Оглавление

Введение

Хранение изображений — задача, стоящая перед разработчиками большинства современных информационных систем. Если число фотографий ограничено, то задача не вызывает больших сложностей: хранение фотографий в виде отдельных файлов будет оптимальным решением. Однако существуют информационные системы, для которых заранее неизвестно количество фотографий, потому что это количество может обновляться во время эксплуатации такой системы (увеличиваться). Примером таких информационных систем могут послужить социальные сети и облачные хранилища. В общем случае задача заключается в реализации возможностей хранения и поиска неструктурированных данных: помимо фотографий аналогичная задача решается для хранения аудио и видео файлов, документов, скомпилированных программ и прочих не имеющих четкой структуры данных.

Способ решения задачи зависит от требований к информационной системе. Так, если у системы есть повышенные требования по безопасности, то допускается добавление шифрования данных в обмен на скорость доступа к этим данным. В случае систем, имеющих повышенное ограничение по памяти, может использоваться сжатие данных, которое так же снижает скорость операции поиска. В данной работе будут рассматриваться тип систем, основное требование которых — наиболее быстрый поиск.

Цель работы — провести исследование и предложить метод оптимизации операции поиска в существующей NoSQL базе данных.

Чтобы достигнуть поставленной цели, требуется решить следующие задачи:

- рассмотреть существующий метод поиска в определенной базе данных;
- рассмотреть способы оптимизации поиска;

- реализовать программно предложенные способы оптимизации;
- провести сравнение результатов.

1 Аналитическая часть

1.1 Способы организации хранения изображений

Неструктурированные данные — данные, которые не соответствуют заранее определённой модели данных[1]. Изображения, как было сказано в введении, являются неструктурированными данными, поэтому способы их хранения часто отличаются от способов хранения структурированных данных.

В исследовании[2] компании Microsoft[3] приведено сравнение 2 способов хранения неструктурированных данных:

1. в виде отдельных файлов в файловой системе;
2. в виде BLOB'ов в базе данных (BLOB[4] — массив двоичных данных).

В результате исследования было выяснено, что поиск и чтение неструктурированных данных при использовании базы данных выполняется быстрее, нежели при хранении в виде файлов в файловой системе. В данной работе рассматривается формат хранения изображений в виде массивов двоичных данных в нереляционных базах данных[5].

1.2 Использование нереляционных баз данных

Использование в данной работе нереляционных баз данных обусловлено тем, что требуется хранить неструктурированные данные. Такие данные не содержат информацию о связях между собой, поэтому зачастую

каждому объекту, описывающему неструктурированные данные, ставится в соответствие некоторый ключ, по которому данный объект можно идентифицировать: это может быть имя файла в системе (если данные хранятся в виде файлов) или сгенерированные специальным образом хэш[6] или число. Данный идентификатор в свою очередь может быть использован в качестве ссылки на изображение в других частях приложения, с помощью которой можно получать доступ к требуемому изображению.

Несмотря на то, что многие современные реляционные базы данных предоставляют возможность хранения массивов двоичных данных (например, в PostgreSQL[7] это реализовано в виде отдельного модуля "lo"[8]), у такого решения есть недостатки.

Если хранить изображения на одном сервере и в одной базе данных с остальными (структурированными) данными, то такой подход обладает минусами монолитной архитектуры:

- сложное горизонтальное масштабирование [9];
- имеется единая точка отказа[10];
- в большинстве случаев структурированные данные вынуждены хранится на ёмких, но более медленных накопителях, ввиду больших требований к объему памяти для неструктурированных данных.

Данные недостатки могут быть устранены за счет использование нескольких серверов: структурированные данные находятся на одном сервере и хранят идентификатор для неструктурированных данных, которые хранятся на другом сервере, и запрашивают их с другого сервера, используя идентификатор. Такой подход использует принципы сервер-ориентированной архитектуры. При таком проектировании сервер с неструктурированными данными хранит пары «ключ-значение», и использование реляционных БД в таком случае избыточно в сравнении с нереляционными БД (в частности БД «ключ-значение»[11]).

Таким образом, в качестве NoSQL БД для хранения изображений в данной работе будут рассматриваться БД «ключ-значение».

1.3 Базы данных «ключ-значение»

База данных «ключ-значение» — парадигма хранения данных, предназначенная для хранения, извлечения и управления ассоциативными массивами (структура данных, более известная сегодня как словарь или хеш-таблица) [11]. Примером таких баз данных могут послужить Redis [12] и Tarantool [13].

Базы данных «ключ-значение» можно разделить на 2 группы:

1. **база данных, которая полностью размещается в памяти** (англ. *"in-memory database"*) — такая база данных использует для хранения только оперативную память сервера;
2. **постоянная база данных** (англ. *"on-disk database"* или *"persitent database"*) — база данных, хранящая большую часть данных в файлах.

Несмотря на то, что база данных, которая полностью размещается в памяти, обладает более высокой скоростью чтения и записи данных [14], использования таких баз данных в рамках поставленной задачи невозможно из-за следующих причин:

- при перезагрузке сервера все данные будут потеряны;
- объем данных ограничен объемом оперативной памяти.

Ввиду этих недостатков использование базы данных, которая полностью размещается в памяти, невозможно, в связи с чем будет использоваться постоянная база данных. Среди постоянных баз данных большинство ориентировано на работу с SSD накопителями [15]. Примером таких баз данных могут послужить RocksDB[16], sled[17]. Однако у SSD накопителей есть недостаток: ограниченное число циклов перезаписи. В связи с

этим в данной работе рассматривается база данных ориентированная на работу с HDD[18].

В качестве исследуемой и улучшаемой базы данных используется база данных «ключ-значение» Pearl [19], удовлетворящая всем описанным выше требованиям:

- является постоянной базой данных;
- ориентирована на работу с HDD.

1.4 База данных Pearl

Для анализа и оптимизации поиска требуется:

1. рассмотреть архитектуру базы данных;
2. проанализировать текущий алгоритм поиска;
3. рассмотреть альтернативы и выбрать наиболее подходящую.

1.4.1 Архитектура

На рисунке 1.1 представлена UML-диаграмма, описывающая архитектуру Pearl.

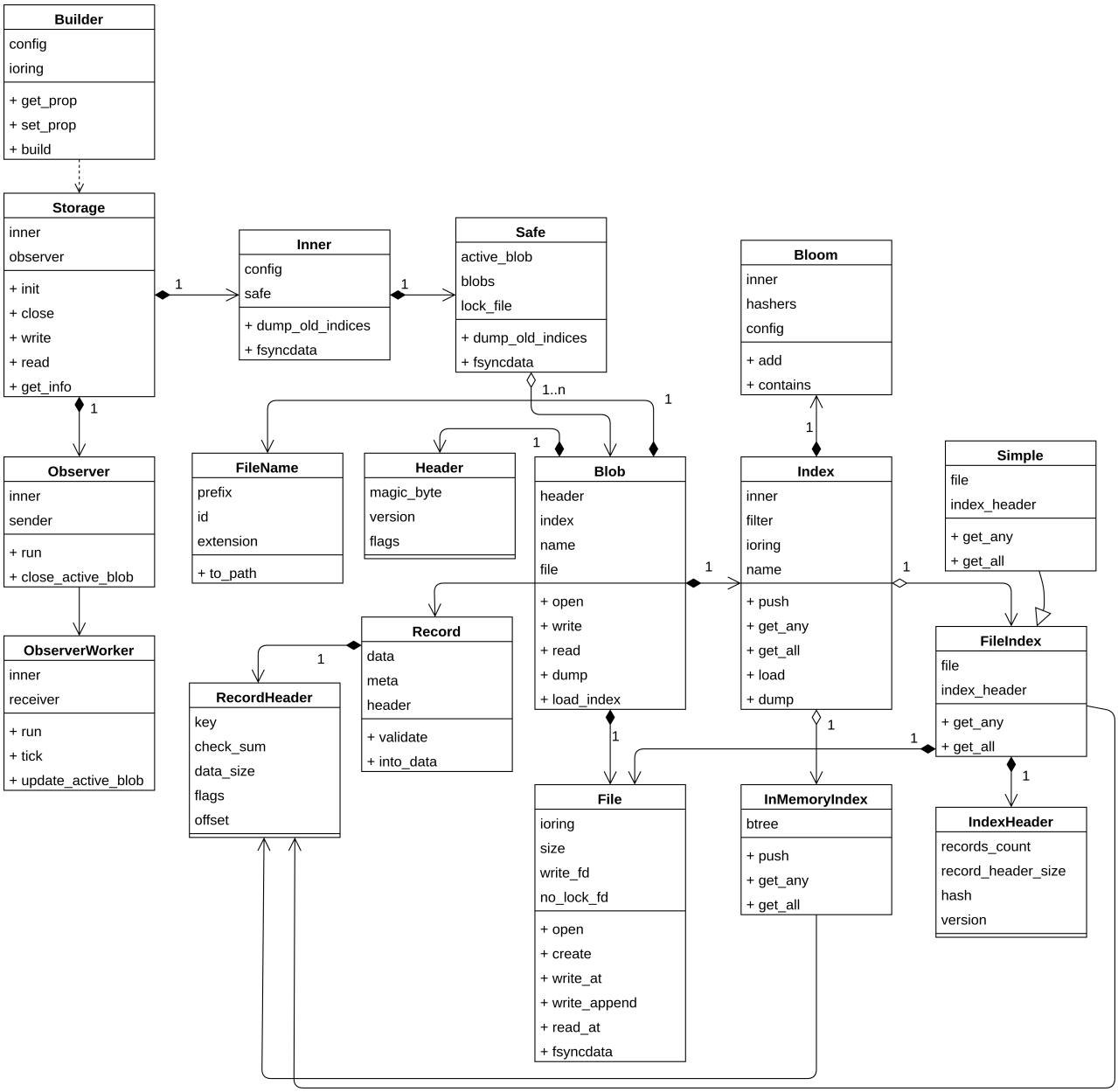

Рис. 1.1: Архитектура Pearl

1.4.2 Текущий алгоритм поиска

При запросе данных по ключу (как уже было описано ранее, в качестве данных выступают массивы бинарных данных, описывающие изображения) есть 2 варианта:

1. найти все данные с заданным ключом;

2. найти первую попавшуюся запись с заданным ключом.

