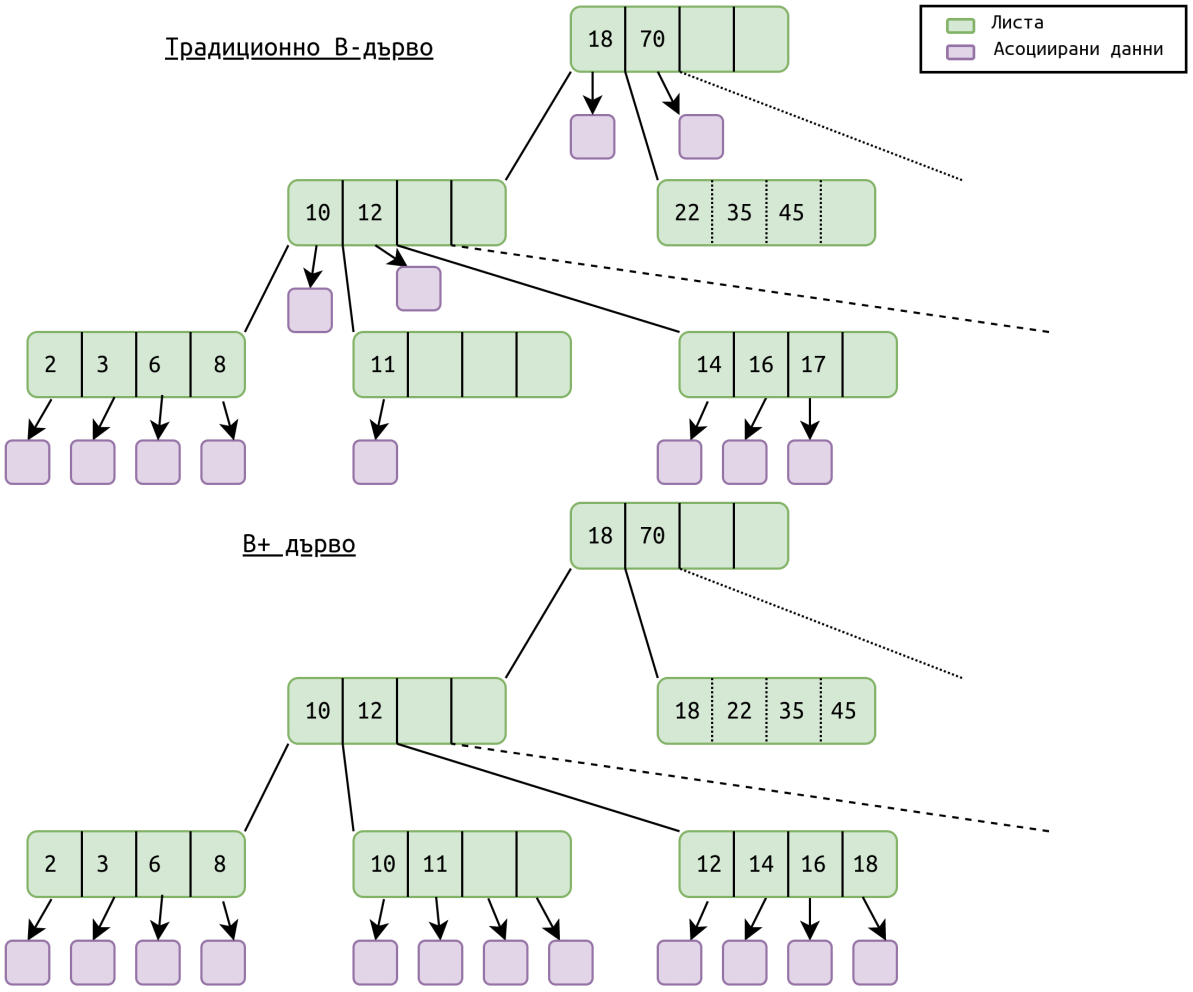
Където n е броят на записи, а m - редът на дървото.

Основните предимства на B-дървото са свързани с големина на възлите и множеството на брой стойности, които се съхраняват във всеки един от тях. Позволява се максимално количество ключове да бъдат запазени в малък брой възли, което минимизира броя прочитания. Това свойство е особено полезно при реализацията на СУБД, тъй като информацията се съхранява на твърдия диск. Извършването на I/O операции са основен проблем в СУБД. Съхранявайки множество стойности, се позволява на текущата операция да придобие по-добър ориентир относно позицията на търсените данни, изключвайки по-голяма част от дървото от следващите стъпи на претърсването при всяко едно сравнение. Това гарантира, че операциите, извършени върху B-дърво осъществяват достъп до не повече листа от височината на дървото[2].

Развитието на тази структура от данни бързо и подсигурява място сред най-разпространените и ефективни методи за организиране на данни [7]. Това довежда до множество преразглеждания на дефиницията и добавяне на оптимизации. Сред

[2] Ако операциите се извършват от множество нишки, използваният метод за синхронизация може да усложни основните операции и да увеличи броя на достъпени листа.

най-често срещаните от тях това е B+ дървото [7, 10]. Промяната, която се въвежда е в съхраняването на данните на потребителя е, че вместо всеки възел да има множество наредени двойки „ключ-стойност”, а листата да не носят никаква информация, наредените двойки се преместват в листата, а възлите с височина повече от 0 съхраняват единствено ключовете. Възлите, притежаващи единствено ключове без асоциирани директно данни, имат за цел единствено да насочат операцията за търсене към листото, използват се като индекс. Сравнение между оригиналната структура за B-дърво и оптимизирания вариант - B+ дърво, е изложено на фиг. 2.5.

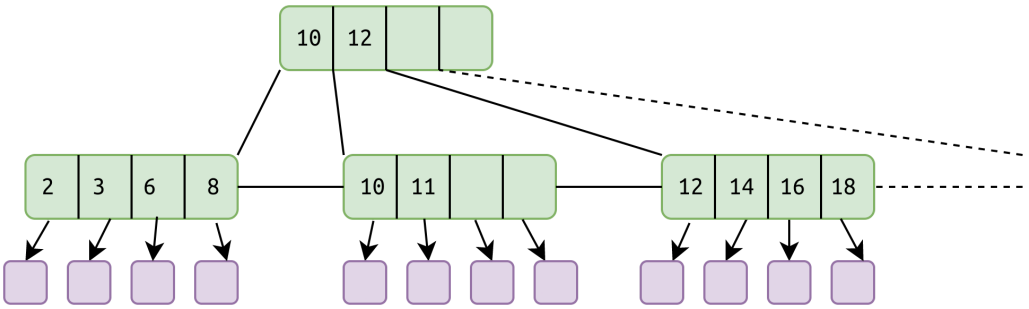


Фиг. 1.5 - Сравнение между структурата на B+ и традиционно B-дърво

Видимо е, че за разлика от традиционното B-дърво, в B+ се обособяват два вида възли - такива, които съхраняват единственото насочващи ключове и разклонения към

поддървета, т.нар *вътрешни* или *разклоняващи* възли, и такива, които съхраняват същинските наредени двойки.

Запазването на насочващи ключове в разклоняващите възли на дървото води до дуплицирано съхраняване на някои от ключовете. Въпреки това поради типичните типове данни, които се използват като ключове в бази данни, допълнителното пространство, което се заема може да бъде пренебрегнато. Друга причина, поради която B+ дърветата са се наложили като ~~*de facto*~~ стандартен~~ния~~ подход при реализацията на тази структура от данни се основава на значително по-лесния процес на изтриване на елемент. Това се дължи на факта, че за разлика от B-дърветата, B+ трябва да се грижи за изтриване единственото от листа, без възможност да се премахва елемент от разклоняващ възел. Следващо предимство на B+ вариацията на B-дърво е лесната реализация на итератор, който да обхожда всички активни[3] записи в структурата - единственото допълнително изискване е наличието на друга често срещана добавка, а именно връзки между съседни *листа* [7]. Това е показано на фиг. 2.6. Сложността на стандартните операции “вмъкване”, “търсене” и “изтриване” остава $O(logn)$ дори в най-лош случай.



Фиг. 1.6 - B+ дърво със свързани съседни листа

1.3.4. Сравнение между използването на хеш-таблица и B-дърво в контекста на СУБД

Съществува съревнование между хеш-таблиците и B-дърветата за основна структура, която да се реализира от СУБД. Основен аргумент в полза на хеш-таблиците е изпълнението на единствен достъп до хард-диска, тъй като това следва да доведе до

[3] Записи, които имат ключ с асоциирани данни.

по-малка цена за I/O операции спрямо B-дърветата, при които е необходимо да се обходи цял път от корена до крайно листо.

B-дърветата лесно настигат хеш-таблиците по ефикасност, когато броят на разклоненията е от порядъка на десетки и стотици. Лесно може да се изчисли, че при възли с размер няколко килобайта и записи с размер десетки байтове, повече от 99% от всички възли са листа. Това гарантира кратки пътища от корена до листата. В допълнение, всяка модерна СУБД съставлява в себе си кеш за страници[4], позволяващ буфериране на I/O операциите. В по-голямата част от случаите този кеш е достатъчно обемен, че да побере всички разклоняващи възли едновременно, което практически гарантира, че при обхождане на дървото от корена до крайно листо се извършва единствена I/O операция за прочит[5].

Освен че, B-дърветата са напълно конкурентноспособни на хеш-таблиците, когато става дума за цена на I/O операции, те имат и ясно изразени предимства. Сред най-характерните от тях са създаване на индекс (bulk loading), вмъкването на множество записи едновременно (bulk insertion), прилагане на предикат върху интервал, сортирано извличане и др.

Поради горепосочените причини структурата B-дърво е избрана за основа на текущата разработка.

1.3.5. Преглед на съществуващи СУБД

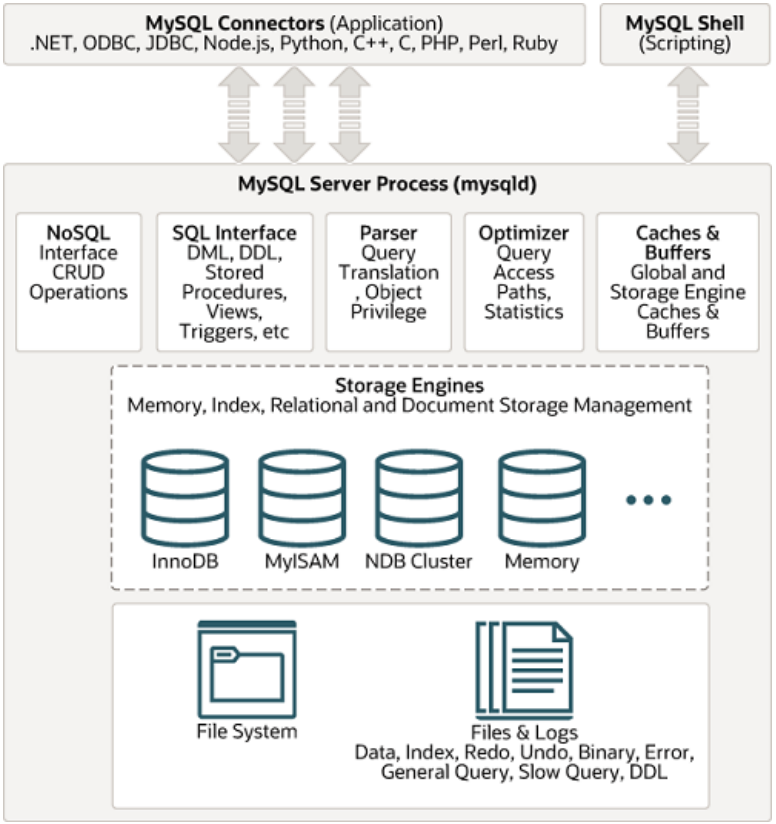
MySQL

MySQL е сред най-разпространените СУБД в света. Използвана е от най-големите компании, разработващи решения и продукти свързани с ИКТ - Tesla, Google, Facebook и др. [22]. MySQL реализира РБД. Архитектурата на MySQL СУБД е представена на фиг. 1.7.

MySQL е изградено от три компонента: потребителска апликация, конзола и сървърен процес. СУБД се използва или посредством конзолния интефейс, или посредством апликация, реализирана на език, който може да взаимодейства с MySQL, или чрез конзолата. И в двата случая се използва езикът SQL [40], който предоставя структуриран начин за написване на заявки. Подадените заявки биват обработени от сървърния daemon. В него се намира основната логика на СУБД – синтактичен

[4] Кешът за страници е разгледан по-детайлно в 3.1.
[5] Възможно е и да не се извърши нито една, ако крайното листо също се намира в кеша.

анализатор на заявки, оптимизатор, управител на буферите, кеш за страници, реализация на операции и пр.



Фиг. 1.7 Архитектура на MySQL

Най-важната част от СУБД е механизмът за съхранение (на англ. Storage engine). Това е компонента от сървъра, който извършва модификации върху съхраняваните данни на ниво физически сървър. MySQL има модулярен дизайн, позволяващ подмяна на използваният механизъм – InnoDB, MyISAM, CSV, Memory, между които най-разпространен е InnoDB. Поддържани са множество функционалности:

- Езикът за модификация на данните (ЕМД) следва ACID модела – транзакциите поддържат записване, връщане назад и възстановяване на потребителските данни;
- Външни ключове – при актуализиране и изтриване, данните се проверяват, за да се гарантира, че не настъпват несъответствия в свързаните таблици;
- Данните се съхраняват на диска по начин, позволяващ оптимизирано търсене по първичния ключ, намалявайки I/O латенцията;

Основните структури, използвани от MySQL са:

- Кеш – Използва се кеширане на информация от таблици и индекси. Минимизира извършение I/O операциите, позволявайки достъп до често използвани данни директно от паметта. Разделя се на страници и ги съхранява под формата на свързан списък. Най-скоро използваните страници стоят в началото на кеша, а най-отодавна използваните - отзад. При премахване на страница , се използва се вариант на LRU алгоритъма.
- Индекс – Физическата репрезентация на таблица. Осъществява се посредством модифициран вариант на B-дърво.

Redis

Redis е сред най-популяраните НРБД в днешно време. Тя е дистрибутирана СУБД и съхранява наредени двойки „ключ-стойност". Често бива възприета като абстракция на хеш-таблица. Всички съхранявани ключове са от от типа низ. Особеност на системата е, че асоциираните с ключовете стойности се могат да бъдат и по-сложни типове данни – списъци, хеш-таблици, множества.

Redis не цели да реализира постоянно съхранение на информация. Въпреки това, съществуват механизми, чрез които това да бъде осъществено: append-only файл и виртуална памет.

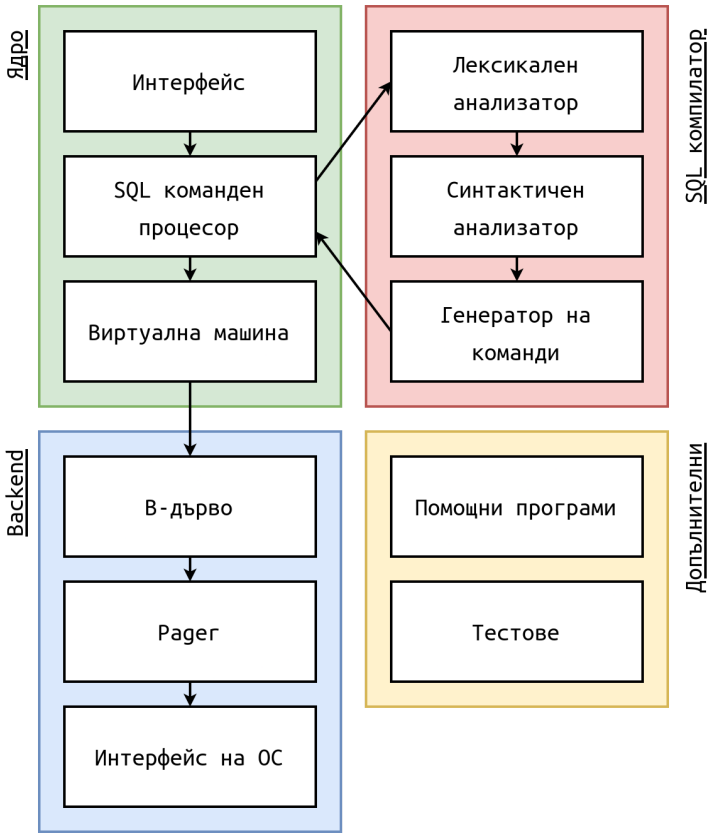
За разлика от MySQL, Redis не се нуждае от допълнителен кеш, тъй като не извършва I/O операции към и от диска. За съхранение на информацията се използва хеш-таблица, а не дървовидна структура.

SQLite

SQLite е имплементация на функционална SQL БД. Тя е сред най-често използваните РСУБД. За разлика от традиционните SQL продукти като MySQL, Oracle, PostgreSQL или SQL Server, основните употреби на SQLite са свързани с Internet of Things (IoT) решения, уебсайтове с ниско до средно потребление или анализ на данни. Тази РСУБД не е подходящ при клиент-сървър модел с големи обеми от данни и високо ниво на конкурентност.

На фиг. 1.8 е илюстрирана архитектурата на SQLite. Тя се състои от няколко компонента, които взаимодействат помежду си, за да осигурят пълната функционалност на БД:

- SQL компилаторът анализира подадените заявки, генерирайки команди, които да бъдат изпълнени от виртуалната машина, която е част от ядрото.
- Ядрото изпълнява командите, които са част от въведената заявка.
- В компонента backend е имплементирата основната логика, свързана със съхраняване на данните.
- Допълнително са предоставени помощни програми и тестове, които да верифицират коректното изпълнение на операциите.



Фиг. 1.8 Архитектура на SQLite

Глава 2. Проектиране на система за управление на бази данни

2.1. Функционални изисквания

- Изграждане на система за съхранение на данните:
  - имплементация на pager модул, който да управлява достъпа до физическата памет на устройството.

    - имплементация на структурата B-дърво, заедно с основните операции, свързани с нея.
- Разширяване на реализацията на B-дърво, така че да могат да бъдат изпълнявани множество операции в един и същи момент, без да се нарушава логическата и физическата цялост на БД, посредством метода latch coupling.
- Разширяване на реализацията на B-дърво чрез вектор на индиректорност (от англ. indirection vector), с цел поддръжка на записи с променлива дължина.
- Възможност за изпълнение на операции върху множество записи едновременно:
    - групово вмъкване (от англ. bulk insertion) и групово зареждане (от англ. bulk loading)
    - групово търсене, филтриране на записи
    - групово премахване (от англ. bulk remove)
- Конфигуриране на предоставените функционалности.
    - Уточняване на типовете данни, които биват съхранявани, стойности на основните структури (размер на страницата, размер на кеша и др.).

2.2. Подбор на технологии

Изборът на технологии е от съществено значение при разработването на всеки технологичен продукт. Това е особено вярно за системи, които имат завишени изисквания към бързодействие, ефикасност и оптимална употреба на ресурсите.

2.2.1. Език за програмиране

Основните езици за програмиране, които се използват при реализацията на системи, целящи производителност са C и C++ [22, 23, 24]. В допълнение като основен техен конкурент се изявява и модерният език Rust [21].

През последните повече от пет десетилетия езикът C се установил като фундаментален в областта на компютърните науки. Използван е като основна

технология в редица успешни продукти, и революционни иновации - UNIX, Linux, Windows, git и др. Той е фаворит при реализацията на embedded решения и е използван като междинна технология при комбиниране на няколко различни (Foreign function interface).

C++ възниква в средата на 80-те години на миналия век като подобрение на езикът C. Първоначално основната разлика между двете тях е поддръжката на обектно-ориентирано програмиране (ООП) и наличието на по-богата стандартна библиотека, но постепенно се появяват все повече програмни парадигми и нови компоненти в инструментариума и.