Разница в алгоритмах поиска в этих 2 вариантах заключается в том, что в 1 варианте, в случае нахождения записи с искомым ключом поиск прекращается, в то время как в случае поиска всех записей, проверяются все `Blob` объекты. Алгоритм поиска в отдельном объекте `Blob` одинаковый для обоих вариантов, поэтому рассмотрение для поиска для второго варианта не снижает общность анализа.

Примечание: Pearl можно настроить таким образом, чтобы он добавлял только уникальные записи, возвращая ошибку на дубликаты, но в таком случае цена добавления возрастает (потому что для каждой записи потребуется делать еще и запрос на чтение). В таком случае возвращаемый набор записей для 2 вариантов будет одинаковым.

Объект `Blob` в документации называют просто *блобом*. Также, в Pearl вводится понятие *активного блоба*. Активный блоб — блоб, индексы которого хранятся в виде структуры `InMemoryIndex` в оперативной памяти компьютера. У блобов, которые не являются активными, индексы хранятся в файле в виде упорядоченного массива структур `RecordHeader`. Значения, хранящиеся по ключу, называют *записью*, в качестве индексов используются *заголовки записей*. В данной работе будут использоваться данные обозначения.

Общий алгоритм поиска

При поиске ключа в Pearl сначала просматриваются индексы активного блоба (структура `InMemoryIndex`). При наличии ключа среди индексов активного блоба структура возвращает заголовок записи, в котором имеется смещение и размер требуемого данного в блоб файле. В таком случае из файла с блобами по указанному смещению и размеру читаются данные, обрабатываются и возвращаются пользователю базы данных.

В случае, если в активном блобе запись с требуемым ключом не была найдена, то поиск выполняется в индексах неактивных блобов. Данные

индексы описываются структурой `FileIndex`. Если какой-то блоб содержит запись с искомым ключом, то индекс такого блоба вернет заголовок записи, что позволит из соответствующего блоб-файла считать запись, соответствующую искомому ключу. Если же после просмотра всех блобов ни один из них не вернул заголовок записи, значит, ключ отсутствует в базе данных.

Алгоритм поиска для индексов активного блоба

Как уже было сказано выше, активный блоб хранит индексы в оперативной памяти. Для хранения используется структура данных из стандартной библиотеки языка Rust[20] – `BTreeMap`[21]. Данная структура данных является B-деревом[22] — сбалансированное дерево поиска.

Данное дерево было открыто в 1970 году, и на момент написания данной работы оно используется в базах данных, например, в PostgreSQL[23].

Алгоритм поиска для индексов неактивного блоба

Для неактивного блоба индексы хранятся в файле в виде упорядоченного массива, то есть для любых 2 заголовков записей заголовок с меньшим смещением в файле соответствует записи с меньшим ключом. На рисунке 1.2 изображена структура файла с индексами.

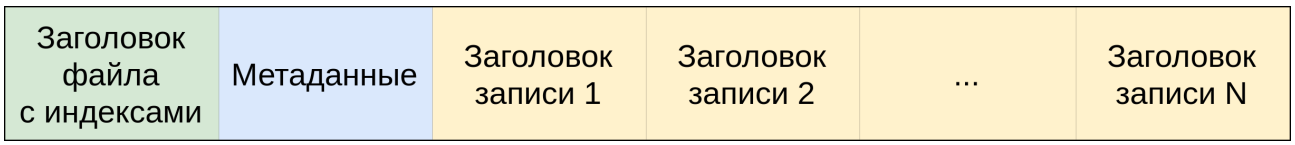

Рис. 1.2: структура файла с индексами

Поиск для неактивного блоба реализован с помощью алгоритма бинарного поиска[24].

1.4.3 Способы оптимизации алгоритма поиска

В рассмотренном алгоритме поиска узким местом является поиск индексов на диске: в то время как в хранилище может быть только один активный блоб, неактивных блобов может быть много больше одного, что приведет к тому, что в каком-то состоянии системы поиск в неактивных блобах будет занимать наибольшую часть времени поиска, в связи с чем требуется проанализировать способы оптимизации хранения и поиска индексов в файле.

В качестве альтернативы хранения и поиска индексов в файле можно предложить 2 структуры данных:

- Хэш-таблица;
- B^+-дерево.

Хранение в виде хэш-таблицы

При организации индексов на основе хэш-таблицы ключи преобразуются в хэши[6]. Хэшом называют некоторое значение, получаемое для ключа как значение некоторой функции, называемой хэш-функцией. Вычисленное значение определенным образом может быть преобразовано в указатель на некоторую часть хэш-таблицы, хранящую нужную запись. Амортизированная сложность[25] поиска для таких индексов — O(1).

На рисунке 1.3 изображен пример простой хэш-таблицы, где в качестве хэш-функции для ключей-чисел применяется функция

$$f(x) = mod_5(17 \cdot x + 11)$$

хэш-таблица представлена массивом, а получаемое значение служит индексом в этом массиве.

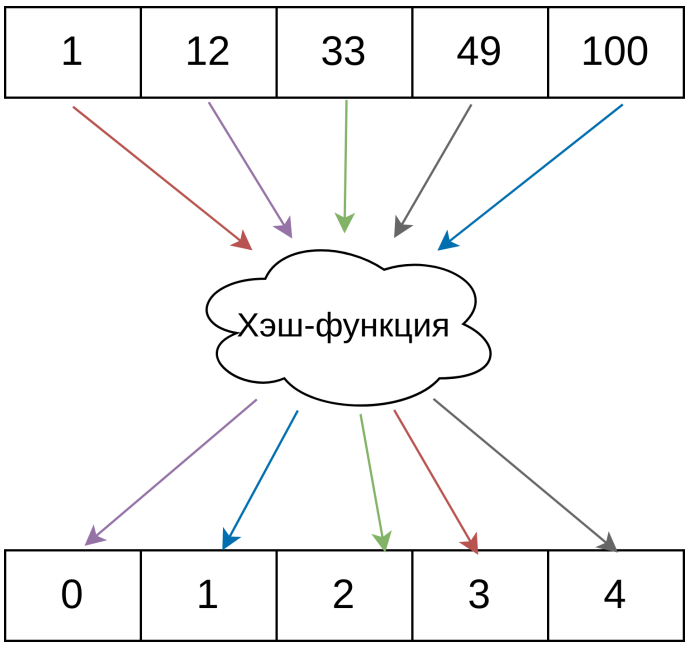

Рис. 1.3: Пример хэш-таблицы

Однако у хэш-таблиц имеется ряд ограничений:

- отсутствие возможности поиска в интервале (потому что последовательность, получающаяся после применения хэш-функцией может не быть упорядоченной по возрастанию ключей);
- отсутствие возможности выборки по префиксу (потому что в общем случае хэш считается для всего ключа);
- возможные длительные вычисления хэш-функции;
- возможные коллизии[6].

Индексы на основе хэш-таблиц сегодня используются в некоторых «key-value» базах данных, например, в Redis[12].

Хранение в виде B$^+$-дерева

Для сохранения упорядоченности ключей часто используется B-дерево и его модификации, одной из которых является B$^+$-дерево. К идее B$^+$-дерева можно прийти итеративными шагами, начиная с бинарного дерева.

m-way-деревом называют дерево поиска, узлы которого могут иметь до m потомков[26]. оно обладает теми же свойствами, что и бинарное дерево, которое является частным случаем m-way-дерева с параметром $m = 2$. На рисунке 1.4 изображен пример m-way-дерева с параметром $m = 4$.

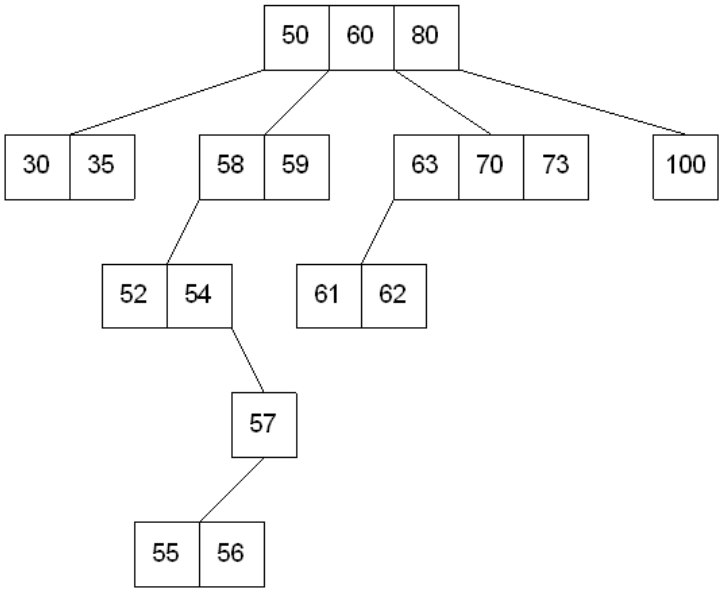

Рис. 1.4: Пример m-way-дерева с параметром $m = 4$

B-дерево – m-way-дерево, обладающее следующими дополнительными свойствами:

- корень имеет как минимум 2 поддерева (или в дереве только 1 вершина);
- каждая вершина, которая не является листом или корнем, должна иметь от $\lceil \frac{m}{2} \rceil$ до m детей;
- количество ключей у каждой вершины на 1 меньше, чем количество ее непустых детей;
- все листья находятся на одном уровне.

На рисунке 1.5 приведен пример B-дерева с максимальным числом детей равным 5.

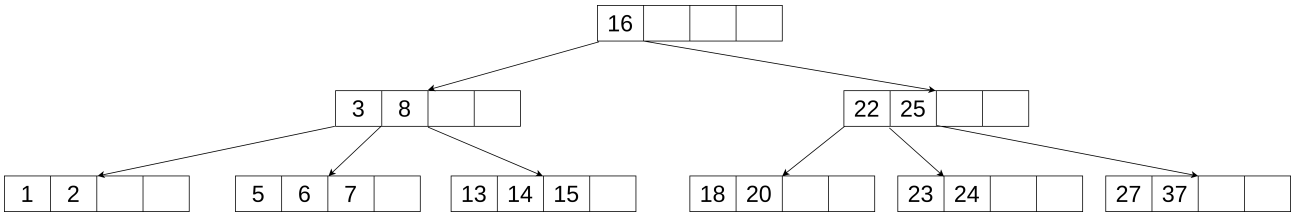

Рис. 1.5: Пример B-дерева с максимальным числом детей равным 5

Высота B-дерева в худшем случае[27]:

$$\overline{height}_{btree} = log_{\lceil \frac{m}{2} \rceil - 1} \left(\frac{n+1}{2} \right) \tag{1.1}$$

В то время как высота бинарного дерева в лучшем случае:

$$height_{binary} = \lfloor log_2(n) \rfloor + 1 \tag{1.2}$$

При $m >= 7$ B-дерево в худшем случае имеет высоту не больше, чем бинарное дерево в лучшем случае (а при достаточно больших n имеет сильно меньшую высоту).

Идея эффективности B-дерева в сравнении с обычным бинарным деревом поиска (или бинарным поиском в массиве) заключается в сокращении числа медленных операций чтения с диска: данные читаются блоками (вершинами, которые содержат до $m - 1$ ключей) и так как высота B-дерева меньше, чем высота бинарного дерева поиска, то и медленных операций чтения будет меньше, что приводит к более быстрой скорости работы.

B^+-дерево похоже на B-дерево, однако имеет важные отличия:

- вся информация, связанная с ключом, хранится в листовых вершинах;
- листовые вершины связаны друг с другом в связный список.

На рисунке 1.6 приведен пример B^+-дерева с максимальным числом детей равным 5.

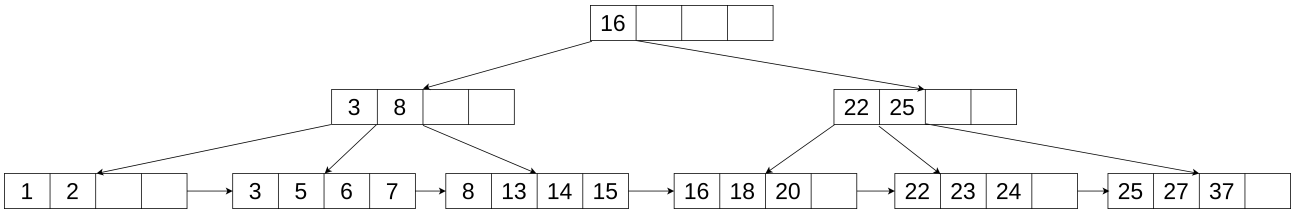

Рис. 1.6: Пример B$^+$-дерева с максимальным числом детей равным 5

Высота B$^+$-дерева в худшем случае[28]:

$$\overline{height}_{bptree} = log_{\lceil \frac{m}{2} \rceil}(n) \tag{1.3}$$

Важно отметить некоторые преимущества и недостатки B$^+$-дерева.

Преимущества:

- вершины, которые не являются листьями, содержат только информацию, связанную с деревом (не содержат информацию, связанную с ключом), что при условии ограниченности размера блока памяти для вершины, приводит к тому, что вершины B$^+$-дерева могут содержать большее количество ключей.
- возможность работы с интервалами ключей (ввиду связанных листовых вершин и содержания в листьях всех ключей);

Недостатки:

- вершины, которые не являются листьями, входят в дерево дважды (в общем случае при одинаковом количестве детей у вершин высота B+-дерева больше, чем высота B-дерева).

Вывод

В данном разделе были рассмотрены:

- способы хранения неструктурированных данных на сервере;
- базы данных, предоставляющие возможность хранения неструктурированных данных;
- база данных Pearl, ее архитектура и алгоритм поиска записи по ключу;
- способы оптимизации поиска в базе данных Pearl.

Было выяснено, что для ускорения операции поиска в базе данных Pearl нужно оптимизировать поиск в неактивных блобах, и было предложено 2 варианта оптимизации хранения заголовков в файлах индексов для неактивных блобов, один из которых (B^+-дерево) позволяет сохранить свойство упорядоченности хранения. Учитывая также все перечисленные в разделе преимущества, в данной работе для оптимизации операции поиска реализуется хранение в виде B^+-дерева.

2 Конструкторская часть

В данном разделе представлены требования к коду оптимизации, а также схемы алгоритмов, выбранных для решения поставленной задачи.

2.1 Требования к коду оптимизации

Код оптимизации должен предоставлять тот же функционал, что и код, который он заменяет. Учитывая архитектуру базы данных Pearl, требуется реализовать алгоритмы сериализации и поиска данных.

При этом предъявляются следующие требования:

- код должен корректно работать и выдавать тот же ответ, что и заменямый код, на любых входных данных;
- должны существовать условия, выполнимые на современных компьютерах, при которых код оптимизации будет работать быстрее.

2.2 Разработка алгоритмов

2.2.1 Алгоритм поиска

На рисунке 2.1 представлена схема алгоритма поиска данных при хранении в формате B$^+$-дерева.

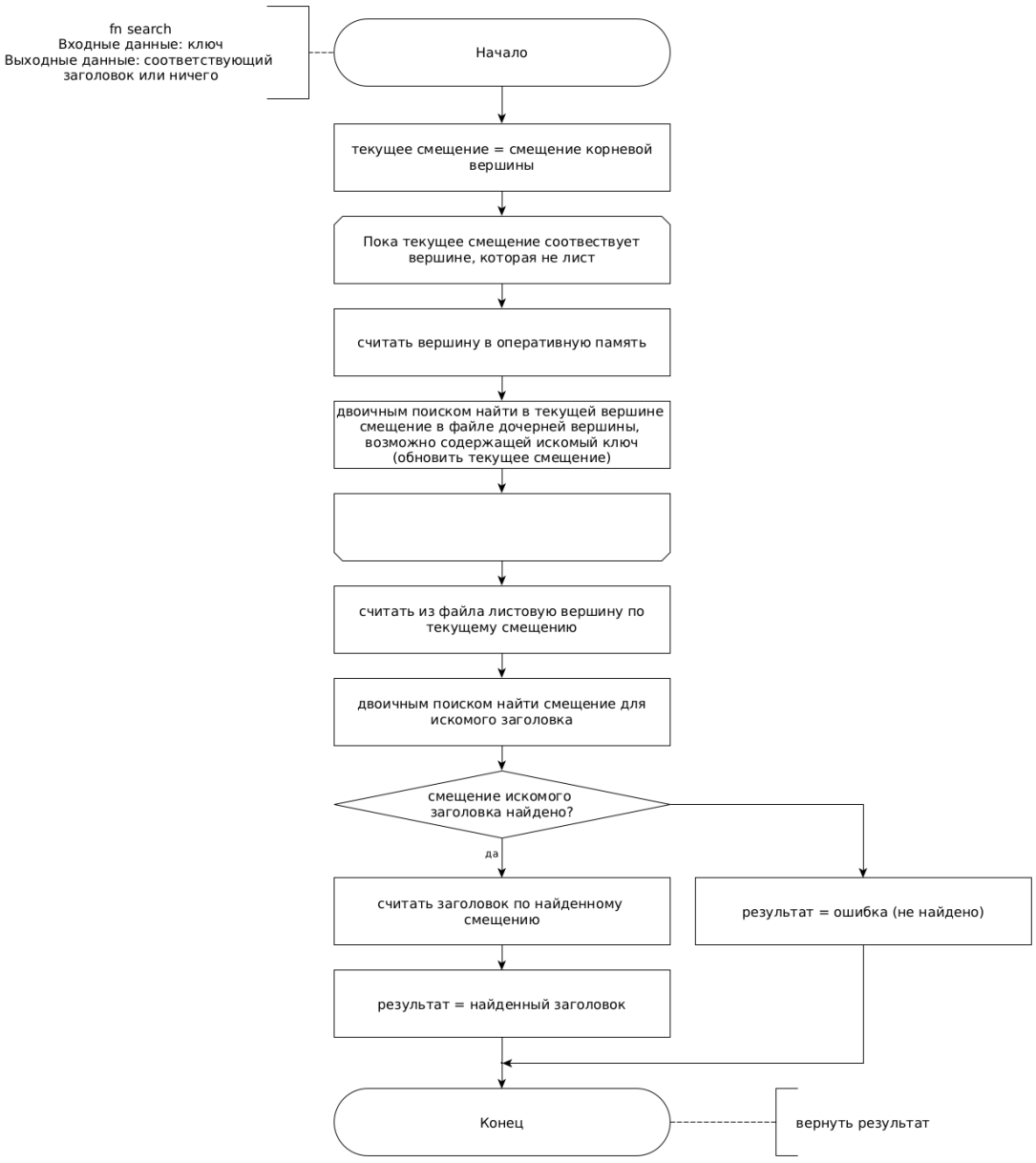

Рис. 2.1: Схема алгоритма поиска данных при хранении в формате B$^+$-дерева

2.2.2 Алгоритм сериализации

На рисунке 2.2 представлена схема алгоритма сериализации данных для хранения в формате B$^+$-дерева.