Rust е модерена технология, разработвана основно от Mozilla, която получава все повече приемственост и разпространение в критични системи. Целящ да бъде конкурентоспособен на C и C++, Rust съдържа най-важните им функционалности, като същевременно решава редица проблеми свързани с разработването на софтуер, използвайки C и C++ - грешки при сегментиране, управление на притежанието, const-correctness и редица други.

Избраната технология е C++, поради множеството свободно достъпни библиотеки, поддържани активно и модерните подобрения с последните стадарти - C++17 и C++20.

2.2.2. Building система

Използвана система за компилиране на проекта е комбинация между CMake и Ninja. CMake е генератор за системи за компилиране. Използва описание на файловете в проекта, за да предостави конфигурация за Ninja, което е програмата, която използва C++ компилатора, за да състави файл с двоичен код.

2.2.3. Използвани библиотеки

Проектът използва редица FOSS (Free and open-source software) билбиотеки, за да реализира дадена функционалност. Те включват:

- Libnop [16] - Ефикасно сериализиране на обекти към бинарен формат
- Catch2 [17] - Съставяне на unit-тестове
- Google Benchmark [18] - Съставяне на анализи на бързодействието на функционалности

- CppCoro [19] - Предоставя множество примитиви за работа със C++ coroutines.
- Fmt [20] - Модерно форматиране на низове

2.2.4. Управление на зависимостите

Вмъкването на множество библиотеки предизвиква проблем, свързан с управлението на зависимостите между различни готови компоненти.

Пример: „Библиотека А" използва „Библиотека Б" 0.1, а „Библиотека В" използва „Библиотека Б" 0.2.

За да се избегне ръчното менажиране на различните версии, е разработен продукта Conan [25]. Задава се файл - conanfile.txt, в който са описани различните използвани библиотеки и техните версии.

2.2.5. Среда за разработка

По време на разработката на дипломния проект е използван редакторът Vim заедно с допълнителни plugins, които осигуряват допълнителни функционалности - синтактичен и семантичен анализ, оцветяване на ключовите думи, интеграция със системата за контрол на версиите и др.

2.2.6. Система на версиите

За управлението на версиите е използвана системата git [41], като реализацията се помещава в публично хранилище в облачната платформа Github. Употребена е система за разклонения, според която съществува един основен клон, а всяка нова функционалност се отделя в нов, който в последствие бива прегледан (code review) и след това смесен в основния. При наличие на коментари и забележки, те първо биват взети предвид, докато съответната промяна (patch) не бъде одобрена от ревюиращия.

При всяка внасяне на промяна в основното хранилище бива задействана Continuous Integration (CI) система, която компилира имплементацията и изпълнява unit-тестовете. В съответствие от резултата на CI средата, промяната бива отбелязана като коректна или некоректна.

2.3. Дизайн на системата

Архитектурата на системата е съставена от няколко модули. Представен е кратък преглед на всеки един от тях.

Компресия

Компресиращият модул е базиран на алгоритъма кодиране на Хъфман [26]. Предоставя възможност за по-оптимизирано съхранение на потребителските данни. Към модулът е предоставен и възможност за превключване на функционалността, посредством конфигурация.

Pager

Pager, това е подсистема на СУБД, която се управлява физическото пространство, на което се съхраняват данни. Предоставя абстракция за извършване на I/O операции, на базата на страници - статично оразмерено количество байтове, върху които се извършват операциите на B-дървото. Към Pager модула влиза и кешът за страници, който позволява по-ефикасна работа с наличните страници, оптимизирайки достъпът до твърдия диск. Други функционалности, които предоставя Pager модулът, са алокатор за страници (вж. 3.1.3), in-memory съхранение на страници (ако целта е B-дървото да не се записва на постоянна памет), вътрешни за страниците операции, управляващи пространството - заделяне, освобождаване, поставяне, извличане (вж. 3.1.6).

B-дърво

B-дървото е основната структура, която се грижи за ефикасното съхраняване на данни [7, 10]. Предоставя оформление, което гарантира ефикасни операции за извличане, вмъкване и изтриване на данни. Дървовидната структура се основава на примитивите предоставени на Pager подсистемата. Част от функционалностите на B-дървото са групови операции - за вмъкване (от англ. bulk insertion), за филтриране на записите, за групово изтриване; съхраняване на структурата върху постоянна памет и др.

Сървър

Сървърът е подсистемата на СУБД, която инстанцира и управлява едно или повече B-дървета. Ако базата данни е релационна, там се извършват релациите между различните таблици (B-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.

2.3.1. Модул за съхранение на данните

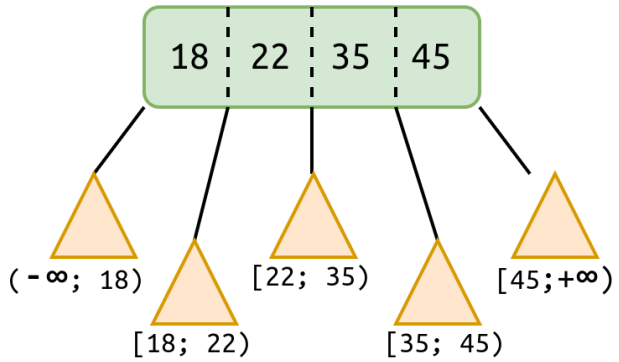
Аналогично на редица други СУБД, съхранението на данни в настоящия проект е базирано на B-дърво. Асоцииран с него е pager - абстракция на I/O операциите, които се извършват по време на изпълнение на операциите върху дървото. Основните негови примитиви са извличане и записване на подадена страница. Съдържанието на всяка една от страниците е валиден възел от B-дървото. Посредством предоставените функционалности от наличния pager, докато се извършват операциите върху дървото, могат да бъдат прочитани възли, както и да се записват променени такива.

Както беше вече споменато, стандартните операции предоставени от дървовидната структура са „вмъкване”, „извличане” и „премахване”. Следва разглеждане и анализ на основните алгоритми и методи, използвани при тях.

Извличане на данни от B-дърво

Оформлението на съхраняваната информация гарантира логаритмична сложност на търсене в дървото. Основният принцип е не по-различен от този за традиционно БДТ - сравнявайки търсения ключ със стойност в корена, алгоритъмът разглежда или лявото, или дясното поддърво, ако ключът е с по-малка или по-голяма стойност респективно. Фундаменталната разлика спрямо бинарното дърво идва от множеството разклонения при всеки един възел. Поради тази причина следва да бъде открит точната връзка с дете, където ключът попада в интервала на съхраняваните от детето

стойности. Другата важна разлика[6] е че тук операцията търсене приключва при листо, а не при първото срещане с търсения ключ при вътрешните възли.



Фиг. 2.1 Примерно оформление на ключове в B-дърво

Пример: Ако търсим ключът 42 в дървото илюстрирано на фиг. 2.1., следващия възел, който бива посетен трябва да съхранява стойности, чийто интервал да съдържа 42. В конкретния случай това е разклонението между 35 и 45.

Често срещана практика при търсенето на подходящо разклонение е да се избягва линейно търсене, поради неоптималната производителност. Стандартен подход е да се употреби бинарно (двоично) търсене (БДТ), което постига $O(logn)$ сложност спрямо $O(1)$ при линейното. Съществуват случаи, при които търсене чрез интерполиране могат да доведат го по-ефикасно изпълнение, поради своята $O(loglogn)$ сложност в най-добър случай. Въпреки това, стандартен подход е да се комбинира търсене чрез интерполиране и бинарно, лимитирайки итерациите, които извършва първото, докато БТ се използва за финализиране и конкретизиране на резултата.

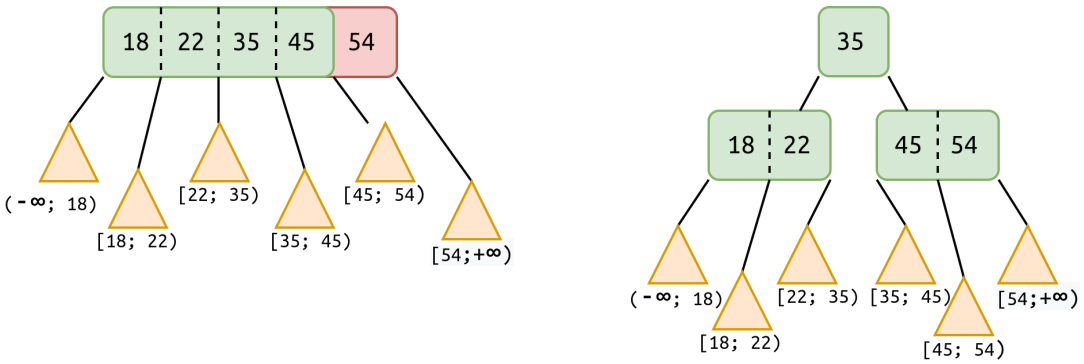
Повтаряме операцията, която води до обхождане на дървото в дълбочина, докато не достигнем листо. Когато това се случи, се извършва проверка дали запис, описан от търсения ключ е наличен в текущата структура, след което подходящ резултат може да бъде върнат.

Вмъкване на данни от B-дърво

[6] Това явление не винаги е вярно, тъй като е силно обвързано с алгоритъма, използван от операцията „премахване”.

Поради множеството свойство на B-дървото, в основата на операцията „вмъкване” стои тяхното възстановяването след извършване на промяна по съдържанието на структурата – балансиране на дървото. При въвеждане на нови стойности в даден възел, е възможно той да се препълни, т.е да има повече записи, отколкото дефиницията за B-дърво от конкретния ред позволява. В такъв случай се извършва „разделяне” и от препълненият възел се конструират нови два. Междинната стойност се добавя в родителя, ако такъв съществува, а в противен случай[7] се създава нов възел, който да заеме мястото на корена в B-дървото.

Пример: В дървото от фиг. 2.1 се вмъква ключът 54, което прави корена на B-дървото препълнен (фиг.2.2a). За да се възстановяват свойствата на структурата, се изпълнява балансиране посредством разделяне. Крайното оформление на дървото е показан на фиг.2.2b.



a) Препълнен възел b) Балансирано B-дърво

Фиг. 2.2 Разделяне на препълнен възел

Съществуват два основни метода за добавянето на нови записи в дървовидната структура: отдолу-нагоре и отгоре-надолу. И двата се основават на алгоритъма за намиране на данни, който е описан в предишна част, за да може да се локализира позицията, на която принадлежи записа. При първият метод, след като подходящото листо[8] е открито, записът се вмъква сред съхраняваните в него. След това ако балансът

[7] Ако възелът няма родител, то той е корен на B-дървото.

[8] Базирайки се на концепцията за B+ дърво, записи се съхраняват единствено в листата. За повече информация виж 1.3.3.

на дървото е нарушен, т.е ако модифицираният възел е препълнен, се извършва алгоритъма за балансиране.

При другият случай, ако добавянето е отгоре-надолу, процесът на търсене освен да локализира позицията, също така и подсигурява предварително баланса на дървото. Това се случва като, ако алгоритъмът за попадне на възел, който е пълен, той бива разделен преди да се препълни. След като това се случи обхождането продължава.

Съществува вариант, при който разделянето на възли може да се разпространи нагоре по дървото. Ако родителският възел, в който попада междинния елемент посочен от разделящата операция, бъде препълнен, върху него също трябва да се изпълни разделяне.

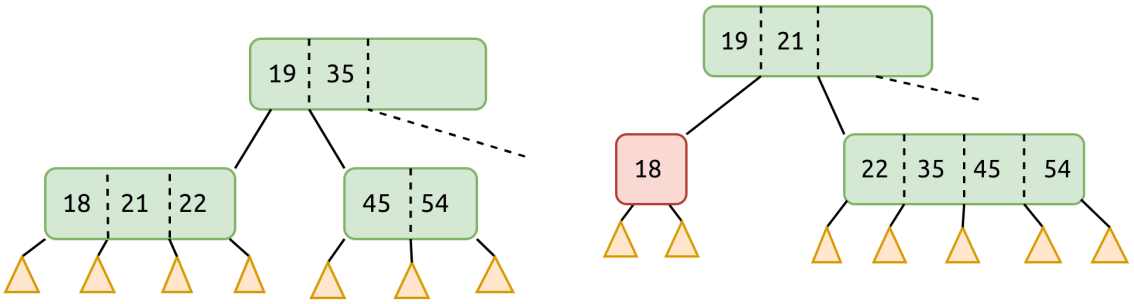
Премахване на данни от B-дърво

Аналогично на вмъкването на нова информация в дървото, нейното премахване също е задължено да възвърне баланса на структурата при завършване. Стандартният подход за това е, когато възел стане твърде празен (спрямо реда, от който е B-дървото), той да бъде смесен със съседен, ако неговият размер позволява. Това е показано на фиг. 2.3.

Алтернативно решение на смесването на възли е заемането. То може да бъде осъществено единствено когато празният възел не е единствено дете. Заемането може да се осъществи или от левия, или от десния съсед. Първо се намира междинната стойност в родителя, т.е този ключ, който задава границата на интервала за поддървото, индуцирано от небалансирания възел. Намереният ключ се премества в празното дете, докато на негово място в родителя се поставя или най-малката стойност от десния съсед, или най-голямата от левия. Пример за заемане от десния съсед и показан на фиг. 2.4.

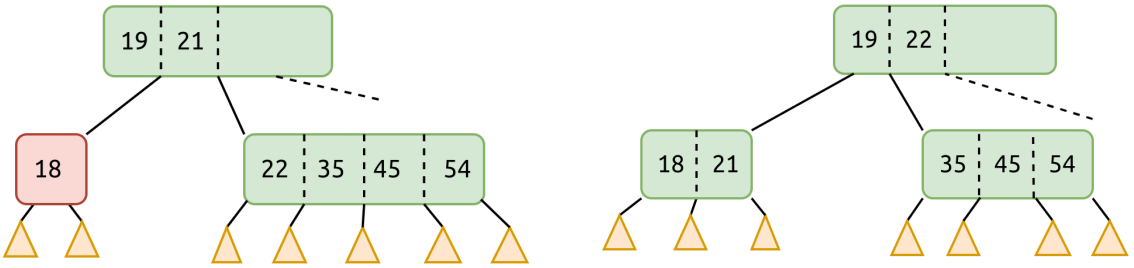
Премахването на записи от дървото е най-сложната операция сред трите стандартни. Това се дължи на няколко причини: първо, ако наредените двойки се намират във вътрешните възли на дървото (а не само в листата), изтриването принуждава да се разменят записи, така че да не се нарушат основните свойства на дървото. Второ, балансирането след изтриване е по-комплексно от това при вмъкване и трето, ограничава възможностите за реализиране на конкурентност. Поради факта, че ако дървото не се балансира след изтриване, то остава валидно за претърсване, редица

учени са разгледали как подобна структура би се реализирала и дали би била ефективна [29, 30].



a) Твърде празен възел b) Балансирано B-дърво

Фиг. 2.3 Смесване на два съседни възела



a) Твърде празен възел b) Балансирано B-дърво

Фиг. 2.4 B-дърво след заемане на ключ от съсед

Обикновено този тип изтриване се свързва с концепцията за *релаксирано* B-дърво. Това се определя от следните характеристики:

- Релаксирано B-дърво от ред l, b, е съставено от вътрешни възли, всеки от които е с размер не по-голям от b, и листа - с размер не по-голям от l. Вътрешен възел с размер равен на b и листо с размер равен на l дефинираме като пълни.
- Всеки вътрешен възел съдържащ $j - 1$ ключове, има j разклонения (j е не по-голямо от максималния размер на възела).

- Стойностите, съхранявани във всеки един от възлите, са сортирани, определяйки интервалите на възможните ключове в съответните разклонения на възела.
- Празен възел, това е възел, чийто размер е равен на 0. Позволено е възел да остане временно празен, по време на изтриване.

Гореизброените свойства по същество операцията „изтриване" се не е последвана от балансиране, тъй като при тази модификация на B-дърво няма долна граница за размера на съществуващите възли.

Възниква въпроса, доколко описаната стратегия за реализация отговаря на изискванията за ефикасност и бързодействие. Освен чисто емпиричния подход на [29, 30], [28] успешно извежда горната граница на сложността на алгоритъма, посредством метода за амортизирана асимптотична сложност [31]. Резултатите посочват, че височината на B-дърво, което изтрива ключове, посредством релаксирано балансиране може да се изрази чрез формула 2.1.

$$h_T = \log_a(\frac{m}{c}) + 1 \qquad (2.1)$$

където:

m е броят на вмъквания в дървото,

c - броят ключове в листо след разделяне ($\frac{b}{2}$),

a - броят ключове във вътрешен възел след разделяне ($\frac{l}{2}$).

Основната разлика между височината на традиционните и този вид B-дървета е, че при първите m е равен на текущия брой записи в дървото, а не на общия брой вмъквания. Доказва се, че максималният брой възли се изчислява, базирайки се на формула 2.2.

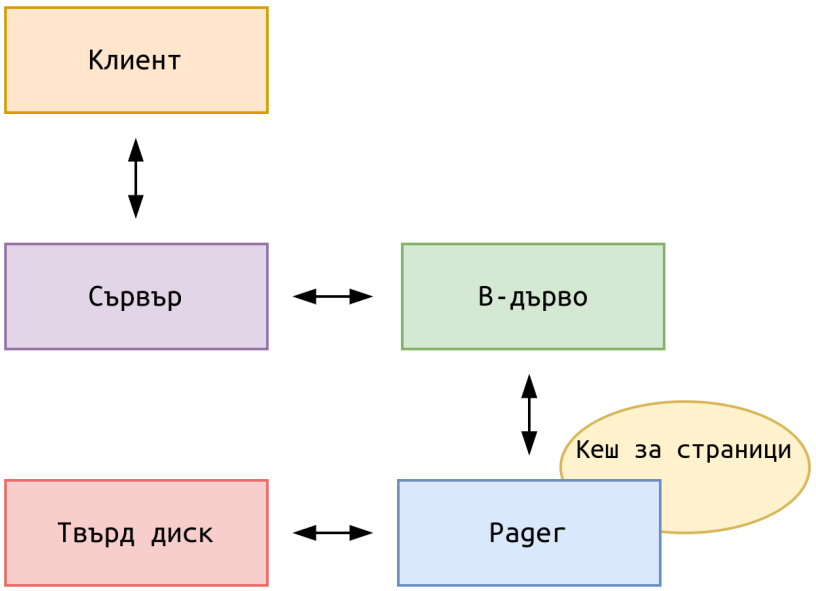
$$n_T = (\frac{m}{c})(\frac{a}{a-1}) + log_a(\frac{m}{c}) + 2 \qquad (2.2)$$

При случай, в който стойността на m е твърде голяма спрямо текущия брой на записи, дървото може да бъде изградено наново посредством ефикасни алгоритми [28, 32].

2.3.2. Комуникационен модул

Моделът клиент-сървър стои в основата на комуникационната подсистема. Той осигурява фондацията на извършваните заявки, управлявайки на високо ниво операциите, подадени от крайния потребител. Клиентът е апликацията, която се изпълнява на машината на потребителя - тя обработва подадени текстови заявки и ги формитира към REST API извиквания, които се обръщат към сървъра. Сървърът от своя страна се грижи за правилната последователност при тяхното изпълнение, използвайки директно интерфейсът предоставен от модулът за съхранение на данни - *insert, get, remove*.

2.3.3. Взаимодействие между подсистемите



Фиг. 2.5 Взаимодействието между подсистемите на СУБД

На фиг. 2.5 е илюстриран основния работен процес на системата. Посредством HTTP, клиентът се свързва към отдалечен сървър. След като се автентикира и се получи потвърждение, че дадения клиент има клиент до желана информация, той може да изпраща заявки към едно или повече B-дървета, инстанцирани на този сървър. Когато сървърът получи заявка, той я обработва и ако тя е валидна, се изпълнява съответната операция върху дървото.

То от своя страна може да се обърне към `Pager` модула, за да прочете или запише някой от своите възли. При всяка I/O операция, pager-а първо проверява в локалния си кеш за страници преди да направи обръщане към твърдия диск или друг вид постоянно хранилище. Ако желаната страница е налична, тя се използва директно от кеша, а в противен случай, се извършва I/O операция.

2.3.4. Конфигурируемост

Основна цел на дипломния проект е да предостави обща функционалност, което позволява множество различни стратегии да бъдат използвани при реализирането на СУБД.

Пример: В зависимост от данните, които се съхраняват, конкретни модули от имплементацията могат да бъдат необходими или ненужни. Ако размерът на съдържанието е статичен, заделянето и освобождаването на слотове не се извършва, което намалява цената на операциите, за разлика от ситуацията при динамично оразмерени записи.

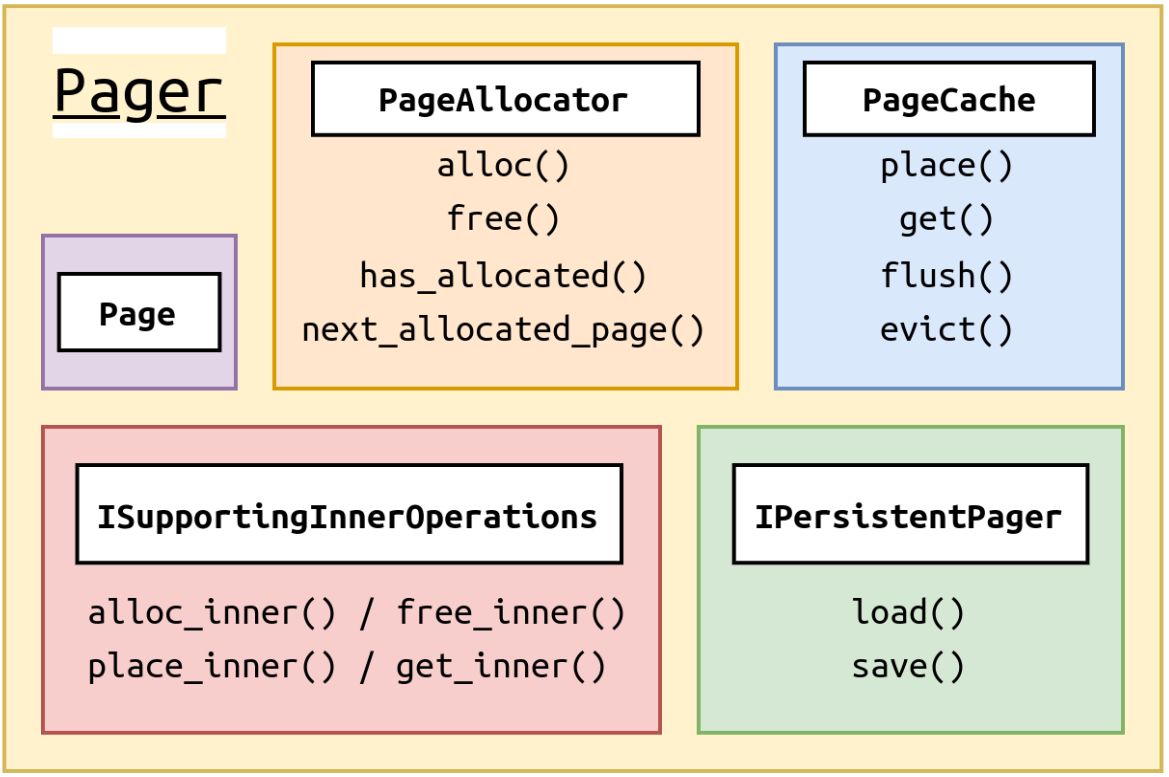
Видове опции

Тук са изброени част от поддържаните опции, които могат да бъдат използвани от библиотеката. Сред тях има взаимноизключващи се такива. Имплементацията позволява да бъде проверена валидността на конфигурацията.

- *PERSISTENT* - дали съхраняването на информация използва пространство на твърдия диск, или се намира само в RAM паметта.
- *COMPRESSION* - дали данните се компресират, посредством модула за компресиране при съхраняване на твърдия диск.
- *DYNAMIC ENTRIES* - дали размерът на съхраняваните записи е статичен или се изменят динамично.
- *PAGER ALLOCATION POLICY* - алгоритъма, за заделяне на страници.
- *PAGER EVICTION POLICY* - алгоритъма, който определя коя от разположените в кеша страниците, разположени в кеша, да бъде извадена.

Глава 3. Разработка на система на управление на бази данни

3.1. Pager



Фиг. 3.1 Основни компоненти на Pager модула

Основната роля на модула `Pager` е да управлява пространството, върху което се оперира СУБД. В подсистемата са отделени няколко компонента, които се грижат за отделна част от функционалността на модула:

- *Страница* - основната единица, върху която оперира `Pager` модула;
- *Кеш за страници* - увеличава ефикасността на I/O операциите, намалявайки латенцията, получена при обръщане на диска;
- *Алокатор за страници* - предоставя функционалност за заделяне и освобождаване на страници;
- Допълни интерфейси, които имплементиране за съответния `Pager` модул, разширяват поддържаната функционалност: *вътрешни за страницата операции* и *постоянност*.

Описаните компоненти са илюстрирани на фиг. 3.1. Те биват разгледани по-подробно в следващите секции.

Модулът се достъпва посредством клас, наследяващ `GenericPager`. Той въвежда абстракция над примитивите за достъп, които изпълняват I/O операции, осигурявайки им максимална ефикасност. Неговата дефиния е представена на фиг. 3.2. Функциите, които изисква от своите наследници са:

- `place()` - поставяне на данни под формата на `Page` на дадена позиция;
- `get()` - извличане на данни под формата на `Page` от дадена позиция;
- `alloc()` - заделяне на място с размер големината на Page върху диска;
- `free()` - освобождаване на място с размер големината на `Page`, определено от позиция.

```
template<typename AllocatorPolicy = FreeListAllocator,
         typename CacheEvictionPolicy = LRUCache>
class GenericPager {
     friend AllocatorPolicy;

public:
     template<typename... AllocatorArgs>
     constexpr explicit GenericPager(
     std::size_t limit_num_pages = PAGECACHE_SIZE / PAGE_SIZE,
         AllocatorArgs &&...allocator_args)
       : m_allocator{limit_num_pages,
                     std::forward<AllocatorArgs>(allocator_args)...},
         m_cache{limit_num_pages} {}

     GenericPager(const GenericPager &) = default;
     GenericPager &operator=(const GenericPager &) = default;

     virtual ~GenericPager() noexcept = default;

     auto operator<=>(const GenericPager &) const noexcept = default;

     /// Allocation API
     [[nodiscard]] virtual Position alloc() = 0;
     virtual void free(Position pos) = 0;

     /// Page operations API
```

```
    [[nodiscard]] virtual Page get(const Position pos) = 0;
    virtual void place(const Position pos, Page &&page) = 0;

    /// Properties
    [[nodiscard]] virtual const AllocatorPolicy &
     allocator() const noexcept {
         std::scoped_lock<std::mutex> _guard{m_mutex};
         return m_allocator;
    }

    [[nodiscard]] virtual const PageCache<CacheEvictionPolicy> &
     cache() const noexcept {
         std::scoped_lock<std::mutex> _guard{m_mutex};
         return m_cache;
    }

protected:
    AllocatorPolicy m_allocator;
    PageCache<CacheEvictionPolicy> m_cache;
    mutable std::mutex m_mutex;
};
```

Фиг. 3.2 Дефиниция на GenericPager

Друго важно свойство на `GenericPager` е, че е шаблон. Като негови параметри се посочват стратегиите, които да бъдат използвани съответно за заделяне и кеширане на страници.

Също така, тъй като се очаква инстанциите на `шаблона` да работи в многонишкова среда, като поле е предоставен и `std::mutex` [1], който да предпазва от състояние на недетерминистичен достъп до данни (от англ. data race).

В текущата имплементация са предоставени две реализации на наследници на `GenericPager`: `Pager` и `InMemoryPager`. И двата предоставят стандартните примитиви заложени в родителския клас, но и ги разширяват според конкретната си насоченост. `Pager` реализира функционалност за запазване и зареждане на състоянието (`IPersistentPager`), както и възможност за опериране във отделните страници (`ISupportingInnerOperations`). Това го прави подходящ за употреби като `Pager` на инстанция на постоянно B-дърво (вж. 3.2) и `Pager` на вектор за индиректност (вж. 3.2.3). `InMemoryPager` от друга страна цели да бъде използван

при инстанции, които се намират изцяло в RAM паметта на машината и не трябва да бъдат запазвани при рестартиране. Негова уникална характеристика е нестандартния `PageCache` - това е инстанция на `NeverEvictCache` класа, който никога не премахва принудително страници.

По-подробен преглед на имплементацията на шаблона `Pager` може да бъде открита в 3.1.2.

3.1.1. Управление на страниците

Страница

Пространството, заето от базата данни е разделено на страници. Те са основната единица, върху която оперира `Pager` модула. Техният размер се задава статично по време на компилация, посредством стойност посочена в конфигурацията на базата данни – `EugeneConfig::PAGE_SIZE` (вж. 3.3.3). Стандартните стойности са степени на 2 - 4КБ, 16КБ или дори 1 МБ. Когато са заредени в паметта, страниците се репрезентират като `std::array` [1] (фиг. 3.3).

```
using Page = std::array<std::uint8_t, PAGE_SIZE>;
```

Фиг. 3.3 Енумератор за тип на страницата

В текущата реализация размерът на възлите в модула на B-дървото съвпада с размера на страницата в `Pager` модула. Тъй като изчисляването на броя записи във възли на дървото се базира на този размер, той е от особено значение за бързодействието на операциите върху дървото - по-голям размер на възлите на дървото увеличава броя разклонения на всяко ниво от дървото, което от своя страна води до малка дълбочина на листата и минимизира броя I/O операции, които се извършват при обхождане.

Съдържанието, съхранявано във всяка една от страниците се различава, някои описват възли от B-дърво, а други слотове (вж. 3.2.3). Типът на страницата се определя от първия байт в нея, като той съвпада с един от стойностите на енумератора `PageType` - или `Node`, или `Slots` (фиг. 3.3).

```
enum class PageType : uint8_t { Node, Slots };
```

Фиг. 3.4 Енумератор за тип на страницата

Независимо от съдържанието, за записването на данни в страница се използва бинарно сериализиране на C++ обекти [16, 33]. Това е осъществено посредством библиотека на Google - Libnop, която предоставя тази функционалност за произволен тип данни.

Всяка страница има позиция, описана от целочисления тип `Position`. Тя се използва като уникален идентификатор на всяка от страниците на диска.

3.1.2. Основни примитиви в Pager

Основната логика, заложена в модула `Pager`, е имплементирана в едноименния шаблон `Pager`. Освен данните наследени от `GenericPager`, се съхранява и уникален идентификатор на инстанцията, както и файлов поток (`std::fstream`), който осигурява достъп до постоянната памет на машината.

При извикване на някоя от функциите, част от публичния интерфейс на шаблона, бива заключен `m_mutex`, което гарантира коректност на изпълнението на операции в многонишкова среда. Частните функции разчитат, че се изпълняват в еднонишкова среда, т.е, че `m_mutex` вече е бил заключен от някоя от функциите, която ги извиква.

Поставяне на страница

На фиг. 3.4 е показана имплементацията на функционалността свързана с запазване (поставяне) на страница.

```
void write(const Page &page, Position pos) {
     if (!at_page_boundary(pos))
          throw BadWrite{};

     m_disk.seekp(pos);
     m_disk.write(
     reinterpret_cast<const char *>(page.data()), PAGE_SIZE);
}

void __place(Position pos, Page &&page) {
     if (auto evict_res = this->m_cache.place(pos, Page(page));
```

```
                                                              evict_res)
            write(evict_res->page, evict_res->pos);
      }

void place(Position pos, Page &&page) override {
      std::scoped_lock<std::mutex> _guard{this->m_mutex};
      return __place(pos, std::move(page));
}
```

Фиг. 3.4 Реализация на Pager::place()

Имплементацията е разделена в няколко функции с цел преизползване на код, както и предпазване от мъртва хватка (от англ. deadlock). Второто е възможно, тъй като други функции от публичния интерфейс също извикват `Pager::place()`, което е предпоставка за повторно заключване на един и същ `std::mutex`, което според C++ стандарта е неопределено поведение, потенциално - мъртва хватка. Сред представените функции единствено `Pager::place()` е част от публичния API на шаблона.

В `Pager::__place()`, подадената страница се поставя в кеша. Ако поради тази операция, някоя страница бъде изгонена (от англ. evicted), тя бива записана на диска посредством `Pager::write()`. Това се дължи на write-behind свойствата на реализирания кеш (вж. 3.1.4).

Извличане на страница

На фиг. 3.5 е показана имплементацията на функционалността свързана с извличане на страница.

```
private:
[[nodiscard]] Page read(Position pos) {
      if (!at_page_boundary(pos))
            throw BadRead(fmt::format("[...]"));
      if (!this->m_allocator.has_allocated(pos))
            throw BadRead(fmt::format("[...]"));
      Page page;
      m_disk.seekp(pos);
      m_disk.read(reinterpret_cast<char *>(&page), PAGE_SIZE);
      return page;
}
```

```
Page __get(Position pos) {
      if (auto p = this->m_cache.get(pos); p)
            return p->get();
      Page p = read(pos);
      if (auto evict_res = this->m_cache.place(pos, Page(p));
                                                        evict_res)
            write(evict_res->page, evict_res->pos);
      return p;
}

public:
[[nodiscard]] Page get(const Position pos) override {
      std::scoped_lock<std::mutex> _guard{this->m_mutex};
      return __get(pos);
}
```

Фиг. 3.5 Реализация на Pager::get()

Аналогично на имплементацията на `Pager::place()`, `Pager::get()` е разделена на няколко отделни функции, така че да могат да бъдат преизползвани отделните операции от други функции и едновременно с това да се избегне мъртвата хватка. Сред представените функции единствено `Pager::get()` е част от публичния API на шаблона.

При извличане преди обръщане към диска, първо се извършва прочит от кеша. Ако търсената страница не е налична там, тя бива прочетена от диска посредством `Pager::read()`, след това поставена в кеша и едва тогава върната на първоначалния извикващ функцията.

3.1.3. Алокатор за страници

Състоянието на страниците в хода на използване на системата се изменя динамично – при премахване на възел от дървото, страница бива освободена, а при създаване на нов възел – заета. Поради тази причина е необходим механизъм за управление на използваните страници, който да предоставя примитиви за промяна на тяхното състояние. Това е осъществено чрез стратегията `AllocatorPolicy`, която е

подадена като шаблонен параметър на типа `Pager`. Съхраняван е като вътрешен член. Посредством него са реализирани другите две задължителни функции от родителския `GenericPager`: `alloc()` и `free()`.

За да се избере типа на алокатора, който да бъде използван се използва `EugeneConfig::PageAllocatorPolicy`. Наготово са предоставени две имплементации – `StackSpaceAllocator` и `FreeListAllocator`. Първият е изцяло базиран на структурата стак [15, 27] и поддържа единствено заделяне на нови страници. Основно предимство е простия метод на работа, но въпреки това е непрактичен при реални продукти и решения. Другият, `FreeListAllocator` използва често срещан метод за реализиране на динамични алокатори, базирайки се на списък със свободните елементи [42]. Неговата имплементация е представена на фиг. 3.6.

```
class FreeListAllocator {
      std::vector<Position> m_freelist;
      std::size_t m_next_page{0};
      std::size_t m_limit_num_pages;
      mutable std::mutex m_mutex;

      NOP_STRUCTURE(FreeListAllocator,
                      m_freelist, m_next_page, m_limit_num_pages);
public:
      explicit FreeListAllocator(
                  std::size_t limit_num_pages = DEFAULT_NUM_PAGES)
         : m_limit_num_pages{limit_num_pages} {}

      FreeListAllocator(const FreeListAllocator &) = default;
      FreeListAllocator &operator=(const FreeListAllocator &) =
                                                          default;

      [[nodiscard]] Position alloc() { [...] }

      void free(const Position pos) { [...] }

      [[nodiscard]] bool
     has_allocated(const Position pos) const { [...] }

      [[nodiscard]] cppcoro::generator<Position>
```

```
      next_allocated_page() const noexcept { [...] }

      // Properties...
};
```

Фиг. 3.5 Реализация на FreeListAllocator

Публичния интерфейс на алокатора е съставен освен от стандартните `alloc()/free()` функции, но и с проверка дали дадена страница е била вече заделена - `has_allocated()`, както и с итератор върху заделените страници - `next_allocated_page()`. Реализацията на итераторът е базирана на библиотеката Cppcoro и C++20 функционалността coroutines [19].

Използвайки макрото `NOP_STRUCTURE`, класът сигнализира на библиотеката за сериализиране Libnop [16] които от полета трябва да бъдат обработени.

Основния механизъм посредством, който се извършват операциите по заделяне и освобождаване е базиран на списък със свободни страници (`m_freelist`), ограничени от горна граница (`m_next_page`). Реализацията на операцията по заделяне е представена на фиг. 3.6.

```
[[nodiscard]] Position alloc() {
     std::scoped_lock<std::mutex> _guard{m_mutex};
     if (!m_freelist.empty()) {
          const Position pos = m_freelist.back();
          m_freelist.pop_back();
          return pos;
     }
     if (m_next_page >= m_limit_num_pages)
          throw BadAlloc(fmt::format("[...]"));
     return m_next_page++ * PAGE_SIZE;
}
```

Фиг. 3.6 Реализация на FreeListAllocator::alloc()

Всички функции от публичния интерфейс на алокатора са защитени посредством `std::mutex`, който позволява те да бъдат извиквани от многонишков контекст. За да се задели нова страница, първо се проверява списъка с вече освободени

такива. Ако той съдържа елементи, се връща най-малката позиция. В противен случай, бива преместена горната граница, а предишната нейна стойност се връща като заделена позиция. Алокаторът поддържа възможността за поставяне на лимит (`m_limit_num_pages`) на стойността на горната граница. При преминаване, алокацията се прекратява от хвърлено изключение `BadAlloc`.

Освобождаването на заделено пространство се извършва аналогично. То е представено на фиг. 3.7.

```
void free(const Position pos) {
     std::scoped_lock<std::mutex> _guard{m_mutex};
     if (pos % PAGE_SIZE != 0)
          throw BadPosition(pos);
     if (pos / PAGE_SIZE == m_next_page - 1) {
          --m_next_page;
          return;
     }
     const auto it = std::find_if(m_freelist.begin(),
               m_freelist.end(),
               [&](const Position curr) { return curr <= pos; }
      );
     if (it < m_freelist.end() && *it == pos)
          throw BadPosition(pos);
      m_freelist.insert(m_freelist.begin() +
                     std::distance(m_freelist.begin(), it), pos);
}
```

Фиг. 3.7 Реализация на FreeListAllocator::free()

Прави се проверка дали дадената позиция не описва страницата, която се намира непосредствено преди текущата стойност на горната граница. Ако това е така, деалокацията се осъществява като границата се измести надолу. В противен случай се намира мястото и в списъка със свободни страници, така че редът му да се запази във възходящ ред. При подаване на некоректна позиция за страница, се хвърля изключението `BadPosition`.