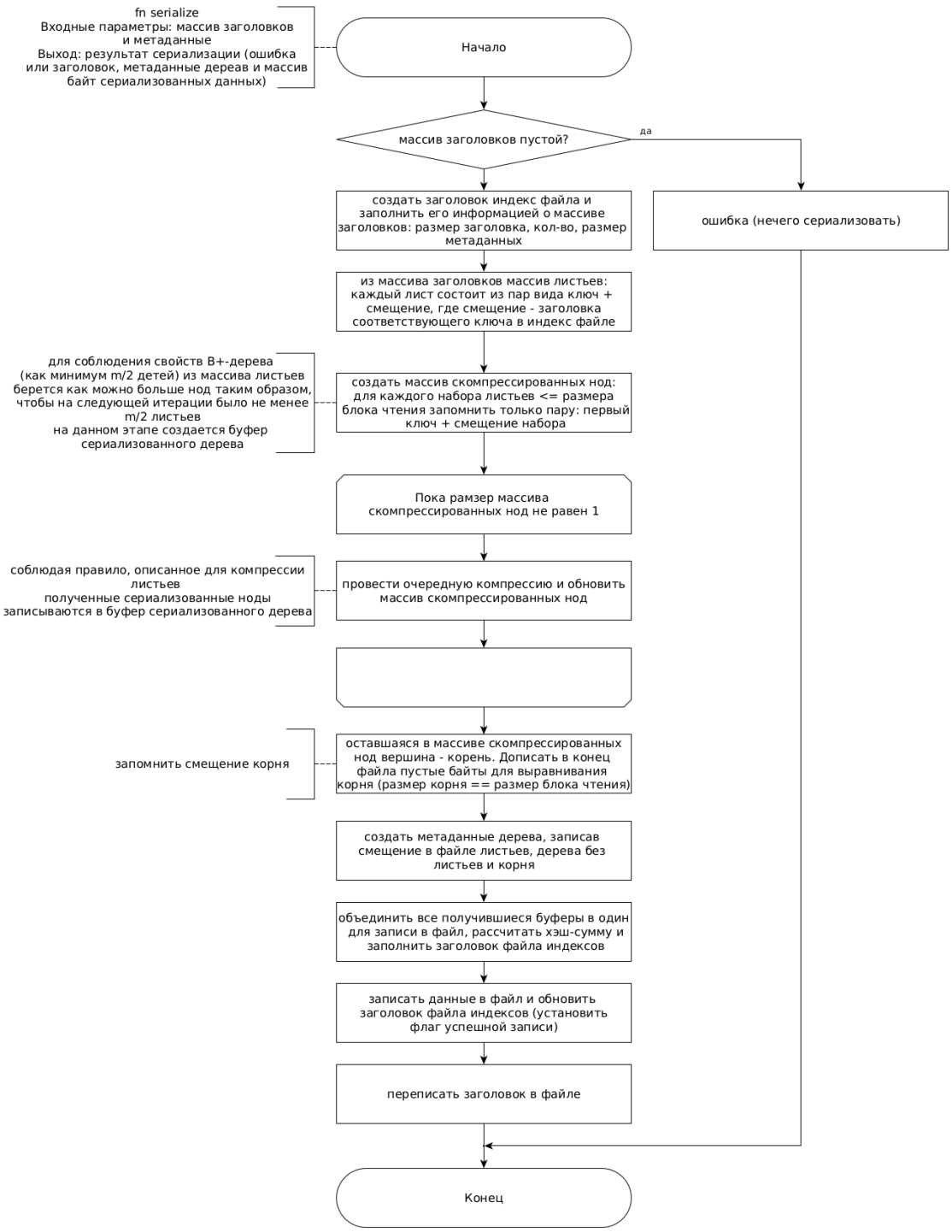

Рис. 2.2: Схема алгоритма сериализации данных для хранения в формате B$^+$-дерева

Вывод

В данном разделе были представлены требования к коду оптимизации и разработаны схемы реализуемых алгоритмов.

3 Технологическая часть

В данном разделе представлены средства разработки программного обеспечения, детали реализации и тестирование функций.

3.1 Средства реализации

В качестве языка программирования для реализации кода оптимизации использовался Rust[20]. Данный выбор обусловлен тем, что проект, поиск в котором требуется оптимизировать, написан на данном языке.

Для тестирования программного обеспечения были использованы инструменты пакетного менеджера Cargo[29], поставляемого вместе с компилятором языка при стандартном способе установке, описанном на официальном сайте языка[20].

В процессе разработки был использован инструмент RLS[30] (англ. *Rust Language Server*), позволяющий форматировать исходные коды, а также в процессе их написания обнаружить наличие синтаксических ошибок и некоторых логических, таких как, например, нарушение правила владения[31].

В качестве среды разработки был выбран текстовый редактор VIM[32], поддерживающий возможность установки плагинов[33], в том числе для работы с RLS[30].

3.2 Реализация алгоритмов

В листинге 3.1 представлена реализация алгоритма сериализации данных в формат хранения в виде B^+-дерева. В листинге 3.2 представлена реализация алгоритма поиска данных в B^+-дереве.

Листинг 3.1: Реализация алгоритма сериализации данных

```rust
/* FileIndex concrete implementation methods */
#[async_trait::async_trait]
impl FileIndexTrait for BPTreeFileIndex {
    /* ... */
    fn serialize(
        headers_btree: &InMemoryIndex,
        meta: Vec<u8>,
    ) -> Result<(IndexHeader, TreeMeta, Vec<u8>)> {
        Serializer::new(headers_btree)
            .header_stage(meta)?
            .leaves_stage()?
            .tree_stage()?
            .build()
    }
}

/* Serializer */

pub(super) struct HeaderStage<'a> {
    headers_btree: &'a InMemoryIndex,
    header: IndexHeader,
    key_size: usize,
    meta: Vec<u8>,
}

pub(super) struct LeavesStage<'a> {
    headers_btree: &'a InMemoryIndex,
    header: IndexHeader,
    leaf_size: usize,
    leaves_buf: Vec<u8>,
    leaves_offset: u64,
    headers_size: usize,
    meta: Vec<u8>,
}

pub(super) struct TreeStage<'a> {
    headers_btree: &'a InMemoryIndex,
    metadata: TreeMeta,
    header: IndexHeader,
    meta_buf: Vec<u8>,
    tree_buf: Vec<u8>,
    leaves_buf: Vec<u8>,
    headers_size: usize,
    meta: Vec<u8>,
}

pub(super) struct Serializer<'a> {
```

```
48      headers_btree: &'a InMemoryIndex,
49  }
50
51  impl<'a> Serializer<'a> {
52      pub(super) fn new(headers_btree: &'a InMemoryIndex) -> Self {
53          Self { headers_btree }
54      }
55      pub(super) fn header_stage(self, meta: Vec<u8>) -> Result<HeaderStage<'a>> {
56          if let Some(record_header) = self.headers_btree.values().next().and_then(|v|
                v.first()) {
57              let record_header_size = record_header.serialized_size().try_into()?;
58              let headers_len = self
59                  .headers_btree
60                  .iter()
61                  .fold(0, |acc, (_k, v)| acc + v.len());
62              let header = IndexHeader::new(record_header_size, headers_len, meta.len());
63              let key_size = record_header.key().len();
64              Ok(HeaderStage {
65                  headers_btree: self.headers_btree,
66                  header,
67                  key_size,
68                  meta,
69              })
70          } else {
71              Err(anyhow!("BTree is empty, can't find info about key len!"))
72          }
73      }
74  }
75
76  impl<'a> HeaderStage<'a> {
77      pub(super) fn leaves_stage(self) -> Result<LeavesStage<'a>> {
78          let hs = self.header.serialized_size()? as usize;
79          let fsize = self.header.meta_size;
80          let msize: usize = TreeMeta::serialized_size_default()? as usize;
81          let headers_start_offset = (hs + fsize) as u64;
82          let headers_size = self.header.records_count * self.header.record_header_size;
83          let leaves_offset = headers_start_offset + (headers_size + msize) as u64;
84          // leaf contains pair: (key, offset in file for first record's header with this
                key)
85          let leaf_size = self.key_size + std::mem::size_of::<u64>();
86          let leaves_buf = Self::serialize_leaves(
87              self.headers_btree,
88              headers_start_offset,
89              leaf_size,
90              self.header.record_header_size,
91          );
92          Ok(LeavesStage {
93              headers_btree: self.headers_btree,
```

```
            leaf_size,
            leaves_buf,
            leaves_offset,
            headers_size,
            meta: self.meta,
            header: self.header,
        })
    }

    fn serialize_leaves(
        headers_btree: &InMemoryIndex,
        headers_start_offset: u64,
        leaf_size: usize,
        record_header_size: usize,
    ) -> Vec<u8> {
        let mut leaves_buf = Vec::with_capacity(leaf_size * headers_btree.len());
        headers_btree.iter().fold(
            (headers_start_offset, &mut leaves_buf),
            |(offset, leaves_buf), (leaf, hds)| {
                leaves_buf.extend_from_slice(&leaf);
                let offsetb = serialize(&offset).expect("serialize u64");
                leaves_buf.extend_from_slice(&offsetb);
                let res = (offset + (hds.len() * record_header_size) as u64, leaves_buf);
                res
            },
        );
        leaves_buf
    }
}

impl<'a> LeavesStage<'a> {
    pub(super) fn tree_stage(self) -> Result<TreeStage<'a>> {
        let tree_offset = self.leaves_offset + self.leaves_buf.len() as u64;
        let (root_offset, tree_buf) = Self::serialize_bptree(
            self.headers_btree,
            self.leaves_offset,
            self.leaf_size as u64,
            tree_offset,
        )?;
        let metadata = TreeMeta::new(root_offset, self.leaves_offset, tree_offset);
        let meta_buf = serialize(&metadata)?;
        Ok(TreeStage {
            headers_btree: self.headers_btree,
            metadata,
            meta_buf,
            tree_buf,
            header: self.header,
            headers_size: self.headers_size,
```

```
142             leaves_buf: self.leaves_buf,
143             meta: self.meta,
144         })
145     }
146 
147     fn key_from_iter<'k>(keys_iter: &mut impl Iterator<Item = &'k Vec<u8>>) ->
        Result<Vec<u8>> {
148         if let Some(key) = keys_iter.next() {
149             Ok(key.clone())
150         } else {
151             Err(anyhow!("Unexpected␣end␣of␣keys␣sequence"))
152         }
153     }
154 
155     fn serialize_bptree(
156         btree: &InMemoryIndex,
157         leaves_offset: u64,
158         leaf_size: u64,
159         tree_offset: u64,
160     ) -> Result<(u64, Vec<u8>)> {
161         let max_amount = Self::max_leaf_node_capacity(btree.keys().next().unwrap().len());
162         let min_amount = (max_amount - 1) / 2 + 1;
163         let mut leaf_nodes_compressed = Vec::new();
164         let mut offset = leaves_offset;
165         let mut keys_iter = btree.keys();
166         let elems_amount = btree.len();
167         let mut current = 0;
168         while elems_amount - current > max_amount {
169             let amount = std::cmp::min(max_amount, elems_amount - current - min_amount);
170             leaf_nodes_compressed.push((Self::key_from_iter(&mut keys_iter)?, offset));
171             (0..(amount - 1)).for_each(|_| {
172                 keys_iter.next();
173             });
174             offset += amount as u64 * leaf_size;
175             current += amount;
176         }
177         leaf_nodes_compressed.push((Self::key_from_iter(&mut keys_iter)?, offset));
178 
179         let mut buf = Vec::new();
180         let root_offset = Self::build_tree(leaf_nodes_compressed, tree_offset, &mut buf)?;
181         Ok((root_offset, buf))
182     }
183 
184     fn process_keys_portion(
185         buf: &mut Vec<u8>,
186         tree_offset: u64,
187         nodes_portion: &[(Vec<u8>, u64)],
188     ) -> Result<(Vec<u8>, u64)> {
```

```
189         let offset = tree_offset + buf.len() as u64;
190         let min_key = nodes_portion[0].0.clone();
191         let offsets_iter = nodes_portion.iter().map(|(_, offset)| *offset);
192         let keys_iter = nodes_portion[1..].iter().map(|(k, _)| k.as_ref());
193         let node_buf = Node::new_serialized(
194             keys_iter,
195             offsets_iter,
196             nodes_portion[0].0.len(),
197             nodes_portion.len() - 1,
198         )?;
199         buf.extend_from_slice(&node_buf);
200         Ok((min_key, offset))
201     }
202 
203     pub(super) fn build_tree(
204         nodes_arr: Vec<(Vec<u8>, u64)>,
205         tree_offset: u64,
206         buf: &mut Vec<u8>,
207     ) -> Result<u64> {
208         if nodes_arr.len() == 1 {
209             return Ok(Self::prep_root(nodes_arr[0].1, tree_offset, buf));
210         }
211         let max_amount = Self::max_nonleaf_node_capacity(nodes_arr[0].0.len());
212         let min_amount = (max_amount - 1) / 2 + 1;
213         let mut current = 0;
214         let mut new_nodes = Vec::new();
215         while nodes_arr.len() - current > max_amount {
216             // amount == min_amount at least (if nodes_arr.len() - current == max_amount
                + 1)
217             // and min operation is necessary to have at least min_amount nodes left
218             let amount = std::cmp::min(max_amount, nodes_arr.len() - current -
                min_amount);
219             let nodes_portion = &nodes_arr[current..(current + amount)];
220             current += amount;
221             let compressed_node = Self::process_keys_portion(buf, tree_offset,
                nodes_portion)?;
222             new_nodes.push(compressed_node);
223         }
224         // min_amount <= nodes left <= max_amount
225         let nodes_portion = &nodes_arr[current..];
226         new_nodes.push(Self::process_keys_portion(
227             buf,
228             tree_offset,
229             &nodes_portion,
230         )?);
231         Self::build_tree(new_nodes, tree_offset, buf)
232     }
233 
```

```
234     fn max_leaf_node_capacity(key_size: usize) -> usize {
235         let offset_size = std::mem::size_of::<u64>();
236         BLOCK_SIZE / (key_size + offset_size) - 1
237     }
238 
239     fn max_nonleaf_node_capacity(key_size: usize) -> usize {
240         let offset_size = std::mem::size_of::<u64>();
241         let meta_size =
242             NodeMeta::serialized_size_default().expect("Can't␣retrieve␣default␣serialized␣
                size");
243         (BLOCK_SIZE - meta_size as usize - offset_size) / (key_size + offset_size) + 1
244     }
245 
246     fn prep_root(offset: u64, tree_offset: u64, buf: &mut Vec<u8>) -> u64 {
247         let root_node_size = if offset < tree_offset {
248             (tree_offset - offset) as usize
249         } else {
250             buf.len() - (offset - tree_offset) as usize
251         };
252         let appendix_size = BLOCK_SIZE - root_node_size;
253         buf.resize(buf.len() + appendix_size, 0);
254         offset
255     }
256 }
257 
258 impl<'a> TreeStage<'a> {
259     pub(super) fn build(self) -> Result<(IndexHeader, TreeMeta, Vec<u8>)> {
260         let hs = self.header.serialized_size()? as usize;
261         let fsize = self.header.meta_size;
262         let msize = self.meta_buf.len();
263         let data_size =
264             hs + fsize + self.headers_size + msize + self.leaves_buf.len() +
                self.tree_buf.len();
265         let mut buf = Vec::with_capacity(data_size);
266         serialize_into(&mut buf, &self.header)?;
267         buf.extend_from_slice(&self.meta);
268         Self::append_headers(self.headers_btree, &mut buf)?;
269         buf.extend_from_slice(&self.meta_buf);
270         buf.extend_from_slice(&self.leaves_buf);
271         buf.extend_from_slice(&self.tree_buf);
272         let hash = get_hash(&buf);
273         let header = IndexHeader::with_hash(
274             self.header.record_header_size,
275             self.header.records_count,
276             self.meta.len(),
277             hash,
278         );
279         serialize_into(buf.as_mut_slice(), &header)?;
```

```
280         Ok((header, self.metadata, buf))
281     }
282
283     fn append_headers(headers_btree: &InMemoryIndex, buf: &mut Vec<u8>) -> Result<()> {
284         headers_btree
285             .iter()
286             .flat_map(|r| r.1)
287             .map(|h| serialize(&h))
288             .try_fold(buf, |buf, h_buf| -> Result<_> {
289                 buf.extend_from_slice(&h_buf?);
290                 Ok(buf)
291             })?;
292         Ok(())
293     }
294 }
```