Друга основна функция, предоставена от алокатора е проверка дали дадена страница е била вече заделена. Нейната реализация е представена на фиг. 3.8.

```
private:
[[nodiscard]] bool __has_allocated(const Position pos) const {
      if (collection_contains(m_freelist, pos))
            return false;
      if (pos >= m_next_page * PAGE_SIZE)
            return false;
      return true;
}

public:
[[nodiscard]] bool has_allocated(const Position pos) const {
      std::scoped_lock<std::mutex> _guard{m_mutex};
      return __has_allocated(pos);
}
```

Фиг. 3.7 Реализация на FreeListAllocator::has_allocated()

Функцията `has_allocated()` проверява дали подадената позиция е отвъд текущата стойност на границата или дали е включена в списъка със освободени страници. Ако нито едно от двете условия не е изпълнено, то следва подадената позиция да е вече заделена. Тъй като, `has_allocated()` бива използвана от друга функция, част от публичния интерфейс, основната логика бива изнесена в частна функция, която не борави с инстанцията на `std::mutex`.

Последната предоставена функционалност от шаблона `FreeListAllocator`, е възможност за итериране по заделените страници, посредством `next_allocated_page()` coroutine. Тя се базира на т.нар повторно влизащи рутини (от англ. reentrant), т.е такива които връщат стойност множество пъти, в случая всяка заделена страница. Имплементацията на този API е представена на фиг. 3.8.

```
[[nodiscard]] cppcoro::generator<Position>
next_allocated_page() const noexcept {
      std::scoped_lock<std::mutex> _guard{m_mutex};
      for (Position i = 0; i < m_next_page * PAGE_SIZE;
                                                i += PAGE_SIZE)
            if (__has_allocated(i))
                  co_yield i;
}
```

Фиг. 3.8 Реализация на FreeListAllocator::next_allocated_page()

Типът на връщана стойност на функцията е генератор, част от CppCoro библиотеката [19], което позволява да бъде използван като итератор, т.е обект с `begin()` и `end()`. Функцията обхожда всички страници преди да достигне до горната граница, проверявайки за всяка една от тях дали е вече заделена. Биват върнати само позиции, които отговарят на изискването.

3.1.4. Кеш за страници

Обръщане към твърдия диск е скъпа операция, която се избягва максимално. Това се постига благодарение на кешът за страници - класът `PageCache`, който съхранява в RAM паметта на машината, последните достъпени страници. Неговият размер, подобно на страниците, е степен на 2 и се посочва през конфигурацията на базата данни - `EugeneConfig::PAGE_CACHE_SIZE`. Изисква се този размер да е число, кратно на размерът на използваната страница. Дефиницията на класа е представена на фиг. 3.9.

```
template<typename Policy>
class PageCache {
      friend Policy;

      [[nodiscard]] constexpr auto evict() {
          return Policy::evict(*this);
       }

public:
      constexpr explicit PageCache(
                  std::size_t limit = PAGECACHE_SIZE / PAGE_SIZE)
          : m_limit{limit > 0 ? limit :
                  std::numeric_limits<decltype(m_limit)>::max()} {}

      [[nodiscard]] optional_ref<Page> get(Position pos) { [...] }

      [[nodiscard]] constexpr CacheEvictionResult
       place(Position pos, Page &&page) { [...] }

      [[nodiscard]] cppcoro::generator<CacheEvictionResult>
```

```
      flush() { [...] }

private:
      const std::size_t m_limit;
      std::unordered_map<Position, CacheEntry> m_index;
      std::list<Position> m_tracker;
      mutable std::mutex m_mutex;
};
```

Фиг. 3.9 Дефиниция на класа PageCache

Съществуват два основни проблема при реализацията на кеш: как се определя коя страница да бъде премахната, когато мястото се изчерпи [35] и в какъв момент се опресняват данните, записани на диска [34].

Относно първия проблем класът управлява няколко колекции. Полето `m_index` е хеш-таблица, която асоциира позиция на страница със съответен кеш запис (фиг. 3.10). Всеки кеш запис съдържа същинската информация за страницата, заедно с флаг за замърсяване - дали съдържанието е било изменяно, и итератор към елемент от `std::list`. Поради свойствата на хеш-таблиците (вж. 1.3.1), имайки позицията на страница, достъпът до съдържанието на страницата, която се намира на тази позиция е константно, т.е оптимално. В допълнение се поддържа и свързан списък с позиции (`m_tracker`). Той бива променян при всеки достъп до някоя от страниците, като поставя последнодостъпения в началото на списъка. Това позволява да се проследи кои страници са били последно достъпени.

```
struct CacheEntry {
      Page m_page;
      std::list<Position>::const_iterator m_cit;
      bool m_dirty;
};
```

Фиг. 3.10 Дефиниция на кеш запис

Класът `PageCache` е шаблонен спрямо подаден policy тип [37], който определя механизма за гонене на страници от кеша (от англ. eviction). Това се случва, когато кешът бъде запълнен и за да може да бъде успешно обработена някоя операция, трябва да бъде освободено място. Тази междинна операция определя важна характеристика на кеша - метода на опресняване. Текущата имплементация използва т.нар write-behind, т.е до момента, в който дадената страница не е премахната от кеша, съдържанието на диска не се опреснява независимо от настъпилите промени. Това е и решението на горепосочения втори проблем. Pager класът в модула използва стратегия за гонене на страници, базирана на метода за използваната най-отдавна (от англ. least-recently used) (LRU) [35, 36]. Реализацията на тази стратегия е представена на фиг. 3.11 и използва предоставени от класа полета `m_tracker` и `m_index`.

```
struct PagePos {
      Page page;
      Position pos;
};

using CacheEvictionResult = std::optional<PagePos>;

struct LRUCache {
      [[nodiscard]] static CacheEvictionResult
       evict(PageCache<LRUCache> &cache) {
            CacheEvictionResult res;
            const Position pos = cache.m_tracker.front();
            const auto &cached = cache.m_index.at(pos);
            if (cached.m_dirty)
                  res = PagePos{.page = cached.m_page, .pos = pos};
            cache.m_index.erase(pos);
            cache.m_tracker.pop_front();
            return res;
      }
};
```

Фиг. 3.11 Дефиниция на LRUCache policy

Реализацията на LRU алгоритъма е сведена до извеждане на първия елемент от `m_tracker`. Функцията `evict()` връща стойност `std::optional`, която съдържа стойност единствено ако изведената страница трябва да бъде записана на диска, т.е е била замърсена.

Друга основна операция на кеша свързани тясно с изгонването на страница е `flush()`. Нейната реализация е представена на фиг. 3.12.

```
[[nodiscard]] cppcoro::generator<CacheEvictionResult> flush() {
      std::scoped_lock<std::mutex> _guard{m_mutex};
      while (!m_tracker.empty())
            co_yield evict();
}
```

Фиг. 3.12 Реализация на PageCache::flush()

Аналогично на `FreeListAllocator::next_allocated_page()`, `flush()` връща `cppcoro::generator`, т.е множество стойности асинхронно, които могат да бъдат итерирани. Това е необходимо, тъй като кеш policy типа, няма достъп до примитивите за запазване на страница върху диска и е необходимо да предаде тази информация на извикващия.

Достъпването на страница от кеша става на база на нейната позиция. Имплементацията е представена на фиг. 3.13.

```
[[nodiscard]] optional_ref<Page> get(Position pos) {
      std::scoped_lock<std::mutex> _guard{m_mutex};
      if (!m_index.contains(pos))
            return {};

      auto it = m_index.find(pos);
      m_tracker.splice(m_tracker.cend(), m_tracker,
                                                it->second.m_cit);
      return it->second.m_page;
}
```

Фиг. 3.13 Реализация на PageCache::get()

Тъй като операцията трябва да бъде изпълнима в многонишков контекст и е част от публичния интерфейс на класа, данните биват предпазени посредством инстанция на `std::mutex`. Проверката се извършва чрез хеш-таблицата. Ако бъде намерен такъв запис, посредством съхранявания итератор от `m_tracker`, записът бива преместен най-отпред, т.е като най-скоро достъпен.

Въвеждането на нов запис в кеша е изложено на фиг. 3.14.

```
[[nodiscard]] constexpr CacheEvictionResult
place(Position pos, Page &&page) {
     std::scoped_lock<std::mutex> _guard{m_mutex};
     CacheEvictionResult evict_res;
     const auto it = m_index.find(pos);
     if (it == m_index.cend()) {
          if (m_tracker.size() >= m_limit)
               evict_res = evict();
          m_tracker.push_back(pos);
     } else {
          m_tracker.splice(m_tracker.cend(), m_tracker,
                                              it->second.m_cit);
     }
     m_index[pos] = CacheEntry{
         .m_page = page,
         .m_cit = m_tracker.cend(),
         .m_dirty = true
     };
     return evict_res;
}
```

Фиг. 3.13 Реализация на PageCache::place()

Поставянето на страница в кеша може да протече по два различни начина в зависимост дали това е обновяване на съдържанието на страницата в кеша или е първоначално въвеждане. Ако страницата не е била налична в кеша досега, то съществува вариант кешът да се препълни, поради което може да се извика `evict()`. На това се дължи и типа на връщана стойност на функцията.

3.1.5. Постоянство

`Pager` предоставя имплементации за методите, декларирани в `IPersistentPager` интерфейса. Това са `load()` и `save()`, които позволяват съответно зареждане и запазване на `Pager` инстанция. Реализацията на първия метод е представена на фиг. 3.14.

```
void load() override {
     std::scoped_lock<std::mutex> _guard{this->m_mutex};
     const std::string pager_allocator_name =
```

```
                                fmt::format("{}-alloc", m_identifier);
    nop::Deserializer<nop::StreamReader<std::ifstream>>
                                  deserializer{pager_allocator_name};
    if (!deserializer.Read(&this->m_allocator))
        throw BadRead("deserializer failed [...]");
}
```

Фиг. 3.14 Реализация на Pager::load()

Функционалността е базирана на библиотеката за бинарно сериализиране и десериализиране Libnop [16]. Посредством поставеното макро `NOP_STRUCTURE`, биват сериализирани само посочените полета от класа. От файл с полученото форматирано име, се прочитат данните, които да бъдат преобразувани в алокатора на `Pager`. Аналогично, в обратния процес, при имплементацията на `Pager::save()`, използвайки `nop::Serializer` и `serializer.Write()` алокаторът бива запазен.

3.1.6. Вътрешни операции върху страници

Освен стандартните примитиви изисквани от `GenericPager`, `Pager` имплементира и интерфейса `ISupportingInnerOperations`. Заради това предоставя и допълнителни функции за управление на пространството вътре в самите страници (`alloc_inner()/free_inner()`), както и за поставяне/прочитане на данни вътре в страница (`place_inner()/get_inner()`).

И четирите функции разчитат на специфично оформление (от англ. layout) на съдържанието на страниците. Няма съвместимост между страници с възли и страници, върху които се правят вътрешни операции, т.е дадена инстанция на `Pager` борави или само с страници-възли, или само с страници, върху които се извършват вътрешни операции. Във всяка страница се въвежда заглавна част (от англ. header), която съдържа информация за заетостта на останалата част. В заглавната част се съдържа побитова карта, където всеки бит определя дали част от страницата е заета. Посредством константата `PAGE_ALLOC_SCALE`, бива определен мащаба на картата, или каква част от страницата се описва с един бит. По подразбиране стойността е 4B, т.е всеки 4B имат по един 1 бит, който маркира дали са използвани, или не. Стойността на `PAGE_ALLOC_SCALE` определя каква е минималната алокация, която може да бъде извършена без да се въвежда вътрешна фрагментация на страниците. При 4B мащаб, алокации под 4B биха довели до вътрешна фрагментация. На база на побитовата карта

могат да се извършват вътрешни операции. Най-простата сред предоставените е `Pager::max_bytes_inner_used()`, която изчислява колко байта са заделени за дадена страница. Имплементацията на тази функция е представена на фиг. 3.15.

```
[[nodiscard]] cppcoro::generator<std::pair<unsigned, bool>>
chunkbit_iter(const Page &p) {
      auto chunk_num = 0;
      for (auto i = PAGE_TYPE_METADATA;
           i < PAGE_TYPE_METADATA + CHUNK_MAP_SIZE;
           ++i, ++chunk_num) {
             const auto val = p.at(i);
             for (auto bit_num = 0; bit_num < CHAR_BIT; ++bit_num)
                  co_yield std::make_pair(
                      chunk_num * CHAR_BIT + bit_num,
                      val & (1 << bit_num)
                  );
      }
}
std::size_t max_bytes_inner_used() noexcept override {
      std::scoped_lock<std::mutex> _guard{this->m_mutex};
      std::size_t chunks = 0;
     for (Position page_pos :
                          this->m_allocator.next_allocated_page())
           for (const auto &[_, bitval] :
                                      chunkbit_iter(get(page_pos)))
                chunks += static_cast<std::size_t>(bitval);
      return chunks * PAGE_ALLOC_SCALE;
}
```

Фиг. 3.15 Реализация на Pager::max_bytes_inner_used()

Функцията е част от публичния интерфейс на шаблона и затова заключва инстанцията на `std::mutex`. Крайният резултат се образува като за всяка една от заделените страници (използва се итератора, предоставен от алокатора) се броят 1-ците в побитовата карта. Втората стъпка се реализира посредством итератор върху побитовата карта - `chunkbit_iter()`.

На фиг. 3.16 е представена имплементацията на функцията `alloc_inner()`, която заделя пространство в серия от последователни страници.

```
Position alloc_inner(std::size_t sz) override {
      std::scoped_lock<std::mutex> _guard{this->m_mutex};

      if (sz == 0) throw BadAlloc("[...]");
      const auto target_chunks = round_upwards(sz,
PAGE_ALLOC_SCALE);
      auto curr_chunks = 0ul;
      std::map<Position, Page> marked_pages;
      std::optional<Position> prev_page_pos;
      auto start_pos = 0ul;

      auto reset = [&] { curr_chunks = 0; marked_pages.clear(); };

      auto alloc_in_page=[&](Page &page, const Position page_pos) {
            for (auto [chunk_num, bitval] : chunkbit_iter(page)) {
                  if (bitval) {
                        reset(); continue;
                  }
                  if (curr_chunks == 0)
                        start_pos = chunk_to_position(page_pos,
                                                      chunk_num);
                  if (++curr_chunks >= target_chunks)
                        break;
            }

            if (curr_chunks > 0)
                  marked_pages.emplace(page_pos, std::move(page));
      };

      for (Position page_pos :
                        this->m_allocator.next_allocated_page()) {
            auto page = __get(page_pos);
            if (page.front() !=
                        static_cast<uint8_t>(PageType::Slots)) {
                  reset();
                  continue;
            }
            if (prev_page_pos
             && prev_page_pos.value() != page_pos - PAGE_SIZE)
                  reset();

            alloc_in_page(page, page_pos);
```

```
            if (curr_chunks >= target_chunks)
                  break;
      }


      while (curr_chunks < target_chunks) {
            auto new_page = SlotPage();
            auto new_page_pos = this->m_allocator.alloc();
            alloc_in_page(new_page, new_page_pos);
            __place(new_page_pos, std::move(new_page));
      }

      for (auto &[mppos, mp] : marked_pages) {
            for (auto [chunk_num, _] : chunkbit_iter(mp)) {
                  auto chpos = chunk_to_position(mppos, chunk_num);
                  if (chpos >= start_pos && chpos < start_pos + sz)
                        chunkbit(mp, chunk_num, true);
            }
            __place(mppos, std::move(mp));
      }
      return start_pos;
}
```

Фиг. 3.16 Реализация на Pager::alloc_inner()

`alloc_inner()` е основната операция сред четирите, предоставени от `ISupportingInnerOperations`. Алгоритъмът по заделяне на пространство работи като първоначално опитва да намери достатъчно голяма дупка във вече използвани страници, итерирайки върху `m_allocator.next_allocated_page()`. За всяка една страница използва `alloc_in_page()`, където се преминава през побитовата карта на текущата страница, обновявайки състоянието на операцията - `start_pos` (начало на заделеното пространство), `prev_page_pos` (позиция на предишната страница, за да се проследи дали заделеното пространство е в поредни страници) и т.н. След като се премине през използваните страници, се проверява дали е намерено необходимото пространство. В случай, че това е така, операцията завършва. Ако това не се случи, следващата стъпка е да се заделят нови страници, докато не се достигне търсеният размер пространство. Последната стъпка от заделянето е да се обходят страниците, за които се знае, че са

част от заделеното пространство и да се *маркират*. По време на първоначалното обхождане, ако всяка част от страница, която бива временно заделена, се отблязва в побитовата карта, съществува голяма вероятност да се наложи да се наложи да бъде поправено в последствие. Поради тази причина всеки път, когато нова страница започне да се разглежда тя се добавя в хеш-таблица. Ако разглежданата част се окаже недостатъчно голяма, се използва ламбда функцията `reset()`, която поставя `start_pos` към текущата и изпразва хеш-таблицата със страници за маркиране.

Освобождаване на вътрешна част от страница се осъществява посредством реализацията показана на фиг. 3.17.

```
void free_inner(Position pos, std::size_t sz) override {
      std::scoped_lock<std::mutex> _guard{this->m_mutex};
      auto pgpos = page_pos_of(pos);
      auto target_chunks = round_upwards(sz, PAGE_ALLOC_SCALE);

      while (target_chunks > 0) {
            auto pg = __get(pgpos);
            for (auto [chunk_num, _] : chunkbit_iter(pg)) {
                  auto chpos = chunk_to_position(pgpos, chunk_num);
                  if (chpos < pos)
                        continue;
                  chunkbit(pg, chunk_num, false);
                  if (--target_chunks <= 0)
                        break;
            }
            __place(pgpos, std::move(pg));
            pgpos += PAGE_SIZE;
      }
}
```

Фиг. 3.17 Реализация на Pager::free_inner()

Алгоритъмът за освобождаване на пространство част от страница е аналогичен на ламбда функцията `alloc_in_page()` в `alloc_inner()` функцията. Една по една се извличат страниците, част от заделеното пространство. Всички стойности от побитовата карта на съответната страница, които описват позиции, попадащи в заделеното пространство, се записват като свободни.

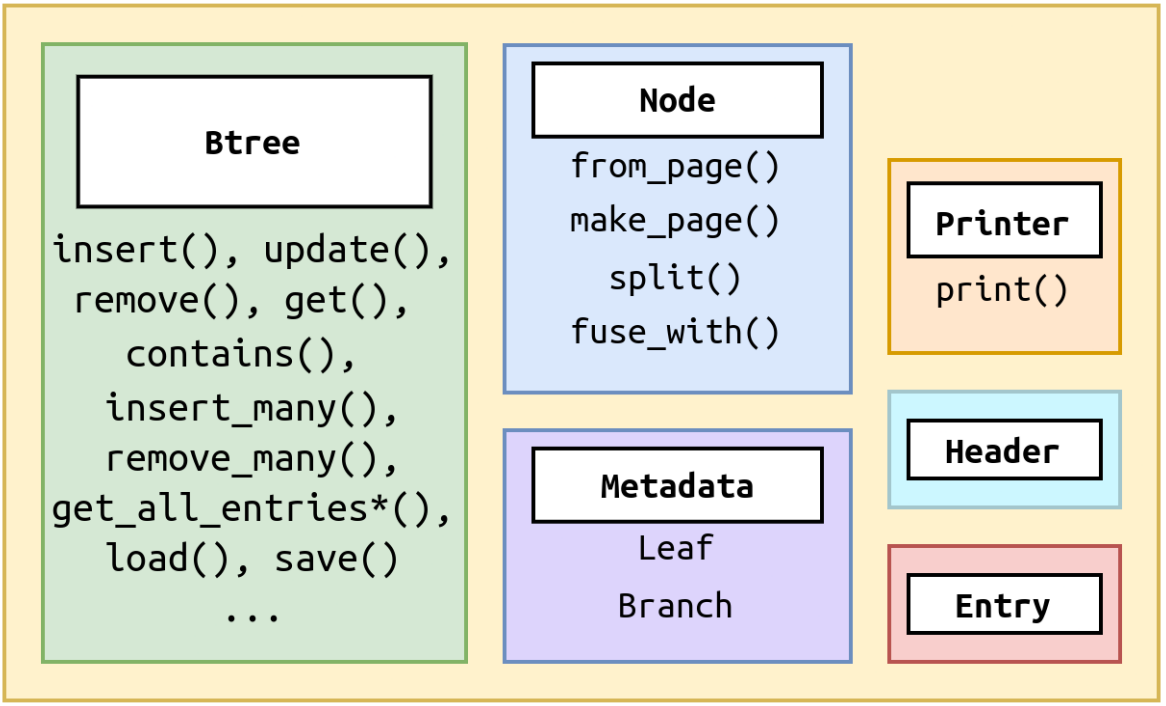
Останалите две от функциите, свързани с интерфейса ISupportingInnerOperations са свързани със съхраняване и извличане на данни от вече заделено пространство. Методите на изпълнение на двете функции са аналогични. Операцията за четене се изпълнява докато не се прочете зададения размер. Ако бъде прочетена цялата страница се извлича следващата. Аналогичен е подходът и при поставяне на данни. На фиг. 3.18 е показана имплементацията на place_inner().

```
void place_inner(Position pos, const std::vector<uint8_t> &data)
                                                        override {
     std::scoped_lock<std::mutex> _guard{this->m_mutex};
     auto start_pos = pos % PAGE_SIZE;
     auto pgpos = page_pos_of(pos);
     auto cursor = 0ul;
     while (cursor < data.size()) {
          auto page = __get(pgpos);
          if (page.front() !=
                          static_cast<uint8_t>(PageType::Slots))
               throw BadWrite(fmt::format("[...]", pgpos));
          auto limit = std::min(data.size() - cursor,
                                PAGE_SIZE - start_pos);
          std::copy_n(data.cbegin() + cursor,
                      limit,
                      page.begin() + start_pos);
          __place(pgpos, std::move(page));
          cursor += limit;
          start_pos = PAGE_HEADER_SIZE;
          pgpos += PAGE_SIZE;
     }
}
```

Фиг. 3.18 Реализация на Pager::place_inner()

Подобно на останалите функции част от публичния интерфейс, първата стъпка е да се заключи достъпа до данните на класа, за да се избегнат непоследователности при многонишков контекст. За всяка извлечена страница се преценява колко байта могат да бъдат поставени - не повече от големината на останалите данни за записване и не повече от останалото пространство в страницата. След като бъдат записани, се обновява информацията за големината на останалите данни и позицията на страницата. Ако има още данни за поставяне, стъпките се повтарят.

3.2. B-дърво



Фиг. 3.19 Основни компоненти на B-дърво модула

Предназначението на B-дърво модула (фиг. 3.19) е да предостави механизъм за съхранение на потребителските данни. Те са организирани в наредени двойки „ключ-стойност". Тази подсистема е имплементирана като шаблон `Btree`, което е реализация на модерното схващане за B-дърво, а именно такова, в което записите се намират единствено в листата, докато останалите възли се използват за насочване на операцията по търсене към съответното листо, което съхранява желаните данни (вж. 1.3.3). Реализацията на дървовидната структура се основава на функционалностите предоставени от Pager модула. Част от публичния интерфейс, предоставен от B-дървото е съставен от[9]:

- `insert()`, `update()` - вмъква или обновява стойност на запис в дървото;
- `remove()` - премахва запис от дървото;
- `get()`, `contains()` - извлича данни свързани с даден запис;

[9] Публичните функции на класа Btree са твърде много, за да бъдат изведени тук всички. За допълнителна информация относно всички функции, част от API, може да бъде разгледано 4.3.1.

- `insert_many()`, `remove_many()` - извършва модификация върху множество записи с една операция;
- `get_all_entries()`, `get_all_entries_filtered()`, `get_all_entries_in_key_range()` - извършва филтриране върху записите, съхранявани в дървото;

Освен изрично посочените, съществуват и други функции, които са предоставени на потребителя. Най-важните от тях биват разгледани подробно в следващите части.

3.2.1. Модели и представяне

Възли

В дървото се срещат два вида възли: разклоняващи възли и листа. Първите съдържат позициите на своите деца (разклонения) и ключове, които използват за сравнение по време на обхождане. Листата съхраняват множество наредени двойки. Специфичните данни свързани с конкретния вид възел са изведени в отделни типове - `Branch` и `Leaf`, а в съхраняващия клас `Node`, се използва конструкцията *variant* [1]. Това позволява да бъдат съхранявани данни, които могат да бъдат от различни типове в хода на своя живот. За да могат метаданните да се сериализират, се използва `nop::Variant` - имплементация, която поддържа операциите сериализиране и десериализиране.

Структурата `Branch` съдържа в себе си вектор с ключове, с които да се извършва сравнение по време на обхождане, вектор с връзки към наследниците, както и вектор, обозначаващ кои от връзките са валидни. Структурата `Leaf` е образувана от множество ключове и също толкова на брой стойности.

Освен информацията, свързана специфично с типа на възела, всеки един от тях съхранява позицията на следващия възел на същата височина и флаг, обозначаващ дали възела е корен на структурата. Извадка от дефиницията на класа `Node` е показана на фиг. 3.20.

```
template<EugeneConfig Config = Config>
class Node {
public:
      struct Branch {
```

```
            std::vector<Ref> refs;
            std::vector<Position> links;
            std::vector<LinkStatus> link_status;
            NOP_STRUCTURE(Branch, refs, links, link_status);
      };

      struct Leaf {
            std::vector<Key> keys;
            std::vector<Val> vals;
            NOP_STRUCTURE(Leaf, keys, vals);
      };

      using Metadata = nop::Variant<Branch, Leaf>;
      [...]
private:
      Metadata m_metadata{};
      bool m_is_root{false};
      nop::Optional<Position> m_next_node_pos{};

      NOP_STRUCTURE(Node, m_metadata, m_is_root, m_next_node_pos);
};
```

Фиг. 3.20 Представяне на възел в B-дърво модула

Стойността на `m_next_node_pos` може да не съдържа нищо, тъй като най-десните възли от дървото няма десен съсед. Използвайки `NOP_STRUCTURE`, типът `Node` може да бъде използван от библиотеката за бинарно сериализиране Libnop [16].

B-дърво

Класът `Btree` е компонентът, който имплементира функционалностите на структурата B-дърво. Основните полета, които съхранява са информация относно състоянието на дървото - статистики (размер, дълбочина), ред, текуща позиция на корена и пр. Описаните данни формират т.нар заглавна част (от англ. header) на структурата. Заглавната част са данните, които биват запазени върху диска, така че да позволят зареждане и запазване на инстанции върху диска.

Всяка инстанция на B-дървото бива определена от уникален идентификатор - “име” на дървото. На негова база се създават файловете, в които се съхраняват данните, както и състоянието на системата.

Фундаментална част от B-дървото е интерфейсът към “физическото пространство” на машината, т.е модулът, занимаващ се с извършване на I/O операции, а именно - Pager. Посредством него биват съхранявани възлите на дървото.

3.2.2. Операции върху възли

Възлите предоставят публичен интерфейс за преобразуване към и от страници. Това са `from_page()` и `make_page()`. Използват се при извършване на I/O операции – прочита се страница, която трябва да бъде интерпретирана като възел, или се записва възел, който за целта трябва да бъде от типа `Page`. Имплементират се посредством сериализация и десериализация. Имплементацията на `from_page()` е показана на фиг. 3.21.

```
[[nodiscard]] static Nod from_page(const Page &p) {
    if (static_cast<PageType>(p.front()) != PageType::Node)
        throw BadRead("cannot create node from page");
    nop::Deserializer<nop::BufferReader> deserializer{
                                                p.data() + 1, PAGE_SIZE - 1};
   Node node;
   if (!deserializer.Read(&node))
       throw BadRead("failed deserializing node");
   return node;
}
```

Фиг. 3.21 Имплементация на Node::from_page()

Преобразуването може да не се извърши успешно по няколко причини. Ако страницата не е отбелязана като страница-възел, то операцията се прекратява посредством изключението `BadRead`. Ако страницата е маркирана като съдържаща възел, но съдържанието ѝ не е коректно сериализиран обект, преобразуването отново завършва неуспешно, хвърляйки изключението `BadRead`.

Имплементацията на `make_page()` е схода, но се използва `nop::Serializer` и `serializer.Write()`.

Другите две функции, достъпни чрез публичния API са свързани с разпределяне на записи между няколко възела - `fuse_with()` и `split()`. Първата функция се

използва за сливане на два възела в един. Задължително е те да бъдат от един и същ тип – или листа, или разклоняващи. Конструира се нов възел, който съдържа комбинираните метаданни, подредени във възходящ ред според стойността на ключовете. Нито един от двата подадени аргументи бива променен. Основната логика в имплементацията се реализира от функцията `merge_many()`. Нейната дефиниция е представена на фиг. 3.22.

```
void merge_many(std::ranges::range auto self,
                std::ranges::range auto diff, auto fun) {
    auto self_begin = std::cbegin(self);
    auto diff_begin = std::cbegin(diff);
    while (self_begin < std::cend(self)
        && diff_begin < std::cend(diff)) {
         const auto use_self = *self_begin < *diff_begin;
         std::size_t idx = use_self
                ? std::distance(std::cbegin(self), self_begin++)
                : std::distance(std::cbegin(diff), diff_begin++);
         fun(use_self, idx);
     }
    while (diff_begin < std::cend(diff))
       fun(false, std::distance(std::cbegin(diff), diff_begin++));
    while (self_begin < std::cend(self))
        fun(true, std::distance(std::cbegin(self), self_begin++));
}
```

Фиг. 3.21 Имплементация на merge_many()

Основното предназначение на функцията merge_many() е да итерира през два наредени интервала (от англ. range) формирайки общата наредба между двата, като на всяка стъпка се извиква подадена функция върху текущо разглежданата стойност. Това е същият проблем, който се появява при реализацията на fuse_with() - ако аргументите са листа, то трябва да бъдат комбинирани двата вектора с ключове и двата вектора със стойности, но на база наредбата единствено на ключовете.

Противоположната на `fuse_with()` операция е `split()`. Тя разделя подадения възел на два - ляв и десен брат. Изчислява се точка на пречупване (от англ. pivot) вземайки предвид подадения `SplitBias`, като левия брат получава всички записи до pivot, а десния - след pivot. `SplitBias` е енумератор (фиг. 3.22), който

описва посредством кой метод да бъде изчислена точката, в която да се раздели възела. Вариантите са `LeanLeft`[10] - левият брат бива напълнен, останалите записи отиват в десния; `LeanRight` - идентично, но наобратно; `DistributeEvenly` - и двете получават по равен брой записи; `TakeLiterally` - буквалната стойност на pivot бива подадена като аргумент.

```
enum class SplitBias {
    LeanLeft,
    LeanRight,
    DistributeEvenly,
    TakeLiterally
};
```

Фиг. 3.22 Дефиниция на енумератор SplitBias

Имплементацията на split() е представена на фиг. 3.23.

```
constexpr auto split(const std::size_t max_num_records,
                     const SplitBias bias,
                     const SplitType type=SplitType::ExcludeMid) {
    Node sibling;
    Key midkey;
    const auto pivot = [max_num_records, bias, this]() {
        switch (bias) {
        break; case SplitBias::LeanLeft:
            return max_num_records - 1;
        break; case SplitBias::LeanRight:
            return std::abs(num_filled() -
                            static_cast<long>(max_num_records)) + 1;
        break; case SplitBias::DistributeEvenly:
            return num_filled() / 2;
        break; case SplitBias::TakeLiterally:
            return max_num_records;
        }
        UNREACHABLE
    }();
    if (is_branch()) {
        auto &b = branch();
```

[10] `SplitBias::LeanLeft` е подходящ, когато в серия от поредни въвеждания данните са подредени в нарастващ ред, тъй като ще намали броя на разделяния на възли.

```
        midkey = b.refs[pivot];
        sibling = {metadata_ctor<Branch>(
                        break_at_index(b.refs, pivot + 1),
                        break_at_index(b.links, pivot + 1),
                        break_at_index(b.link_status, pivot + 1)),
                    parent()};
        if (type == SplitType::ExcludeMid)
            b.refs.pop_back();// Branch nodes do not copy mid-keys
    } else {
        auto &l = leaf();
        sibling = {metadata_ctor<Leaf>(
                        break_at_index(l.keys, pivot),
                        break_at_index(l.vals, pivot)),
                    parent()};
        midkey = sibling.leaf().keys.front();
    }
    return std::make_pair<Key, Nod>(std::move(midkey),
                                            std::move(sibling));
}
```

Фиг. 3.23 Дефиниция на Node::split()

Изпълнението на операцията започва с изчисляване на pivot. След това според типа на възела биват разпределени метаданните между текущия и новия брат. При разделяне на разклоняващ възел, се подава и тип на разклоняването (SplitType), който се използва да уточни дали междинният елемент да бъде запазен в левия брат или премахнат. В края се връща като резултат междинния елемент.

3.2.3. Операции върху B-дърво

Реализираната структура B-дърво осигурява по-голямата част от публичния API, предоставен на потребителя. Това включва извличане на записи асоциирани с конкретен ключ, филтриране върху множество записи според зададена функция, вмъкване или обновяване на единичен запис или множество такива и премахване на данни. Осигурява API за запазването и зареждане на структурата, както и извличане на информация свързана със текущото състояние на дървото – валидност, статистики (брой текущи елементи, дълбочина и пр.), ред, идентификатор, текущ корен и др.

Извличане

Търсенето на запис асоцииран с ключ в дървото е най-важната операция за тази структура. Основният алгоритъм се преизползва както при вмъкване, така и при премахване на запис. Имплементацията се намира във функцията `search()` (фиг. 3.24).

```
SearchResultMark search_subtree(const Key &target_key,
                                const Nod &origin,
                                const Position origin_pos) {
    TreePath path;
    Nod curr = origin;
    Position curr_pos = origin_pos;
    std::optional<std::size_t> curr_idx_in_parent{};
    bool key_is_present = false;
    std::size_t key_expected_pos;

    while (true) {
        path.push(PosNod{
          .node_pos = curr_pos,
          .idx_in_parent = curr_idx_in_parent,
          .idx_of_key = {}});

         if (curr.is_branch()) {
              const auto &branch_node = curr.branch();
             const std::size_t index = [&] {
                 auto it = std::lower_bound(
                                        branch_node.refs.cbegin(),
                                        branch_node.refs.cend(),
                                        target_key);
                  return it - branch_node.refs.cbegin() +
                                 (it != branch_node.refs.cend() &&
                                                 *it == target_key);
              }();

             if (branch_node.link_status[index] ==
                                              LinkStatus::Inval)
                 throw BadTreeSearch("[...]");
              curr_idx_in_parent = index;
             curr_pos = branch_node.links[index];
             curr = Nod::from_page(m_pager->get(curr_pos));
        } else if (curr.is_leaf()) {
             const auto &leaf_node = curr.leaf();
```

```
            key_expected_pos = std::lower_bound(
                                        leaf_node.keys.cbegin(),
                                        leaf_node.keys.cend(),
                                target_key) - leaf_node.keys.cbegin();
            key_is_present = key_expected_pos <
                                            leaf_node.keys.size()
                && leaf_node.keys[key_expected_pos] == target_key;
            if (key_is_present)
                path.top().idx_of_key = key_expected_pos;
            break;
        } else
            throw BadTreeSearch("[...]");
    }
    if (!curr.is_leaf())
        throw BadTreeSearch("[...]");
    return SearchResultMark{
                    .node = curr,
                    .path = path,
                    .key_expected_pos = key_expected_pos,
                    .key_is_present = key_is_present};
}

SearchResultMark search(const Key &target_key) {
     return search_subtree(target_key, root(), rootpos());
}
```

Фиг. 3.24 Дефиниция на Btree::search_subtree()

Функцията `search()` е обвивка на вътрешната `search_subtree()` функция. Реализацията на последната плътно следва описанието на операцията: започвайки от корена се сканира текущия възел, в търсене на разклонение, което води към възел, индуциращ поддърво, съдържащо търсения запис. Това се повтаря докато не се достигне листо. Тогава извличане се прекратява или с успешно намерен запис или с резултат, сигнализиращ, че не съществува такъв. Крайната стойност, която бива върната след изпълнение на операцията е инстанция на структурата `SearchResultMark`. В нея е отбелязана допълнителна информацията за изпълнението на функцията - дали е намерен търсения ключ; листото, в което е направена проверката; на коя позиция е в листото; пътят, който се изминава в дървото, за да се достигне до него. Пътят бива запазен посредством структурата `TreePath`. Тя представлява стак от “позиционирани

възли” (фиг. 3.25), т.е всяка съдържа позицията на възела, индекса на възела в родителския списък с разклонения, както и индекса на намерения ключ, ако такъв съществува. При завършване на обхождането, освен резултата от търсенето, на потребителя се предоставя и конструирания път.

```
struct PosNod {
      Position node_pos;
      std::optional<std::size_t> idx_in_parent;
      std::optional<std::size_t> idx_of_key;
};
```

Фиг. 3.25 Дефиниция на PosNod

Основните функции, които директно употребяват функционалността на `search()` са:

- `contains(key)` - булева проверка за наличността на запис;
- `get(key)` - извличане на стойността, асоциирана с подадения ключ;
- `get_min_entry()` - извличане на наредената двойка, чийто ключ е най-малък;
- `get_max_entry()` - извличане на наредената двойка, чийто ключ е най-голям.

Всички те могат да хвърлят изключение по време на своето изпълнение. То съдържа съобщение, описващо характера на възникналата грешка.

Вмъкване

Вмъкването на запис в дървото се състои от три основни подоперации: намиране на подходящата позиция, извършване на промяната, възстановяване на баланса на дървото. Имплементацията на тази функционалност е представена на фиг. 3.26.

```
InsertionReturnMark insert(const Key &key, const RealVal &val) {
      return place_kv_entry(Entry{.key = key, .val = val},
                            ActionOnKeyPresent::AbandonChange);
}
```

```
InsertionReturnMark place_kv_entry(const Entry &entry,
    ActionOnKeyPresent action = ActionOnKeyPresent::AbandonChange,
    SplitBias split_bias = SplitBias::DistributeEvenly) {

    auto search_res = search(entry.key);
    if ((action == ActionOnKeyPresent::AbandonChange &&
                                        search_res.key_is_present)
    || (action == ActionOnKeyPresent::SubmitChange &&
                                       !search_res.key_is_present))
        return InsertedNothing();
    auto &leaf_node = search_res.node.leaf();
    leaf_node.keys.insert(
            leaf_node.keys.cbegin() + search_res.key_expected_pos,
                                                        entry.key);
    leaf_node.vals.insert(
            leaf_node.vals.cbegin() + search_res.key_expected_pos,
                                                set_value(entry.val));
    m_pager->place(search_res.path.top().node_pos,
                                       search_res.node.make_page());

    ++m_size;

    rebalance_after_insert(search_res.path, split_bias);
    return InsertedEntry();
}
```

Фиг. 3.26 Дефиниция на Btree::place_kv_entry()

Първата стъпка изцяло преизползва вътрешните функции `search()` и `search_subtree()`, реализиращи търсене. За да се вмъкне подадения запис в листо, трябва да се достъпи листото получено от `SearchReturnMark` и да се използва `std::vector::insert()` функцията. Особеност на тази част от операцията е функцията `set_value()` (вж. 3.2.5). Последната част е изведена като отделната функция – `rebalance_after_insert()`. Тя приема като аргумент изминатия път и го използва, за да може да се измине пътя в обратна посока. Ако се намери възел, който е препълнен, посредством `is_node_over()` функцията, се извършва разделяне. Процесът продължава, докато не се достигне възел, чийто баланс да не е засегнат. В най-лошия случай, разделянето се разпространява до корена на дървото и след като и

той се раздели, на негово място се поставя нов корен. Това е начинът, по който дървото увеличава своята височина. Реализацията на „разделяне” е предоставена от шаблона `Node` чрез функцията `Node::split()`. Имплементация на `rebalance_after_insert()` е изведена на фиг. 3.27.

```
void rebalance_after_insert(TreePath &visited, SplitBias bias) {
    std::optional<Nod> node;
    while (true) {
        const PosNod &path_of_node = visited.top();
        if (!node)
            node = Nod::from_page(
                            m_pager->get(path_of_node.node_pos));
        if (!is_node_over(*node))
            break;
        if (node->is_root()) {
            (void) make_root(MakeRootAction::NewTreeLevel);
            break;
        }

        auto [midkey, sibling] = node_split(*node, bias);
        auto sibling_pos = m_pager->alloc();
        node->set_next_node(sibling_pos);

        visited.pop();
        const PosNod &path_of_parent = visited.top();
        auto parent = Nod::from_page(
                          m_pager->get(path_of_parent.node_pos));

        const auto idx = path_of_node.idx_in_parent.value();
        parent.branch().refs.insert(
                    parent.branch().refs.cbegin() + idx, midkey)
        parent.branch().links.insert(
          parent.branch().links.cbegin() + idx + 1, sibling_pos);
        parent.branch().link_status.insert(
          parent.branch().link_status.cbegin() + idx + 1,
                                              LinkStatus::Valid);
         m_pager->place(sibling_pos, sibling.make_page());
        m_pager->place(path_of_node.node_pos, node->make_page());
        m_pager->place(path_of_parent.node_pos,
parent.make_page());
         node = parent;
     }
```

```
}
```

Фиг. 3.27 Имплементация на Btree::rebalance_after_insert()

След разделяне на възел биват обновени 3 възела (т.е 3 страници). Една от тях е родителя и неговия вектор с връзки, поради което този възел е извлечен чрез Pager полето. Същевременно следващата итерация на цикъла ще повтори прочитане на родител, тъй като това ще бъде следващия разглеждан възел. Поради тази причина се въвежда оптимизацията да се съхранява допълнителен възел със стойността на предишния разглеждан такъв.