Листинг 3.2: Реализация алгоритма поиска данных

```
1  #[derive(Debug, Clone)]
2  pub(crate) struct BPTreeFileIndex {
3      file: File,
4      header: IndexHeader,
5      metadata: TreeMeta,
6      root_node: [u8; BLOCK_SIZE],
7  }
8
9  struct Node {/* fields omitted */}
10
11 impl Node {
12     pub(super) fn key_offset_serialized(buf: &[u8], key: &[u8]) -> Result<u64> {
13         let meta_size = NodeMeta::serialized_size_default()? as usize;
14         let node_size = deserialize::<NodeMeta>(&buf[..meta_size])?.size as usize;
15         let offsets_offset = meta_size + node_size * key.len();
16         let ind = match Self::binary_search_serialized(key,
               &buf[meta_size..offsets_offset]) {
17             Ok(pos) => pos + 1,
18             Err(pos) => pos,
19         };
20         let offset = offsets_offset + ind * size_of::<u64>();
21         deserialize(&buf[offset..(offset + size_of::<u64>())]).map_err(Into::into)
22     }
23 }
24
25 #[async_trait::async_trait]
26 impl FileIndexTrait for BPTreeFileIndex {
27     /* from_file */
28     /* from_records */
29     /* file_size */
30     /* records_count */
```

```
31      /* read_meta */
32      /* get_record_headers */
33      /* validate */
34
35      async fn find_by_key(&self, key: &[u8]) -> Result<Option<Vec<RecordHeader>>> {
36          let root_offset = self.metadata.root_offset;
37          let mut buf = [0u8; BLOCK_SIZE];
38          let leaf_offset = self.find_leaf_node(key, root_offset, &mut buf).await?;
39          if let Some((fh_offset, amount)) =
40              self.find_first_header(leaf_offset, key, &mut buf).await?
41          {
42              let headers = self.read_headers(fh_offset, amount as usize).await?;
43              Ok(Some(headers))
44          } else {
45              Ok(None)
46          }
47      }
48
49      async fn get_any(&self, key: &[u8]) -> Result<Option<RecordHeader>> {
50          let root_offset = self.metadata.root_offset;
51          let mut buf = [0u8; BLOCK_SIZE];
52          let leaf_offset = self.find_leaf_node(key, root_offset, &mut buf).await?;
53          if let Some((first_header_offset, _amount)) =
54              self.find_first_header(leaf_offset, key, &mut buf).await?
55          {
56              let header = self.read_headers(first_header_offset, 1).await?.remove(0);
57              Ok(Some(header))
58          } else {
59              Ok(None)
60          }
61      }
62  }
63
64  impl BPTreeFileIndex {
65      async fn find_leaf_node(&self, key: &[u8], mut offset: u64, buf: &mut [u8]) ->
            Result<u64> {
66          while offset >= self.metadata.tree_offset {
67              offset = if offset >= self.metadata.root_offset {
68                  Node::key_offset_serialized(&self.root_node, key)?
69              } else {
70                  self.file.read_at(buf, offset).await?;
71                  Node::key_offset_serialized(buf, key)?
72              };
73          }
74          Ok(offset)
75      }
76
77      async fn find_first_header(
```

```
78          &self,
79          leaf_offset: u64,
80          key: &[u8],
81          buf: &mut [u8],
82      ) -> Result<Option<(u64, u64)>> {
83          let leaf_size = key.len() + size_of::<u64>();
84          let buf_size = self.leaf_node_buf_size(leaf_size, leaf_offset);
85          let pointers_size = self.file.read_at(&mut buf[..buf_size], leaf_offset).await?;
86          self.search_header_pointer(&buf[..pointers_size], key, buf_size as u64 +
                leaf_offset)
87      }
88
89      fn leaf_node_buf_size(&self, leaf_size: usize, leaf_offset: u64) -> usize {
90          let buf_size = (self.metadata.tree_offset - leaf_offset) as usize;
91          buf_size
92              .min(BLOCK_SIZE - (BLOCK_SIZE % leaf_size))
93              .min((self.metadata.tree_offset - leaf_offset) as usize)
94      }
95
96      fn search_header_pointer(
97          &self,
98          header_pointers: &[u8],
99          key: &[u8],
100         absolute_buf_end: u64,
101     ) -> Result<Option<(u64, u64)>> {
102         let leaf_size = key.len() + size_of::<u64>();
103         let mut l = 0i32;
104         let mut r: i32 = (header_pointers.len() / leaf_size) as i32 - 1;
105         while l <= r {
106             let m = (l + r) / 2;
107             let m_off = leaf_size * m as usize;
108             match key.cmp(&header_pointers[m_off..(m_off + key.len())]) {
109                 CmpOrdering::Less => r = m - 1,
110                 CmpOrdering::Greater => l = m + 1,
111                 CmpOrdering::Equal => {
112                     return Ok(Some(self.with_amount(
113                         header_pointers,
114                         m as usize,
115                         absolute_buf_end,
116                         key.len(),
117                     )?));
118                 }
119             }
120         }
121         Ok(None)
122     }
123
124     fn with_amount(
```

```
        &self,
        header_pointers: &[u8],
        mid: usize,
        buf_end: u64,
        key_size: usize,
    ) -> Result<(u64, u64)> {
        let leaf_size = key_size + size_of::<u64>();
        let is_last_rec = (mid + 1) * leaf_size >= header_pointers.len()
            && (buf_end >= self.metadata.tree_offset);
        let mid_offset = mid * leaf_size + key_size;
        let cur_offset =
            deserialize::<u64>(&header_pointers[mid_offset..(mid_offset +
                size_of::<u64>())])?;
        let next_offset = if is_last_rec {
            self.metadata.leaves_offset
        } else {
            let next_offset = mid_offset + leaf_size;
            deserialize(&header_pointers[next_offset..(next_offset + size_of::<u64>())])?
        };
        let amount = (next_offset - cur_offset) / self.header.record_header_size as u64;
        Ok((cur_offset, amount))
    }

    async fn read_headers(&self, offset: u64, amount: usize) ->
        Result<Vec<RecordHeader>> {
        let mut buf = vec![0u8; self.header.record_header_size * amount];
        self.file.read_at(&mut buf, offset).await?;
        buf.chunks(self.header.record_header_size).try_fold(
            Vec::with_capacity(amount),
            |mut acc, bytes| {
                acc.push(deserialize(&bytes)?);
                Ok(acc)
            },
        )
    }

    async fn read_index_header(file: &File) -> Result<IndexHeader> {
        let header_size = IndexHeader::serialized_size_default()? as usize;
        let mut buf = vec![0; header_size];
        file.read_at(&mut buf, 0).await?;
        IndexHeader::from_raw(&buf).map_err(Into::into)
    }

    async fn read_root(file: &File, root_offset: u64) -> Result<[u8; BLOCK_SIZE]> {
        let mut buf = [0; BLOCK_SIZE];
        file.read_at(&mut buf, root_offset).await?;
        Ok(buf)
    }
```

```
171
172     async fn read_tree_meta(file: &File, header: &IndexHeader) -> Result<TreeMeta> {
173         let meta_size = TreeMeta::serialized_size_default()? as usize;
174         let mut buf = vec![0; meta_size];
175         let fsize = header.meta_size as u64;
176         let hs = header.serialized_size()?;
177         let meta_offset = hs + fsize + (header.records_count * header.record_header_size)
                as u64;
178         file.read_at(&mut buf, meta_offset).await?;
179         TreeMeta::from_raw(&buf).map_err(Into::into)
180     }
181     /* serialize */
182     /* validate_header */
183     /* hash_valid */
184 }
```