Освен създаване на нов запис, дървото поддържа и възможност за презаписване на стойност, асоциирана със същия ключ, посредством `update()`. Реализацията е идентична, с малката разлика, че когато се засече наличието на вече съществуващ ключ, не се връща грешка, а се замества с новата посочена стойност. Това се дължи на стойността, която се подава за `ActionOnKeyPresent action`. Във функцията place_kv_entry(), която бива извикана както от `insert()`, така и от `update()`, стойността на action се взима предвид, ако подадения ключ е вече наличен в дървото. Възможните стойности за `action` са показани на фиг. 3.28.

```
enum class ActionOnKeyPresent { SubmitChange, AbandonChange };
```

Фиг. 3.28 Дефиниция на ActionOnKeyPresent

Важна особеност е, че при конструирането на нов възел, той първо бива поставен като „следващ възел” на разделения, а едва след това бива добавен в списъка с разклонения на родителя. Това отговаря на изискванията на B-link дърво, което предоставя възможността за реализация на транзакции [6, 9]. По-детайлно описание следва в 3.2.4.

Крайната стойност, която се връща от семейството функции, обслужвани от place_kv_entry() е InsertionReturnMark. Неговата дефиниция е представена на фиг. 3.29. Стойността на върнатата стойност описва крайното състояние на операцията.

```
struct InsertedEntry {};
struct InsertedNothing {};
using InsertionReturnMark =
```

```
                    std::variant<InsertedEntry, InsertedNothing>;
```

Фиг. 3.29 Дефиниция на InsertionReturnMark

Премахване

Изтриването на запис от дърво отново взаимства част от своята имплементация от операцията за извличане на данни. Аналогично на вмъкване, за да се премахне запис, той първо трябва да бъде локализиран в дървото. Тъй като всички записи се съхраняват в листата, записът се отстранява единствено от там, оставайки ключа в разклоняващите възли. Въпреки това, свойствата за търсене на дървото се запазват. Имплементацията на функцията е изложена на фиг. 3.30.

```
RemovalReturnMark remove(const Key &key) {
      auto search_res = search(key);

      if (!search_res.key_is_present)
            return RemovedNothing();

      auto &node_leaf = search_res.node.leaf();
      const auto &node_path = search_res.path.top();

      const auto removed = get_value(
                  node_leaf.vals.at(search_res.key_expected_pos));

      node_leaf.keys.erase(
            node_leaf.keys.cbegin() + search_res.key_expected_pos);
      if constexpr (Config::DYN_ENTRIES) {
            auto slot_id_it = node_leaf.vals.cbegin() +
                                        search_res.key_expected_pos;
            ind_vector().remove_slot(*slot_id_it);
      }
      node_leaf.vals.erase(
            node_leaf.vals.cbegin() + search_res.key_expected_pos);
       m_pager->place(
                  node_path.node_pos, search_res.node.make_page());
      --m_size;
     if constexpr (Config::BTREE_RELAXED_REMOVES)
          rebalance_after_remove_relaxed(search_res.path);

      return RemovedVal{.val = removed};
}
```

Фиг. 3.30 Имплементация на Btree::remove()

Въвежда се особен случай при употребата на динамично оразмерени типове, което налага използване на функцията `get_value()`, както и `ind_vector()` (вж. 3.2.5).

Балансирането, което се извършва, подлежи на конфигурация. Опцията `EugeneConfig::BTREE_RELAXED_REMOVES` отбелязва, че то не трябва да извършва смесвания на възли, което е традиционния метод, а да премахва единствено празни възли. Текущата реализация е на алгоритъма за релаксирано изтриване на записи, описан в [28]. Извършва се във частната функция `rebalance_after_remove_relaxed()`. Подобно на балансирането след вмъкване, отново се използва предоставения като аргумент `TreePath`. Балансирането започва от листото, от което е премахнат запис и продължава докато не бъде срещнат възел, чийто размер е различен от 0. Ако текущия е празен, той бива изтрит от списъка с разклонения на родителя. След края на операцията е гарантирано, че няма листо, което да не съдържа елементи. Реализацията е показана на фиг. 3.31.

```
void rebalance_after_remove_relaxed(TreePath &search_path) {
    while (!search_path.empty()) {
        const PosNod &path_of_curr = search_path.top();
        auto node = Nod::from_page(
                            m_pager->get(path_of_curr.node_pos));
        if (!node.is_empty() || node.is_root())
            return;

        m_pager->free(path_of_curr.node_pos);
        search_path.pop();
        auto parent_node = Nod::from_page(
                        m_pager->get(search_path.top().node_pos));
        auto &parent_links = parent_node.branch().links;
        parent_links.erase(parent_links.cbegin() +
                            path_of_curr.idx_in_parent.value());
    }
}
```

Фиг. 3.31 Имплементация на Btree::rebalance_after_remove_relaxed()

Съхраняване

Запазването и зареждането на инстанции на B-дърво се осъществява благодарение на допълнително съхранена метаинформация - заглавна част на структурата (фиг. 3.32).

```
struct Header {
      std::uint32_t magic{HEADER_MAGIC};
      Position tree_rootpos;
      std::size_t tree_size;
      std::size_t tree_depth;
      long tree_num_leaf_records;
      long tree_num_branch_records;

      NOP_STRUCTURE(Header, magic, tree_rootpos, tree_size,
tree_depth, tree_num_branch_records, tree_num_leaf_records);
};
```

Фиг. 3.31 Дефиниция на класа Header

Функцията `save()`, изпълняващ запазване на дървото, сериализира тази метаинформация за структурата и я запазва в отделен файл. Същия файл се използва от функцията `load()` при зареждане.

Освен данните директно свързани със физическата репрезентация на дървото, съхранява се също и инстанция на шаблона `Pager`, използвайки неговата `save()` функция. Такава съществува единствено на инстанциите, които имат свойството да бъдат запазвани на твърд диск, а не се намират само в RAM паметта на машината (имплементират `IPersistentPager`). Pager запазва всички промени, направени по страници в кеша, както и данните, съхранявани от алокатора на страници. Зареждането извършва аналогични операции, но с обратна насоченост – зареждат се данните на алокатора, метаинформацията на дървото от заглавната част и пр.

Освен експлицитно, `load()` може да бъде извикан и при инстанциране на B-дърво. При създаване на обект се подава аргумент на конструктора, обозначаващ дали структура с подадения идентификатор вече съществува и трябва да бъде заредена или да бъде инициализирана нова, използвайки `bare()`.

Имплементацията на `load()` функцията е представена на фиг. 3.32.

```
void load() {
      if constexpr (requires { m_pager->load(); }) {
            Header header_;
            nop::Deserializer<nop::StreamReader<std::ifstream>>
                              deserializer{header_name().data()};
            if (!deserializer.Read(&header_))
                  throw BadRead("[...]");
           m_rootpos = header_.tree_rootpos;
           m_size = header_.tree_size;
           m_depth = header_.tree_depth;
           m_num_records_leaf = header_.tree_num_leaf_records;
           m_num_records_branch = header_.tree_num_branch_records;
           m_num_links_branch = m_num_records_branch + 1;

           m_pager->load();
      }
      if constexpr (Config::DYN_ENTRIES) {
            if (!m_ind_vector.has_value())
                  m_ind_vector.emplace(
            fmt::format("{}-indvector",name()),
            IndirectionVector<Config>::ActionOnConstruction::Load);
            else
                  ind_vector().load();
      }
}
```

Фиг. 3.32 Имплементация на Btree::load()

Първата част от `load()` имплементацията се изпълнява само при инстанции на pager с `load()` метод, а втората само при структури, чиято конфигурация е позволила динамично оразмерените записи.

Функцията `save()`, която запазва заглавната част на структурата и прочита метаданните за другите запазени полета, работи по сходен начин.

3.2.4. Групови операции върху B-дърво

Освен операциите, извършвани върху дървото поединично, има множество възможности за оптимизирано изпълнение на една операция върху множество записи наведнъж [32, 38]. Следва да бъдат разгледани различните функционалности, които биват поддържани от системата.

Групово търсене

Груповото търсене или *филтриране* се базира на свойството на B+ дърветата да съхраняват всички свои записи в листата. Често срещан механизъм е всеки възел да познава своя съсед, независимо дали левия, или десния. В настоящата имплементация всеки възел, включително листата съдържат информация за позицията на своя десен съсед, намиращ се на същата височина. В допълнение, поради свойствата на B-дърветата, всички записи се намират в листата, наредени възходящо. Това позволява ефикасно итератиране по всички записи, намиращи се в дървото понастоящем.

Достъпването до всички записи е реализирано посредством C++20 функционалността coroutines и библиотеката CppCoro [1, 19]. Coroutine, това е рутина или функция, реализирана за кооперативна многозадачност [39]. Благодарение на основното си свойство за *повторно връщане*, те биват използвани и за т.нар генератори. Използвайки това свойство и типа `generator<T>` от библиотеката `CppCoro` е реализирана и функцията `get_all_entries()`. Тя имплементира итератор по листата на дървото, връщайки стъпка по стъпка всички записи. Въпреки това, няма момент, в който всички записи да са заредени в RAM паметта едновременно. Имплементацията на функцита е представена на фиг. 3.33.

```
cppcoro::generator<const Entry &> get_all_entries() {
      Nod curr = get_corner_subtree(root(), CornerDetail::MIN);
      if (!curr.is_leaf())
            throw BadTreeSearch("[...]");

      while (true) {
            for (const auto &&[key, val] :
                        iter::zip(curr.leaf().keys, curr.leaf().vals))
                  co_yield Entry{.key = key, .val = get_value(val)};
            if (!curr.next_node())
                  co_return;
            curr = Nod::from_page(m_pager->get(*curr.next_node()));
      }
}
```

Фиг. 3.33 Имплементация на Btree::get_all_entries()

Функцията използва `get_corner_subtree()` като първа стъпка от итерирането. Нейното предназначение е на посочена височина да намери възела, съдържащ или най-малкия, или най-големия запис, като кое от двете се определя от подадената стойност за `CornerDetail` - `Min` или `Max`. Тъй като в B-дървото всеки възел е свързан със следващия, за да се итерира през всички записи, трябва да се започне от най-малкия.

Посредством `get_all_entries()` се реализира и филтриране по дървото – `get_all_entries_filtered()`. Като аргумент се подава функтор, който да определи дали запис е отговаря на подадено изискване (фиг. 3.34).

```
cppcoro::generator<const Entry &> get_all_entries_filtered(
               auto filter_function,
               std::optional<std::function<bool(const Entry &)>>
                                          wrap_up_function = {}) {
     for (const auto &entry : get_all_entries()) {
          if (filter_function(entry))
               co_yield entry;
          else if(wrap_up_function && (*wrap_up_function)(entry))
               co_return;
     }
}
```

Фиг. 3.34 Имплементация на Btree::get_all_entries_filtered()

Въвежда се допълнителната оптимизация за прекратяване. Това се определя от допълнителен функтор за прекратяване на итерирането - `wrap_up_function()`. Не е задължителен и затова е обвит в `std::optional`. За разлика от `filter_function()`, `wrap_up_function()` разчита на точно определен списък с аргументи и връщана стойност.

Освен филтриране върху дървото може да бъде извършвано и търсене в интервал, т.е извличане на всички записи, чиито ключ попада в подаден интервал. Имплементацията е изведена във фиг. 3.35.

```
cppcoro::generator<const Entry &> get_all_entries_in_key_range(
                                              const Key &key_min,
                                              const Key &key_max) {
```

```
auto range_filter_function = [&](const Entry &entry) {
     return key_min <= entry.key && entry.key < key_max;
};

auto wrap_up_function = [key_max](const Entry &entry) {
     return entry.key >= key_max;
};

for (const auto &entry : get_all_entries_filtered(
                         range_filter_function,
                         std::make_optional(wrap_up_function)))
     co_yield entry;
}
```

Фиг. 3.35 Имплементация на Btree::get_all_entries_in_key_range()

Реализацията се базира на `get_all_entires_filtered()` заедно с `wrap_up_function`. Филтрират записите, които са по-малки от минималната стойност в интервала, и прекратяват операцията при откриване на по-голяма или равна на максималната стойност на интервала.

Групово вмъкване

Освен стандартното вмъкване на единичен запис е предоставена и възможност за оптимизирано вмъкване на множество записи наведнъж, посредством интерфейса `insert_many()`, получаващ като аргумент `std::ranges::range` от записи. Изискване към подадения аргумент е да подава записите в подреден възходящо ред. Алгоритъмът за въвеждане на множество от записи е описан в [32].

Подобно от останалите модифициращи функции на дървото, груповото вмъкване също се състои от три стъпки: разделяне на прости насипи (от англ. simple bulk), докато се намира подходяща позиция на всеки един от записите, извършване на промяната, водеща до валидно дърво за търсене, и балансиране на B-дървото.

Разделянето на прости насипи се осъществява по следния начин: започвайки от първия елемент на подадената колекция записи се намира подходящо листо, в което да бъде вмъкнат. След това се намира записа, чийто ключ има най-висока стойност и се побира в същото листо. Не се взема размера на даденото листо, а единственото интервала от стойности, които могат да бъдат запазвани в него, определен от

родителския възел. След като бъде определена подмножеството от подадения набор записи, се преминава към втората стъпка: вмъкване в дървото.

Вмъкването на подмножество се осъществява, конструирайки ново B-дърво, чийто корен замества избраното листо. Това B-дърво се нарича *дърво на вмъкването*. След като то е осъществено, голямата структура, от която е част дървото на вмъкване, представлява валидно дърво за търсене и традиционните операции за извличане, вмъкване и премахване функционират коректно.

Въпреки това, за да се гарантира максимално бързодействие е необходимо да се извърши балансиране. В случай то се осъществява посредством следния алгоритъм:

A) Именуваме корена на дървото на вмъкване с p и задаваме текущата височина на 1.

B) Разделяме p, задавайки като точка на разделяне индекса на корена на дървото на вмъкване в списъка с разклонения. Получените ляв и десен възел отбелязваме с p_L и p_R респективно.

C) Взимат най-левия и най-десния възел – q_L и q_R респективно, от текущата височина в дървото на вмъкване

D) Извършваме смесване на p_L с q_L в p_L и аналогично с p_R и q_R.

E) Инкрементираме текущата височина и ако не е достигната височината на корена, се връщаме в стъпка B).

Имплементацията на цялостния алгоритъм, изпълняван от `insert_many()` API е представена на фиг. 3.36, а функцията, която извършва модификацията по дървото - `place_kv_entries()` - на фиг. 3.37. Тя връща хеш-таблица, съдържаща `InsertionReturnMark` за всеки един от ключовете, които са част подадения насип.

```
std::unordered_map<Key, InsertionReturnMark> insert_many(
                                    std::ranges::range auto &&bulk,
  ActionOnKeyPresent action = ActionOnKeyPresent::AbandonChange) {
      namespace rng = std::ranges;
      if (rng::empty(bulk))
            return {};
      auto tmp = m_size;
      auto &&[insertion_marks, insertion_trees] =
```

```
                                    place_kv_entries(bulk, action);
      rebalance_after_bulk_insert(insertion_trees);
      m_size = tmp + rng::count_if(bulk, [&](const auto &entry) {
           return insertion_marks.contains(entry.key) &&
               std::holds_alternative<InsertedEntry>(
                                    insertion_marks.at(entry.key));
      });
      return insertion_marks;
}
```

Фиг. 3.36 Имплементация на Btree::insert_many()

```
auto place_kv_entries(std::ranges::range auto &&bulk,
                           ActionOnKeyPresent action) {
    namespace rng = std::ranges;

    ManyInsertionReturnMarks insertion_marks;
    std::vector<InsertionTree<Key>> insertion_trees;

    for (auto simple_bulk_cbegin = rng::cbegin(bulk);
         simple_bulk_cbegin != rng::cend(bulk);) {
        auto search_result = search(simple_bulk_cbegin->key);
        PosNod path_to_leaf = consume_back<PosNod>(
                                              search_result.path);
        std::optional<Position> parent_pos =
            search_result.path.empty()
                ? std::nullopt
                : std::make_optional(
                               search_result.path.top().node_pos);
        const auto leaf_vals_cbegin = search_result.node.leaf()
                                                    .vals.cbegin();
        const auto leaf_keys_cbegin = search_result.node.leaf()
                                                    .keys.cbegin();
        const auto leaf_keys_cend = search_result.node.leaf()
                                                      .keys.cend();
        const auto simple_bulk_cend = [&] {
            if (leaf_keys_cbegin == leaf_keys_cend)
                return rng::cend(bulk);
            const Key &leaf_highkey = *(leaf_keys_cend - 1);
            return std::find_if(simple_bulk_cbegin,
                 rng::cend(bulk), [&](const auto &entry) {
                     return entry.key > leaf_highkey;
```

```
            });
        }();
    auto entry_of_key_it = [&](auto it) {
        const auto &v = *(leaf_vals_cbegin + std::distance(
                                        leaf_keys_cbegin, it));
        return Entry{.key = *it,
                     .val = get_value(v)};
    };
    auto iterate_over_simple_bulk = [
         simple_bulk_cit = simple_bulk_cbegin,
         leaf_keys_cit = leaf_keys_cbegin,
         simple_bulk_cend, leaf_keys_cend, &entry_of_key_it]()
                  mutable -> cppcoro::generator<const Entry> {
        while (simple_bulk_cit != simple_bulk_cend &&
               leaf_keys_cit != leaf_keys_cend)
            if (simple_bulk_cit->key < *leaf_keys_cit)
                co_yield *simple_bulk_cit++;
            else
                co_yield entry_of_key_it(leaf_keys_cit++);
        while (leaf_keys_cit != leaf_keys_cend)
            co_yield entry_of_key_it(leaf_keys_cit++);
        while (simple_bulk_cit != simple_bulk_cend)
            co_yield *simple_bulk_cit++;
    };
    insertion_trees.push_back(
        InsertionTree{.path = search_result.path,
                      .tree = clone_only_blueprint(),
                      .lofence = simple_bulk_cbegin->key,
                      .hifence = simple_bulk_cend >
                                         simple_bulk_cbegin
                                 ? (simple_bulk_cend - 1)->key
                                 : simple_bulk_cbegin->key,
                      .leaf_pos = path_to_leaf.node_pos});
    auto &insertion_tree = insertion_trees.back();
    rng::for_each(iterate_over_simple_bulk(),
                                       [&](const auto &entry) {
        insertion_marks[entry.key] = insertion_tree.tree
                                           .place_kv_entry(
      typename Nod::Entry{.key = entry.key, .val = entry.val},
                                action, SplitBias::LeanLeft);
    });
    auto pos = [&] {
```

```
            if (!parent_pos)
                return (m_rootpos =insertion_tree.tree.rootpos());
            const auto new_pos = m_pager->alloc();
            auto parent = Nod::from_page(
                                        m_pager->get(*parent_pos));
            parent.branch().links[*path_to_leaf.idx_in_parent] =
                                                          new_pos;
            m_pager->place(*parent_pos, parent.make_page());
            if (*path_to_leaf.idx_in_parent > 0) {
                auto prev_sibling_pos = parent.branch().links
                                [*path_to_leaf.idx_in_parent - 1];
                auto prev_sibling = Nod::from_page(
                                  m_pager->get(prev_sibling_pos));
                prev_sibling.set_next_node(new_pos);
                m_pager->place(
                      prev_sibling_pos, prev_sibling.make_page());
            }
            return (insertion_tree.leaf_pos = new_pos);
        }();
        m_pager->place(pos, insertion_tree.tree.root()
                                                    .make_page());
        simple_bulk_cbegin = simple_bulk_cend;
        path_to_leaf.node_pos = insertion_tree.tree.rootpos();
        insertion_tree.path.push(path_to_leaf);
    }

    return std::make_pair(std::move(insertion_marks),
                          std::move(insertion_trees));
}
```

Фиг. 3.37 Имплементация на Btree::place_kv_entries()

3.2.5. Записи с динамичен размер

Дотукописаната система поддържа съхраняване единствено на записи със статично определен размер, т.е такива, които не го изменят динамично в хода на изпълнението на програмата. Такива са например всички примитивни типове. Това се дължи на метода, по който въведените данни биват оформени в заделените страници на ниво двоичен код. При инстанциране на `Btree` се изчислява колко записи могат да бъдат записани в една страница и съответно в един възел на дървото. Това се случва използвайки двоично търсене във функцията `bare()`. Това изчисление не получава

верен отговор само за динамично оразмерени типове като `std::string` и `std::vector`. За да могат да бъдат съхранявани и записи с променлива дължина, се въвежда нова абстракция, която посредством индирекция да изпълнява операциите коректно.