3.3 Функциональные тесты

В листинге 3.3 представлены функции для тестирования разработанного кода.

Листинг 3.3: Функциональные тесты

```
1  use super::prelude::*;
2
3  const META_SIZE: usize = 100;
4  const META_VALUE: u8 = 17;
5
6  #[tokio::test]
7  async fn serialize_deserialize_file() {
8      let mut inmem = InMemoryIndex::new();
9      (0..10000u32)
10         .map(|i| serialize(&i).expect("can't␣serialize"))
11         .for_each(|key| {
12             let rh = RecordHeader::new(key.clone(), 1, 1, 1);
13             inmem.insert(key, vec![rh]);
14         });
15     let meta = vec![META_VALUE; META_SIZE];
16     let findex =
17         BPTreeFileIndex::from_records(&Path::new("/tmp/bptree_index.b"), None, &inmem,
               meta, true)
18             .await
19             .expect("Can't␣create␣file␣index");
20     let (inmem_after, _size) = findex
```

```
        .get_records_headers()
        .await
        .expect("Can't get InMemoryIndex");
    assert_eq!(inmem, inmem_after);
}

#[tokio::test]
async fn check_get_any() {
    const RANGE_FROM: u32 = 100;
    const RANGE_TO: u32 = 9000;

    let mut inmem = InMemoryIndex::new();
    (RANGE_FROM..RANGE_TO)
        .map(|i| serialize(&i).expect("can't serialize"))
        .for_each(|key| {
            let rh = RecordHeader::new(key.clone(), 1, 1, 1);
            inmem.insert(key, vec![rh]);
        });
    let meta = vec![META_VALUE; META_SIZE];
    let findex = BPTreeFileIndex::from_records(
        &Path::new("/tmp/any_bptree_index.b"),
        None,
        &inmem,
        meta,
        true,
    )
    .await
    .expect("Can't create file index");
    let presented_keys = RANGE_FROM..RANGE_TO;
    for key in presented_keys.map(|k| serialize(&k).unwrap()) {
        assert_eq!(inmem[&key][0], findex.get_any(&key).await.unwrap().unwrap());
    }
    let not_presented_ranges = [0..RANGE_FROM, RANGE_TO..(RANGE_TO + 100)];
    for not_presented_keys in not_presented_ranges.iter() {
        for key in not_presented_keys.clone().map(|k| serialize(&k).unwrap()) {
            assert_eq!(None, findex.get_any(&key).await.unwrap());
        }
    }
}

#[tokio::test]
async fn check_get() {
    const MAX_AMOUNT: u32 = 3;
    const RANGE_FROM: u32 = 100;
    const RANGE_TO: u32 = 9000;

    let mut inmem = InMemoryIndex::new();
    (RANGE_FROM..RANGE_TO)
```

```
69          .map(|i| (i % MAX_AMOUNT + 1, serialize(&i).expect("can't serialize")))
70          .for_each(|(times, key)| {
71              let rh = RecordHeader::new(key.clone(), 1, 1, 1);
72              let recs = (0..times).map(|_| rh.clone()).collect();
73              inmem.insert(key, recs);
74          });
75      let meta = vec![META_VALUE; META_SIZE];
76      let findex = BPTreeFileIndex::from_records(
77          &Path::new("/tmp/all_bptree_index.b"),
78          None,
79          &inmem,
80          meta,
81          true,
82      )
83      .await
84      .expect("Can't create file index");
85      let presented_keys = RANGE_FROM..RANGE_TO;
86      for key in presented_keys.map(|k| serialize(&k).unwrap()) {
87          assert_eq!(
88              inmem[&key],
89              findex.find_by_key(&key).await.unwrap().unwrap()
90          );
91      }
92      let not_presented_ranges = [0..RANGE_FROM, RANGE_TO..(RANGE_TO + 100)];
93      for not_presented_keys in not_presented_ranges.iter() {
94          for key in not_presented_keys.clone().map(|k| serialize(&k).unwrap()) {
95              assert_eq!(None, findex.find_by_key(&key).await.unwrap());
96          }
97      }
98  }
```

Вывод

В данном разделе были рассмотрены средства, с помощью которых было реализовано ПО, а также представлены листинги кода с реализацией сериализации B^+-дерева и поиска в нём. Также были приведены коды, использовавшиеся для тестирования правильности работы кода.

4 Исследовательская часть

В данном разделе будут измерены и сравнены скорости работы начального и оптимизирующего кодов.

4.1 Отдельные замеры скорости работы файловых индексов

4.1.1 Технические характеристики машины, на которой проводились данные замеры

- Операционная система: Manjaro [34] Linux [35] x86_64.
- Память: 8 GiB.
- Процессор: Intel Core™ i7-8550U[36].

Тестирование проводилось на ноутбуке, включенном в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, окружением, а также непосредственно системой тестирования.

4.1.2 Замеры

В таблице 4.1 и на графике 4.1 представлено среднее время сериализации в мс для 2 реализаций файловых индексов в зависимости от количества заголовков (размер в тысячах записей). В таблице 4.2 и на графике 4.2

представлено среднее время запросов первой записи, соответствующей искомому ключу, а в таблице 4.3 и на графике 4.3 представлено среднее время запросов всех записей, соответствующих искомому ключу, в зависимости от количества заголовков (размер в тысячах записей).

Размер	B$^+$-дерево	Бинарный поиск
50	31	20
100	61	39
200	114	68
500	288	186
1000	343	309
10000	7522	4821

Таблица 4.1: Сравнение среднего времени сериализации для 2 реализаций файловых индексов, размер в тысячах, время в мс

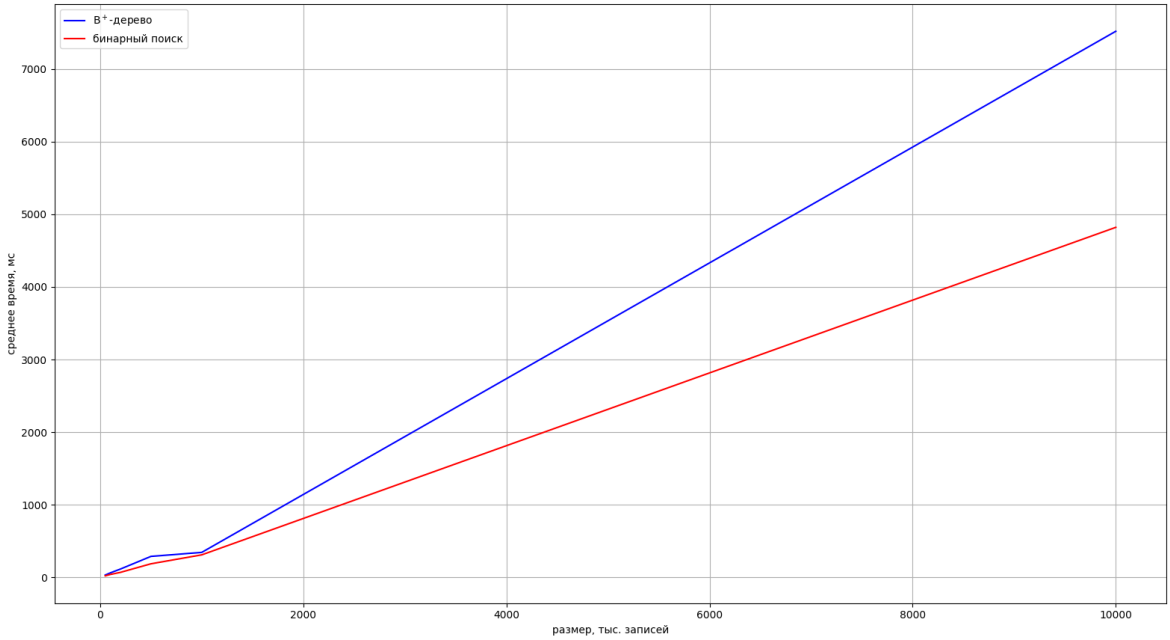

Рис. 4.1: График зависимости времени сериализации от количества записей для 2 реализаций