Слотове

Основната структура, която се използва, за да се опише един тип, който изменя размера си в хода на изпълнение, е слота (`Slot`). Той представлява дескриптор на заето пространство - стартова позиция и размер (фиг. 3.38). Също така се съдържа и булев флаг, маркиращ дали в слота се съдържа валидна информация, т.е дали се използва. Всеки слот описва единствена стойност.

```
struct Slot {
      storage::Position pos = 0;
      std::size_t size = 0;
      bool occupied;

      NOP_STRUCTURE(Slot, pos, size, occupied);
};
```

Фиг. 3.38 Дефиниция на структурата Slot

Вектор за индиректност

Имплементира се т.нар *вектор за индиректност* (още известен като масив за слотове), в който се съдържат слотове за всички текущи стойности в дървото. Във възлите на B-дървото вместо потребителските данни като стойност, се записва индекса на съответния слот, който ги описва. Поради тази причина в конфигурационната структура се срещат два типа за стойност: `Val` и `RealVal`. Чрез първата се описва типа данни, които да бъдат съхранявани във векторите на листата, т.е фактическите данни, върху които се извършват операциите. `RealVal` се използва за API, където е необходимо да се достъпят данните, въведени от потребителя - например при извличане.

На фиг. 3.39 е представена дефиницията на класа `IndirectionVector`.

```
using SlotId = std::size_t;

template<EugeneConfig Config>
class IndirectionVector {
public:
      enum class ActionOnConstruction { Load,
                                        DoNotLoad };

      template<typename... Args>
      explicit IndirectionVector(std::string identifier,
                                 ActionOnConstruction action,
                                 Args &&...args)
          : m_identifier{identifier},
            m_slot_pager{std::make_shared<PagerType>(
                           fmt::format("{}-pager", identifier),
                                 std::forward<Args>(args)...)} {
            switch (action) {
                  break; case ActionOnConstruction::Load: load();
                  break; case ActionOnConstruction::DoNotLoad:
                                                          DO_NOTHING;
            }
      }

      IndirectionVector(const IndirectionVector &) = default;
      IndirectionVector &operator=(const IndirectionVector &) =
                                                              default;
      void load() { [...] }
      void save() { [...] }
      SlotId set_to_slot(const RealVal &) { [...] }
      void replace_in_slot(const SlotId,
                           const RealVal &, const auto) { [...] }
      void remove_slot(const SlotId) { [...] }
      RealVal get_from_slot(const SlotId) { [...] }

      SlotId alloc_slot() { [...] }
      void free_slot(SlotId) { [...] }

private:
      std::vector<Slot> m_slots;
      std::string m_identifier;
      std::shared_ptr<PagerType> m_slot_pager;
       mutable std::mutex m_mutex;
```

```
      NOP_STRUCTURE(IndirectionVector, m_slots, m_identifier);
};
```

Фиг. 3.39 Дефиниция на класа IndirectionVector

Управление на слотовете

Управлението на слотове се извършва от класа `IndirectionVector`, посредством интерфейс за заделяне и освобождаване на слотове; записване и изтриване; извличане и обновяване. Класът работи, използвайки вътрешна инстанция на Pager, който управлява пространството, върху което са разположени потребителските данни. Използва се поддържаният интерфейс за вътрешни операции по страници, посредством който се реализират операциите върху слотове. На фиг. 3.40 са представени две от функциите, експортирани от класа.

```
private:
SlotId alloc_slot() {
      auto it = std::find_if(m_slots.cbegin(),
                             m_slots.cend(), [&](const auto &slot) {
            return !slot.occupied;
      });

      return static_cast<SlotId>([&] {
            if (it == m_slots.cend())
                  return static_cast<SlotId>(std::distance(it,
m_slots.cbegin()));
            m_slots.emplace_back();
            return m_slots.size() - 1;
      }());
}

public:
SlotId set_to_slot(const RealVal &val) {
      std::scoped_lock<std::mutex> _guard{m_mutex};
      nop::Serializer<nop::StreamWriter<std::stringstream>>
                                                      serializer;
      if (!serializer.Write(val))
            throw BadWrite("[...]"));
      const auto str = serializer.writer().stream().str();
```

```
        std::vector<uint8_t> val_data{str.cbegin(), str.cend()};
        const auto sz = val_data.size();

        const auto pos = m_slot_pager->alloc_inner(sz);
        m_slot_pager->place_inner(pos, val_data);

        auto slot_id = alloc_slot();
        m_slots.emplace(m_slots.cbegin() + slot_id, pos, sz);
        return slot_id;
}
```

Фиг. 3.40 Имплементация на IndirectionVector::{alloc_slot(), set_to_slot()}

`set_to_slot()` е част от публичния интерфейс на класа. Служи за вмъкване на потребителски данни, т.е за стойността подадена като аргумент се сериализира, заделя се слот и място и се пространство в съответната позиция, а накрая на извикващия се връща идентификатора на избрания слот. Примитивите за заделяне на физическо пространство и записване са изцяло предоставени от `Pager`, наследявайки `ISupportingInnerOperations`. За да се определи кой слот от целия вектор за индиректност да бъде използван се извиква `alloc_slot()`. Тази функция итерира през вектора, търсейки слот, чийто флаг `occupied` не е поставен. Ако такъв не съществува в текущия момент, бива създадена нова инстанция на `Slot`, която се поставя в края на вектора.

`IndirectionVector` поддържа примитиви за запазване и зареждане на състоянието, а именно `load()` и `store()`. Чрез макрото `NOP_STRUCTURE` са описани и данните, които да бъдат сериализирани.

Вектор за индиректност в B-дървото

Едно от полетата на B-дървото е инстанция на `IndirectionVector`, която е обвита от `std::optional`. Вътре бива поставена стойност единствено при наличие на опцията `EugeneConfig::DYN_ENTRIES`. Това се случва от функциите `Btree::bare()` и `Btree::load()`. Инициализацията е представена на фиг. 3.41.

```
if constexpr (Config::DYN_ENTRIES) {
    if (!m_ind_vector.has_value())
```

```
        m_ind_vector.emplace(fmt::format("{}-indvector", name()),
                                        IndirectionVector<Config>::
                             ActionOnConstruction::DoNotLoad);
}
```

Фиг. 3.41 Инициализация на IndirectionVector m_ind_vector

За да се избегне натрупване на логиката на операциите на дървото с такава, занимаваща се да извлича и поставя правилните данните, спрямо конфигурационната опция `DYN_ENTRIES`, се въвеждат функциите `Btree::get_value()` и `Btree::set_value()`. Фиг. 3.42 представя тяхната имплементация.

```
RealVal get_value(Val val_or_slot) {
      if constexpr (Config::DYN_ENTRIES) {
            return ind_vector().get_from_slot(val_or_slot);
      } else {
            static_assert(std::same_as<Val, RealVal>);
            return val_or_slot;
      }
}

Val set_value(RealVal val) {
      if constexpr (Config::DYN_ENTRIES) {
            return ind_vector().set_to_slot(val);
      } else {
            static_assert(std::same_as<Val, RealVal>);
            return val;
      }
}
```

Фиг. 3.42 Имплементация на `Btree::{set, get}_value()`

`get_value()` се използва, когато е извлечен стойност от вектора в листото. Тъй като на потребителя трябва да бъде върнат данните, които са били въведени, то ако конфигурацията подсказва, че се използват динамични записи, тогава стойността всъщност е `SlotId` и трябва да се извърши прочит от `Pager` инстанцията на `IndirectionVector` инстанцията. На сходен принцип работи и `set_value()`.

3.2.6. Паралелност

Дотук описанието на дипломния проект разглежда предимно алгоритмите и структурите от данни, които се използват, за да предоставят функционалността в еднонишков контекст. За да бъде реализирана и паралелна обработка на заявки е използвано т.нар *блокиращо* синхронизиране или заключване (от англ. locking). То се характеризира с употребата на ключалка (в случая инстанция на `std::mutex`), чиито операции (заключване и отключване) ограничават достъпа до *критична секция* от кода. За критична секция се приема част от имплементацията, в която множество нишки на изпълнение достъпват данни, което поради моделът на памет (от англ. memory model), използван от C++, води до недетерминистичен достъп и обновяване на стойностите.

Паралелност в Pager модула

В Pager модула има три основни компонента, които да бъдат защитени - `Pager` класа, кешът за страници и алокатора за страници. И в трите имплементации се използват инстанции на `std::mutex` като полетеа на класа. които биват заключени в началото на всяка функция, част от публичния интерфейс на класа (фиг. 3.43).

```
private:
mutable std::mutex m_mutex;

public:

[...]
{ // начало на критичната секция
     std::scoped_lock<std::mutex> _guard{m_mutex};
     [...]
} // край на критичната секция
[...]
```

Фиг. 3.43 Заключване на публичен API, използвайки полето `m_mutex`

`m_mutex.lock()` и `m_mutex.unlock()` API изискват `m_mutex` да не бъде константен, което ограничава употребата във функции на класа, които приемат `const this`. За да бъде използван в такива, в декларацията на `m_mutex` се използва `mutable`.

Управлението на `m_mutex` се случва чрез локални шаблони `std::scoped_lock<std::mutex>`, използвайки парадигмата Resource Acquisition

Is Initialization (RAII). В конструктора на `std::scoped_lock` се извиква `m_mutex.lock()`, а в деструктора - `m_mutex.unlock()`. Това предпазва от възможността да бъде пропуснато извикването на втората операция.

Функциите, част от частния интерфейс, не използват `m_mutex`. Това е, за да се предпази от повторно заключване на един и същ мутекс от една нишка, което според C++ стандарта е недефинирано поведение.

Паралелност в модула B-дърво

За изпълнението на паралелност в този модул се комбинират няколко различни подхода със заключване. Промяна в, което и да е от, полетата на шаблона `Btree`, биват защитени от инстанция на `std::mutex` - `m_mutex`, т.е промяна на `m_size`, `m_depth` или `m_rootpos` биват съпътствани със съответен `std::scoped_lock<std::mutex>`.

Използваните ключалки са разпространени в литературата като резета (от англ. latches). Тяхното основно предназначение е да ограждат критичните секции в операциите, изпълнявани върху вътрешни структури от множество нишки едновременно. Те предпазват *логическата репрезентация* на базата данни. Нямат за цел да могат да се изпълняват в обратна посока (от англ. rollback).

Логическата цялост на дървото се предпазва чрез метода latch coupling. Всеки възел има асоциирано резе (от англ. latch), което може да бъде придобито в един от два режима - за четене или за писане, посредством `std::shared_lock` или съответно `std::unique_lock`. Това означава, че при придобиване на достъп със състояние на четене (`std::shared_lock`), не се ограничава достъпа на други нишки, изпълняващи се едновременно. Същевременно, ако ключалката бъде заключена с `std::unique_lock`, това блокира както четящи нишки, така и пишещи. На фиг. 3.44 са изложени двата макроса, които се използват при придобиване на достъп, съответстващ с необходимото от текущата операция.

```
#define ACQUIRE_READ_LATCH(l) \
    std::shared_lock<std::mutex> _guard{(l)};

#define ACQUIRE_WRITE_LATCH(l) \
    std::unique_lock<std::mutex> _guard{(l)};
```

Фиг. 3.44 Макроси за придобиване на ниво на достъп

Заключването се осъществява преди достъп до съответния възел, използвайки коректния режим на заключваме, зависещ от изпълняваната операция. Отключване на възел се осъществява според няколко изисквания:

- Ако възелът е разклоняващ и е *безопасен*, то той се отключва след като бъде придобита ключалка за наследика.
- Ако възелът е разклоняващ, но не е безопасен, ключалката се задържа до края на операцията.
- Ако възелът е листо, то той се отключва след приключване на операцията.

Безопасен е възел, който независимо от модификацията, която ще извърши съответната операция, при изпълнение на балансиращата функция допълнителните промени няма да се разпространят извън текущия възел, “абсорбирайки” всякакви промени. Също така, тъй като разклоняващите възли се грижат за насочването на операциите към съхранения в листо запис, всички листа се приемат за безопасни. Имплементацията на проверките за безопасност според операцията са представени на фиг. 3.45.

```
constexpr bool is_node_insertion_safe(const Nod &node) {
      if (node.is_branch())
            return node.num_filled() < max_num_records_branch();
      return false;
}

constexpr bool is_node_remove_safe(const Nod &node) {
      if (node.is_branch())
            return !node.empty();
      return false;
}
```

Фиг. 3.45 Имплементации на Btree::is_node_{insertion, remove}_safe

3.2.7. Конфигуриране

Системата се конфигурира посредством инстанция на клас, наследяващ `Config`. За обозначаване на типове се използват `using` декларации, а за стойности -

`static constexpr` идентификатори. При предаване на конфигурации като параметър - и шаблонен, и стандартен, се използва концепцията `EugeneConfig`. На фиг. 3.46 е изложена конфигурацията по подразбиране.

```
struct Config {
      using Key = int;
      using Val = int;
      using Ref = int;

      using PageAllocatorPolicy = storage::FreeListAllocator;
      using PageEvictionPolicy = storage::LRUCache;
      using PagerType =
          storage::Pager<PageAllocatorPolicy, PageEvictionPolicy>;

      static inline constexpr int PAGE_CACHE_SIZE = 1_MB;
      static inline constexpr bool APPLY_COMPRESSION = true;
      static inline constexpr int BRANCHING_FACTOR_LEAF = 0;
      static inline constexpr int BRANCHING_FACTOR_BRANCH = 0;
      static inline constexpr bool PERSISTENT = true;

      static inline constexpr bool BTREE_RELAXED_REMOVES = true;
      static inline constexpr bool DYN_ENTRIES = false;
      using RealVal = Val;
};
```

Фиг. 3.46 Конфигурация по подразбиране

Конфигуриране на B-дърво

B-дървото се конфигурира посредством подадената, като шаблонен параметър, конфигурация. Опциите, които се избират са следните:

Типове

- `Key` - типът на ключа, който се използва в дървото;
- `Val` - типът на стойност, с която боравят алгоритмите на дървото;
- `Ref` - типът на ключа, който се използва в разклоняващите възли на дървото;
- `RealVal` - типът на потребителските данни, които се съхраняват в дървото. За да бъде използван, трябва опцията DYN_ENTRIES да е поставена. В противен случай `RealVal` съвпада с типа `Val`.

- `PagerAllocatorPolicy` - policy клас, която да бъде използвана за заделяне на нови страници.
- `PagerEvictionPolicy` - policy клас, която да бъде използва при определяне на това коя страница да бъде премахната от кеша за страници;
- `PagerType` - клас, който да бъде използван като интерфейс към Pager модула.

Константи

- `PAGE_CACHE_SIZE` - размер на кеша за страници, подаден в байтове;
- `DYN_ENTIRES` - маркира дали `Val` и `RealVal` се различават, т.е дали типовете са динамично оразмерени;
- `BTREE_RELAXED_REMOVES` - маркира дали да бъде използвано релаксирано премахване на елементи;
- `BRANCHING_FACTOR_BRANCH` - изрично уточнен ред на дървото, относно броя елементи в разклоняващите възли;
- `BRANCHING_FACTOR_LEAF`[11] - изрично уточнен ред на дървото, относно броя елементи в разклоняващите възли;
- `PERSISTENT` - маркира дали съхраняваните данни се намират единствено в паметта или на постоянна памет.

Освен чрез директно наследяване, могат да бъдат използвани и макросите `EU_CONFIG` и `EU_CONFIG_DYN` за улеснена конфигурация на B-дървото. Те са реализирани както е изложено на фиг. 3.47. Тяхната функционалност е лимитирана, но облекчава дефиницията на проста конфигурация.

```
#define EU_CONFIG(Conf, K, V) \
    struct Conf : Config { \
        using Key = K; \
        using Ref = K; \
        using Val = V; \
        using RealVal = V; \
    };
```

[11] `BRANCHING_FACTOR_BRANCH` и `BRANCHING_FACTOR_LEAF` не могат да бъдат произволни стойности, а трябва да отговарят на изискванията в `Btree::sanity_check()`.

```
#define EU_CONFIG_DYN(Conf, K, V) \
    struct Conf : Config { \
        using Key = K; \
        using Ref = K; \
        using RealVal = V; \
        static constexpr bool DYN_ENTIRES = true; \
    };
```

Фиг. 3.47 Улеснен метод за конфигуриране чрез макроси

3.3. Компресия

Компресиране е процес, при който се изменя физическата репрезентация на данни, така че те да заемат по-малко място, запазвайки тяхната логическа стойност. Те биват класифицирани спрямо това дали възникват загуби при минимизирането на обема. Първите намират своето приложение, но са непрактични при използване в СУБД. Съществуват редица методи за реализиране на алгоритъм за компресиране без загуба, сред които са алгоритъм на Шенън-Фано, кодиране по дължина, речниково кодиране и ентропологично кодиране. Избраният метод е известен като кодиране на Хъфман [15]. Това алгоритъм без загуба на данни, чиято ефективност се е доказала през годините.