Размер	B$^+$-дерево	Бинарный поиск
50	11	98
100	11	110
200	18	115
500	18	135
1000	18	138
10000	19	163

Таблица 4.2: Сравнение среднего времени запросов первого заголовка для 2 реализаций файловых индексов, размер в тысячах, время в мкс

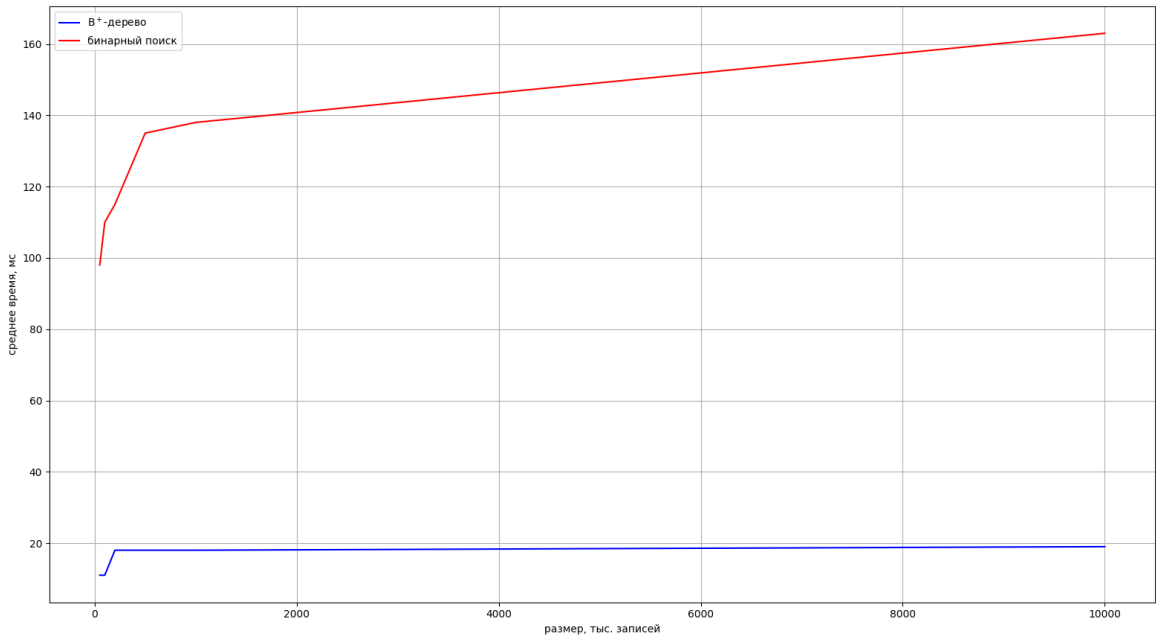

Рис. 4.2: График зависимости времени поиска от количества записей для 2 реализаций

Размер	B$^+$-дерево	Бинарный поиск
50	11	108
100	11	132
200	18	142
500	18	151
1000	18	161
10000	19	173

Таблица 4.3: Сравнение среднего времени запросов всех заголовков для 2 реализаций файловых индексов, размер в тысячах, время в мкс

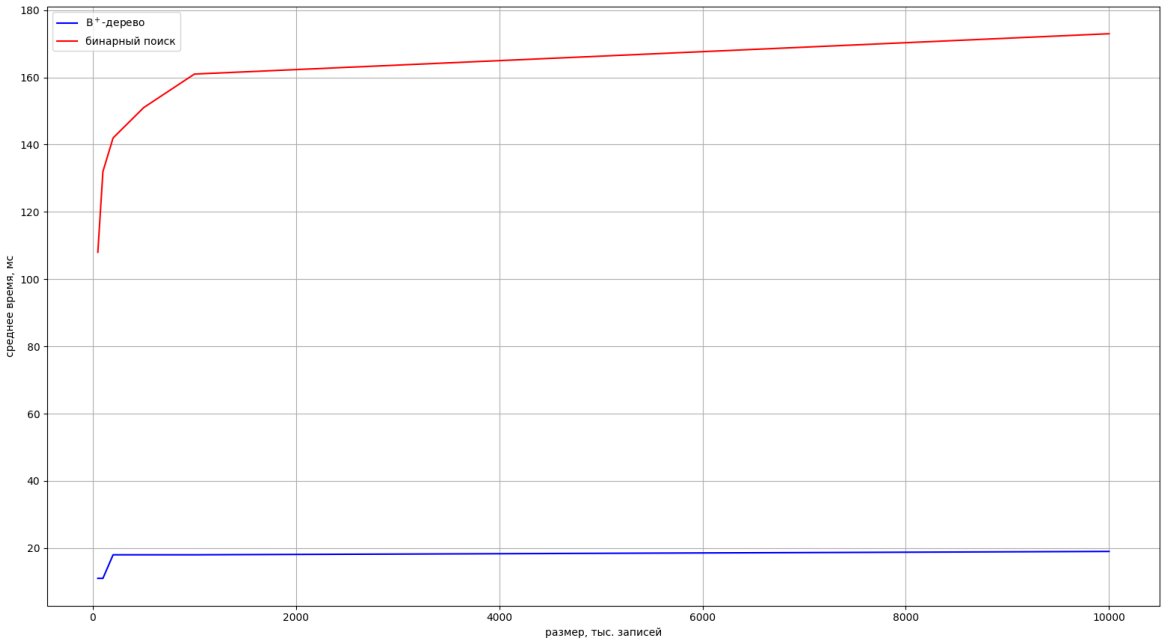

Рис. 4.3: График зависимости времени поиска всех записей для соответствующего ключа от количества записей для 2 реализаций

Вывод

В данном подразделе в некоторых случаях реализация B^+-дерева оказалась быстрее больше, чем в 9 раз. Однако данный тест не является показательным, потому что в таком эксперименте весь кэш системы в процессе вычислений находится в распоряжении индексов, поэтому B^+-дерево им более удачно пользуется. При этом процесс сериализации B^+-дерева дольше в среднем на 55% в связи с тем, что помимо сериализации упорядоченного набора заголовков строится дерево.

4.2 Замеры скорости работы в составе кластера

4.2.1 Использование кластера с Pearl

Для тестирования изменений, связанных со скоростью поиска, можно воспользоваться распределенным хранилищем, использующим в своей основе Pearl — Bob [37]. Помимо реализации логики управления кластером, данный проект содержит утилиты для его тестирования.

4.2.2 Технические характеристики сервера, на котором проводится тестирование

- Операционная система: Linux [35] x86_64.
- Память: 64 GiB.
- Процессор: Intel® Xeon® Silver 4210R CPU @ 2.40GHz[36].

4.2.3 Конфигурация кластера

Исследуемый кластер имеет следующую конфигурацию:

- один сервер;
- 10 дисков;
- максимальное количество записей в 1 блобе = 200000;
- максимальный размер блоб файла = 10 Гб;

- размер записи = 50 Кб.

Утилита тестирования будет содержать 24 клиентов, для выполнения одновременных запросов к бобу.

4.2.4 Стратегии запросов данных

Перед непосредственным выполнением запросов утилита сперва формирует заданное количество клиентов, выделяя для каждого свой асинхронный поток и диапазон ключей, которые клиент должен запросить. Диапазон ключей рассчитывается по следующему принципу: утилита получает на вход главный диапазон, который требуется запросить, затем она делит этот диапазон на m равных целых частей, где m – число клиентов (если невозможно, то последний клиент запросит на $x < m$ запросов больше) и распределяет эти диапазоны между созданными клиентами. В данном эксперименте рассматриваются 2 стратегии запросов данных:

- *случайные запросы* — каждый клиент запрашивает данные из своего диапазона случайным образом;
- *упорядоченные запросы* — клиенты запрашивают данные параллельно, однако каждый клиент запрашивает ключи из своего диапазона последовательно.

4.2.5 Замеры

В таблице 4.4 и на графике 4.4 представлено среднее количество ответов кластера в секунду (rps) для случайных запросов ключей в зависимост от размера кластера, а в таблице 4.5 и на графике 4.5 представлено среднее количество ответов кластера в секунду (rps) упорядоченных запросов ключей от размера кластера.