Кодиране на Хъфман

Кодирането на Хъфман приема като входен аргумент данни, които да бъдат компресирани (*съобщение)*, и генерира тяхната кодирана репрезентация. Той бива описан от следните стъпки [26]:

A) За всеки байт от подаденото съобщение се образува тривиално дърво - структура *trie* [10] (фиг. 3.47). В неговия корен се отбелязва *вероятността* на дадения байт информацията.

B) Намират се двата възела, чиято вероятност е най-малка, и се комбинират в нов възел. Корена на новообразуваната структура съдържа сумата от вероятностите на смесените възли.

C) Ако съществуват поне 2 дървета се повтаря стъпка B).

Алгоритъма за декомпресиране работи аналогично.

```
struct huff_trie {
      huff_trie *left{nullptr};
     huff_trie *right{nullptr};
      uint8_t character;
      long int char_occurrence;
      std::string bit;

      huff_trie() = default;
      huff_trie(long int num, uint8_t c) :
                              character(c), char_occurrence(num) {}

      bool operator<(const huff_trie &second) const {
          return this->char_occurrence < second.char_occurrence;
      }
};
```

Фиг. 3.47 Дефиниция на huff_trie структура

3.4. Сървър

3.5. Коректност на имплементацията

В хода на разработка освен реализацията на основните функционалности, са създавани и поддържани unit-тестове, целящи да верифицират правилното поведение на системата. Те са разпределени в отделените файлове с разширение `*.cpp`. За имплементиране и изпълнение на тестове е използвана библиотеката Catch2 [17].

Тестове върху Pager модула

Pager модулът е тестван посредством няколко тестови случая, някои от които имат подразделения на отделни секции:

- "Page" проверява основните примитиви на Pager класа - `place()` и `get()`;
- "Persistent Pager" проверява имплементациите на функциите load() и store(), сериализирайки `Pager`, който използва `StackAllocator` и такъв, който използва `FreeListAllocator`;
- "Page stack allocator" верифицира коректността на имплементацията на класа `StackAllocator` - заделяне с `alloc()` и освобождаване с `free()`;

- "Page free list allocator" верифицира коректността на имплементацията на класа FreeListAllocator - заделяне с `alloc()`, освобождаване с `free()`, както и правилното управление на създадения `freelist`;
- "Page cache with LRU policy" потвърждава правилното изпълнение при поставяне и извличане на страници, както и при премахване на страници от кеша (фиг. 3.48);
- "Pager inner operations" отговаря за правилното изпълнението на примитивите, наложени от `ISupportingInnerOperations - {alloc, free, place, get}_inner()`;
- "Pager concurrency" отговаря за правилното изпълнение на функционалностите на Pager модула в многонишкова среда.

```
TEST_CASE("Page cache with LRU policy", "[pager]") {
      PageCache<LRUCache> cache(4);
      Page p;
      for (int i = 0; i < 4; ++i) {
            std::fill(p.begin(), p.end(), i);
            REQUIRE(!cache.place(i * PAGE_SIZE, Page(p)));
            REQUIRE(p == cache.get(i * PAGE_SIZE)->get());
      }
      REQUIRE(!cache.get(42).has_value());
      std::fill(p.begin(), p.end(), 42);
      auto evict_res1 = cache.place(4 * PAGE_SIZE, Page(p));
      REQUIRE(evict_res1.has_value());
      REQUIRE(cache.get(4 * PAGE_SIZE)->get() == p);
      REQUIRE(std::find(evict_res1->page.cbegin(),
          evict_res1->page.cend(), 0) != evict_res1->page.cend());
      REQUIRE(evict_res1->pos == 0 * PAGE_SIZE);
      std::fill(p.begin(), p.end(), 13);
      auto evict_res2 = cache.place(5 * PAGE_SIZE, Page(p));
      REQUIRE(evict_res2.has_value());
      REQUIRE(cache.get(5 * PAGE_SIZE)->get() == p);
      REQUIRE(std::find(evict_res2->page.cbegin(),
          evict_res2->page.cend(), 1) != evict_res2->page.cend());
      REQUIRE(evict_res2->pos == 1 * PAGE_SIZE);
}
```

Фиг. 3.48 Unit-тест "Page cache with LRU policy"

Тестове върху класа Node от модула B-дърво

Класът `Node` е тестван посредством няколко тестови случая, сред които някои използват повече от една секция:

- ”Node serialization” проверява дали се извършва преобразуването между възел и страница;
- ”Persistent nodes“ верифицира, че инстанции на класа могат да бъдат запазвани върху диск, а след това да бъдат правилно разчитани;
- ”Paging with many random nodes” запазва множество инстанции върху диск, принуждавайки Pager обекта да извършва алокации и прехвания от кеша;
- “Split nodes“ изпълнява разделяне върху възли, използвайки различни енумератори за `SplitBias` аргумента (фиг. 3.49);
- “Fuse nodes” изпълнява смесване на разклоняващи възли и листа.

```
auto validate_branch = [](const Nod::Branch &branch_before_split,
                          const Nod::Branch &left,
                          const Nod::Branch &right,
                          const std::size_t pivot_idx,
                          const auto midkey) {
     REQUIRE(midkey == branch_before_split.refs[pivot_idx]);
     REQUIRE(std::equal(branch_before_split.refs.cbegin(),
                    branch_before_split.refs.cbegin() + pivot_idx,
                    left.refs.cbegin()));
     REQUIRE(std::equal(branch_before_split.links.cbegin(),
              branch_before_split.links.cbegin() + pivot_idx + 1,
              left.links.cbegin()));

    REQUIRE(std::equal(branch_before_split.link_status.cbegin(),
        branch_before_split.link_status.cbegin() + pivot_idx + 1,
        left.link_status.cbegin()));
    REQUIRE(std::equal(branch_before_split.refs.cbegin() +
                                                    pivot_idx + 1,
                                  branch_before_split.refs.cend(),
                                  right.refs.cbegin()));
     REQUIRE(std::equal(branch_before_split.links.cbegin() +
                                                    pivot_idx + 1,
                                 branch_before_split.links.cend(),
```

```
                                    right.links.cbegin()));
    REQUIRE(std::equal(branch_before_split.link_status.cbegin() +
                                                    pivot_idx + 1,
                           branch_before_split.link_status.cend(),
                           right.link_status.cbegin()));
};

SECTION("Even distribution of entries") {
     auto [branch_midkey, branch_sib_node] = branch_node.split(
                        BRANCH_LIMIT, SplitBias::DistributeEvenly);
     validate_branch(branch_before_split,
                     branch,
                     branch_sib_node.branch(),
                     (BRANCH_LIMIT + 1) / 2,
                     branch_midkey);
     [...]
}
```

Фиг. 3.49 Секция "Even distribution of entries" от unit-тест “Split nodes”

Тестове върху класа Btree от модула B-дърво

Класът Btree е тестван посредством няколко тестови случая, сред които някои използват повече от една секция:

- ”Btree operations” в отделни секции проверява дали основните операции върху дървото се извършват правилно. Част от тях са секциите “Create tree”, “Insertion”, “Removal”, “Update”, “Queries”.
- ”Btree bulk operations” проверява дали груповите операции се изпълняват правилно.
- ”Btree persistence” верифицира, че дървото заедно с неговите полета успешно се запазват на диск.
- ”Btree configs” проверява дали имплементацията работи по еднакъв начин с различни конфигурации.
- ”Btree utils” потвърждава, че функциите, асоциирани с `Node` класа се изпълняват правилно;
- ”Btree dyn entries” проверява дали дървото правилно изпълнява операции при динамично оразмерени записи;

- ”Btree concurrency” потвърждава, че дървото изпълнява правилно операциите си в многонишков контекст.

```
SECTION("Queries") {
      Bt bpt("/tmp/eugene-tests/btree-operations/queries");
      std::vector<Bt::Entry> inserted_entries;
      const int limit = 1000;
      for (const int item : std::ranges::views::iota(0, limit)) {
            inserted_entries.push_back(Bt::Entry{item, item});
            bpt.insert(item, item);
      }

      REQUIRE(*bpt.get_min_entry() ==
                                    Bt::Entry{.key = 0, .val = 0});
      REQUIRE(*bpt.get_max_entry() ==
                  Bt::Entry{.key = limit - 1, .val = limit - 1});
      std::vector<Bt::Entry> fetched_entries;
      for (const Bt::Entry &item : bpt.get_all_entries())
            fetched_entries.push_back(item);
      REQUIRE(std::ranges::equal(fetched_entries,
                                                 inserted_entries));
      std::vector<Bt::Entry> inserted_entries_with_odd_keys{
                                                  inserted_entries};
      std::erase_if(inserted_entries_with_odd_keys,
        [](const Bt::Entry &entry) { return entry.key % 2 == 0; });

      std::vector<Bt::Entry> fetched_entries_with_odd_keys;
      for (const Bt::Entry &item : bpt.get_all_entries_filtered(
                                    [](const Bt::Entry entry) {
                                         return entry.key % 2 != 0;
       }))
            fetched_entries_with_odd_keys.push_back(item);

     REQUIRE(std::ranges::equal(fetched_entries_with_odd_keys,
                                  inserted_entries_with_odd_keys));
      std::vector<Bt::Entry> inserted_entries_in_given_range{
                                                  inserted_entries};
      std::erase_if(inserted_entries_in_given_range,
                                       [](const Bt::Entry &entry) {
             return entry.key < 65'900 || entry.key >= 66'000; });
```

```
    std::vector<Bt::Entry> fetched_entries_in_given_range;
    for (const Bt::Entry &item :
                bpt.get_all_entries_in_key_range(65'900, 66'000))
        fetched_entries_in_given_range.push_back(item);

    REQUIRE(std::ranges::equal(fetched_entries_in_given_range,
                               inserted_entries_in_given_range));
}
```

Фиг. 3.50 Секция "Queries" от unit-тест “Btree operations”

3.6. Примерна реализация на база данни

Създадена е примерна база данни, която съхранява описание на хора, асоциирани с уникален идентификатор (фиг. 3.51)

```
struct Person {
    std::uint64_t phonenumber;
    std::uint32_t age;
    std::string name;
};

using PersonId = std::uint64_t;

using Key = PersonId;
using Val = Person;
```

Фиг. 3.51 Типове, които да бъдат съхранявани в базата данни

След като те бъдат дефинирани, следва да бъде генерирана конфигурация за желаното B-дърво. Това е изложено на фиг. 3.52.

```
EU_CONFIG_DYN(Person, Key, Val);

Btree<PersonConfig> btree;
```

Фиг. 3.52 Създаване на конфигурация и дърво за съхранение

Това е необходимото настройване на библиотеката, което е необходимо. В примерната реализирана база данни, се поддържат основните операции на B-дървото, посредством интерфейс на командния ред. Основният задвижващ код е показан на фиг. 3.53.

```
int main(int, char**) {
    while (1) {
        std::cout << "Operation = ";
        switch (getchar()) {
            break; case 'o': eu_open(); break;
            break; case 'c': eu_close(); break;
            break; case 'i': eu_insert(); break;
            break; case 'u': eu_update(); break;
            break; case 'r': eu_remove(); break;
            break; case 'q': return 0;
          }
     }
}
```

Фиг. 3.53 Основен задвижващ код на примерната база данни

Всяка една от извиканите функции, използва публичния интерфейс на модулът B–дърво. Реализацията на част от тях е показана на фиг. 3.54.

```
void eu_insert() {
    std::cout << " --- Insert ---\n";
    Person p;
    std::cout << "Person = ";
    std::cin >> p;
    auto rv = btree.insert(counter_id, p);
    if (std::holds_alternative<InsertedEntry>(rv)) {
        std::cout << "insertion succeeded with id = "
                                          << counter_id <<'\n';
        ++counter_id;
    } else
        std::cout << "insertion failed\n";
}

void eu_remove() {
     std::cout << " --- Remove ---\n";
     PersonId id;
```

```
        std::cout << "Id = ";
        std::cin >> id;
        auto rv = btree.remove(id);
        if (std::holds_alternative<RemovedVal>(rv))
                std::cout << "remove succeeded\n";
        else
                std::cout << "remove failed\n";
}
```

Фиг. 3.53 Основен задвижващ код на примерната база данни

Глава 4. 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 правил 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 изложения на фиг. 4.4.

```
    Start 1: TestCompression
1/5 Test #1: TestCompression ..................   Passed    0.12 sec
    Start 2: TestBtree
2/5 Test #2: TestBtree ........................   Passed    0.13 sec
    Start 3: TestNode
3/5 Test #3: TestNode .........................   Passed    0.01 sec
    Start 4: TestPager
4/5 Test #4: TestPager ........................   Passed    0.01 sec
    Start 5: TestUtil
5/5 Test #5: TestUtil .........................   Passed    0.01 sec

100% tests passed, 0 tests failed out of 5

Total Test time (real) =   0.27 sec
```

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

4.2. Ръководство за имплементиране на бази данни

Конфигурация

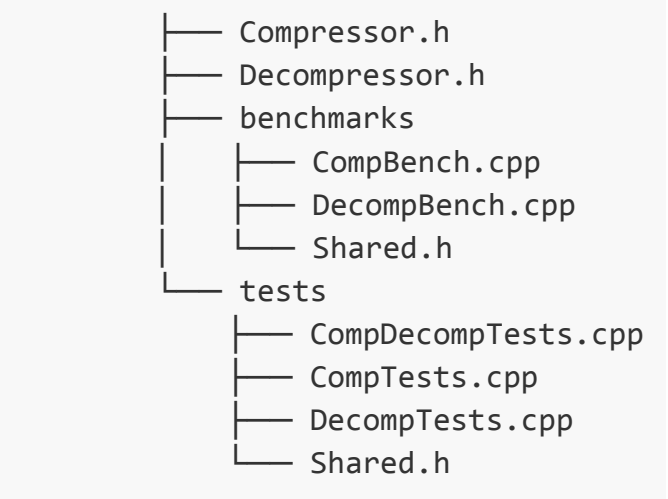
Първата стъпка при създаване на нова база данни, използвайки библиотеката, е да се дефинира конфигурация. Това се случва, като се наследи класа `Config`, намиращ се в `Config.h`. Съществуват различни видове конфигурации според желаната функционалност.

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 `Phonenumber`), 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 функционалносте, които имплементират.

```
├── Config.h
├── Logger.h
├── Main.cpp
├── Util.cpp
├── Util.h
├── server
│   ├── CMakeLists.txt
│   ├── handler
│   │   ├── Handler.cpp
│   │   └── Handler.h
│   ├── src
│   │   ├── ClientGroup.cpp
│   │   ├── ClientGroup.h
│   │   ├── User.cpp
│   │   └── User.h
│   └── tests
│       └── ExampleUsage.cpp
└── storage
    ├── IndirectionVector.h
    ├── Pager.cpp
    ├── Pager.h
    ├── btree
    │   ├── Btree.cpp
    │   ├── Btree.h
    │   ├── BtreeBench.cpp
    │   ├── BtreePrinter.h
    │   ├── Node.cpp
    │   └── Node.h
    └── compression
        ├── BenchCompression.cpp
```



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 имплентира и `ISupportingInnerOperations` и `IPersistentPager`.

клас `InMemoryPager`	Конкретна имплементация на `GenericPager`, която имплементира се 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 `LRUCache`	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 B-дърво.
клас `Btree::Header`	Заглавната част на B-дърво, която съдържа метаданните на инстанцията, без потребителските данни.

клас `Btree::MemConfig`	Еквивалентна конфигурация, която да съхранява дървото в паметта, вместо на диска.
клас `Btree::PosNod`	Възел заедно с информация за неговата позиция на диска и в дървото.
клас `Btree::TreePath`	Описва път в дървото.
клас `Btree::InsertionReturnMark`	Краен резултат от извършено вмъкване.
клас `Btree::InsertionTree`	Дърво създадено по време на групово вмъкване.
клас `Btree::SearchResultMark`	Краен резултат от извършено търсене.
енумератор клас `Btree::CornerDetail`	Дескриптор за операцията, която намира ъгъл в дървото.
енумератор клас `MakeRootAction`	Означава действието

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 извличането 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, извличане и обработка на големи графове базирани данни, съдържаща модулите Pager, B-дърво, Compression и Server.
- Pager подсистемата управлява физическото пространство, върху което се разполага базата данни. B-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.

Използвана литература

[1] C++ Reference, en.cppreference.com/w/cpp, 2022

[2] Codd E. F. , “A Relational Model of Data for Large Shared Data Banks”, 1970

[3] Bayer R. , “Organization and maintenance of large ordered indices”, 1970

[4] Lynch, “Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services”, 2002

[5] Vogels W., "Eventually consistent", 2009

[6] Graefe G., “A Survey of B-Tree Locking Techniques”, 2010

[7] Comer, D., “The Ubiquitous B-Tree”, 1979

[8] Tarjan R., “Deletion Without Rebalancing in Multiway Search Trees”, 2009

[9] Lehman P., “Efficient Locking for Concurrent Operations on B-trees”, 1981

[10] Knuth D., “The art of computer programming, Vol. 3, Searching and sorting”, 1972

[11] Pagh R., “Cuckoo hashing”, 2001

[12] Herlihy M., “Hopscotch hashing”, 2008

[13] Windley P. F., “Trees, Forests and Rearranging”, 1960

[14] Adelson-Velsky G., Landis E., "An algorithm for the organization of information", 1962

[15] Наков Пр, Добриков П., “Програмиране = ++Алгоритми”, ISBN 954-8905-06-X, 2003

[16] Libnop, https://github.com/google/libnop, 2022

[17] Catch2, https://github.com/catchorg/Catch2, 2022

[18] Google benchmark, https://github.com/google/benchmark, 2022

[19] CppCoro, https://github.com/lewissbaker/cppcoro, 2022

[20] Fmt, https://github.com/fmtlib/fmt, 2022

[21] Noria - “Fast web applications through dynamic, partially stateful dataflow” in Rust, github.com/mit-pdos/noria, 2022

[22] MySQL, https://www.mysql.com/, 2022

[23] GDBM, https://www.gnu.org.ua/software/gdbm/, 2022

[24] Redis, https://github.com/redis/redis, 2022

[25] Conan, https://conan.io, 2022

[26] Huffman, D., “A Method for the Construction of Minimum Redundancy Codes”, 1952

[27] Cormen T., Leiserson C., Rivest R., Stein C., “Introduction to Algorithms”, Second Edition, 2001, ISBN 0-262-03293-7

[28] S. Sen, R. E. Tarjan, “Deletion without rebalancing in multiway search trees”, 2014

[29] Gray J., Reuter A., “Transaction Processing: Concepts and Techniques”, 1993

[30] Maier D., Salveter. S., “Hysterical B-trees”, 1981

[31] Tarjan R., “Amortized Computational Complexity”, 1985

[32] Pollari-Malmi K., Soisalon-Soininen E., “Concurrency Control and I/O-Optimality in Bulk Insertion”, 2004

[33] C++ Serialization, https://isocpp.org/wiki/faq/serialization, 2022

[34] Drepper U., “What Every Programmer Should Know About Memory”, 2007

[35] O'Neil P., Gerhard W., “The LRU-K Page Replacement Algorithm for Database Disk Buffering”, 1993

[36] Gu X. , Ding C., “On the Theory and Potential of LRU-MRU Collaborative Cache Management”, 2011

[37] Alexandrescu A., “Modern C++ Design: Generic Programming and Design Patterns Applied”, ISBN 978-0-201-70431-0, 2001

[38] Graefe G., “Modern B-tree techniques”, 2010

[39] Conway M., "Design of a Separable Transition-diagram Compiler", 1963

[40] Chamberlin D., Raymond F.,"SEQUEL: A Structured English Query Language", 1974

[41] Git Source Control Management, https://git-scm.com, 2022

[42] Memory Management, https://www.memorymanagement.org, 2022

[43] Nimbe, P.; Ofori Frimpong, S.; Opoku, M., "An Efficient Strategy for Collision Resolution in Hash Tables", 2014

Съдържание