Размер кластера, тыс.	B^+-дерево, rps	Бинарный поиск, rps
2000	5000	3700
4000	3100	2800
6000	2650	2500
8000	2150	2050

Таблица 4.4: Среднее количество ответов кластера в секунду (rps) для случайных запросов ключей

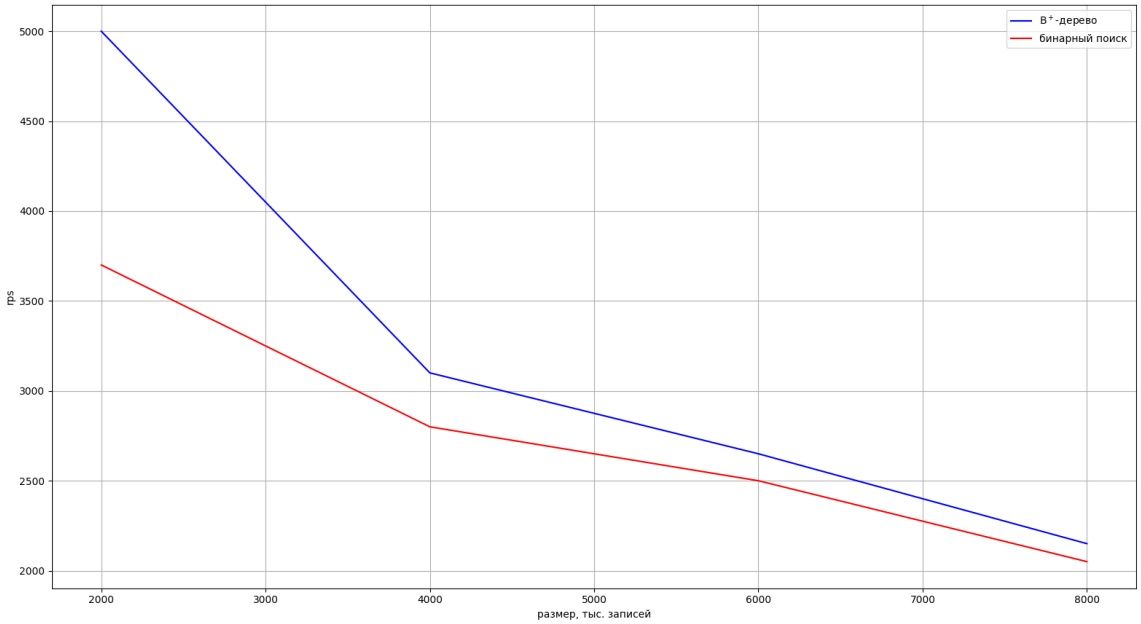

Рис. 4.4: График зависимости среднего количества ответов кластера в секунду (rps) для случайных запросов ключей от размера кластера

Размер кластера, тыс.	B^+-дерево, rps	Бинарный поиск, rps
2000	17000	12500
4000	16500	11000
6000	16000	11000
8000	16000	10500

Таблица 4.5: Среднее количество ответов кластера в секунду (rps) для упорядоченных запросов ключей

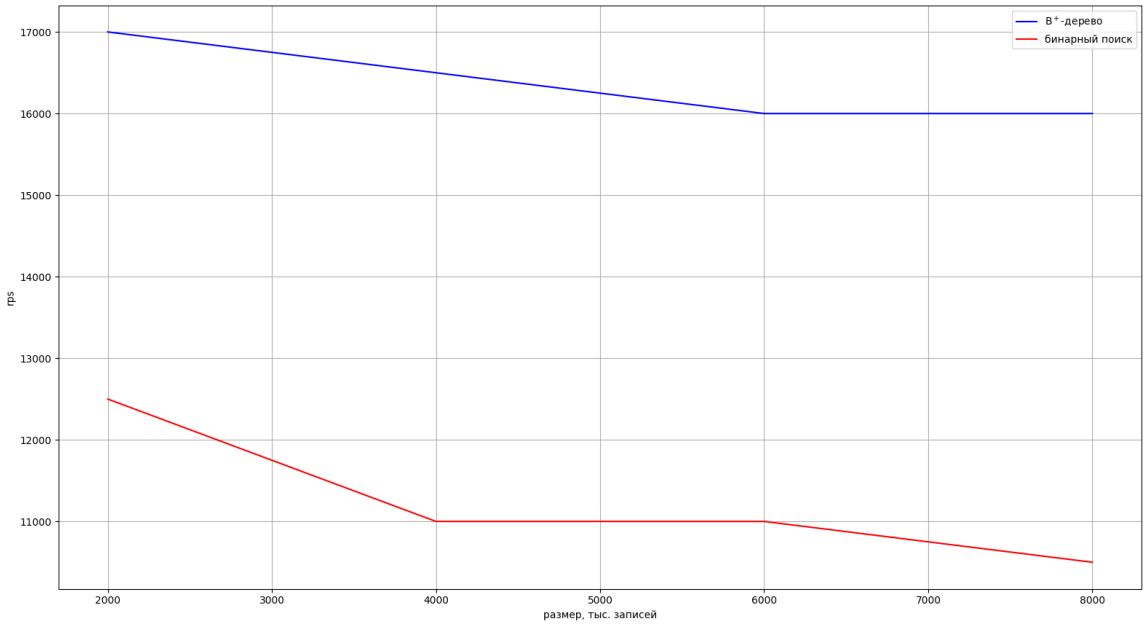

Рис. 4.5: График зависимости среднего количества ответов кластера в секунду (rps) для последовательных запросов ключей от размера кластера

Вывод

В данном разделе были протестированы старая и новая реализация файловых индексов изолированно от базы данных, а также в составе кластера. Данный эксперимент показал, что в изолированном случае индексы могут быть найдены в 9 раз быстрее, в то время как в составе кластера скорость работы увеличилась не так значительно: в зависимости от видов запросов и степени наполнения кластера скорость работы индексов на основе B^+-дерева от 5 до 60% быстрее.

Заключение

В ходе курсовой работы было проведено исследование и предложен метод оптимизации операции поиска в базе данных Pearl.

Для этого были выполнены следующие задачи:

- рассмотрен существующий метод поиска в определенной базе данных;
- рассмотрены способы оптимизации поиска и выбран один из них;
- реализован программно предложенный способ оптимизации;
- проведено сравнение результатов.

В ходе сравнения результатов было выяснено, что выбранное в качестве оптимизации B^+-дерево действительно дает преимущество по скорости работы от 5% до 900% в зависимости от способов использования.

Литература

[1] Аналитика неструктурированных данных. Режим доступа https://www.osp.ru/os/2012/06/13017038 (дата обращения 06.05.2021).

[2] To BLOB or Not To BLOB: Large Object Storage in a Database or a Filesystem? Режим доступа https://arxiv.org/pdf/cs/0701168.pdf (дата обращения 04.05.2021).

[3] Microsoft. Режим доступа https://www.microsoft.com/ (дата обращения 05.05.2021).

[4] BLOB data in NoSQL. Режим доступа https://www.sswug.org/bentaylor/editorials/blob-data-in-nosql-2/ (дата обращения 07.05.2021).

[5] Мартин Фаулер Прамодкумар Дж. Садаладж. NoSQL: новая методология разработки нереляционных баз данных = NoSQL Distilled // «Вильямс». 2013. с. 192.

[6] Шнайер Брюс. Прикладная криптография. Протоколы, алгоритмы, исходные тексты на языке Си // Триумф. 2002.

[7] PostgreSQL. Режим доступа https://www.postgresql.org/ (дата обращения 03.05.2021).

[8] lo – module for large objects. Режим доступа https://www.postgresql.org/docs/9.1/lo.html (дата обращения 03.05.2021).

[9] Scaling horizontally vs. scaling vertically. Режим доступа https://www.section.io/blog/scaling-horizontally-vs-vertically/ (дата обращения 04.05.2021).

[10] Точка отказа. Режим доступа https://dic.academic.ru/dic.nsf/eng_rus/327695/ (дата обращения 04.05.2021).

[11] The key-value database defined. Режим доступа https://aws.amazon.com/nosql/key-value/ (дата обращения 05.05.2021).

[12] Redis. Режим доступа https://redis.io/ (дата обращения 25.04.2021).

[13] Tarantool. Режим доступа https://www.tarantool.io/en/ (дата обращения 25.04.2021).

[14] Performance of in-memory databases. Режим доступа https://reader.elsevier.com/reader/sd/pii/S1319157816300453?token=BBFEFA43B9F01ADE11AD6E8A02E8898BCABC8324C4215B3EE5B5F3BB8B6C9D91F4C originRegion=eu-west-1&originCreation=20210516111531 (дата обращения 05.04.2021).

[15] What is ssd? Режим доступа https://www.crucial.ru/articles/about-ssd/what-is-an-SSD (дата обращения 05.04.2021).

[16] RocksDB. Режим доступа https://github.com/facebook/rocksdb (дата обращения 07.04.2021).

[17] Sled. Режим доступа http://sled.rs/ (дата обращения 05.05.2021).

[18] HDD. Режим доступа https://smarthdd.com (дата обращения 05.05.2021).

[19] Pearl. Режим доступа https://github.com/qoollo/pearl (дата обращения 05.05.2021).

[20] Rust Programming Language [Электронный ресурс]. Режим доступа: https://doc.rust-lang.org/std/index.html (дата обращения: 20.07.2020). 2017.

[21] BTreeMap. Режим доступа https://doc.rust-lang.org/std/collections/struct.BTreeMap.html (дата обращения 11.04.2021).

[22] В. Левитин А. Пространственно-временной компромисс: B-деревья // Алгоритмы. Введение в разработку и анализ. 2006. С. 331–339.

[23] B-Tree Indexes implementation in PostgreSQL. Режим доступа: https://www.postgresql.org/docs/11/btree-implementation.html (дата обращения: 08.04.2021).

[24] В. Левитин А. Метод декомпозиции: бинарный поиск // Алгоритмы. Введение в разработку и анализ. 2006. С. 180–183.

[25] Амортизационный анализ. Режим доступа https://neerc.ifmo.ru/wiki/index.php?title=\T2A\CYRA\T2A\cyrm\T2A\cyro\T2A\cyrr\T2A\cyrt\T2A\cyri\T2A\cyrz\T2A\cyra\T2A\cyrc\T2A\cyri\T2A\cyro\T2A\cyrn\T2A\cyrn\T2A\cyrery\T2A\cyrishrt_\T2A\cyra\T2A\cyrn\T2A\cyra\T2A\cyrl\T2A\cyri\T2A\cyrz (дата обращения: 11.03.2021).

[26] Multiway Trees. Режим доступа: http://faculty.cs.niu.edu/~freedman/340/340notes/340multi.htm (дата обращения: 16.04.2021).

[27] B-Trees. Режим доступа: http://www.cse.unsw.edu.au/~cs9024/10s2/lects/BtreeNotes.pdf (дата обращения: 17.04.2021).

[28] B+-Trees. Режим доступа: http://www.smckearney.com/adb/notes/lecture.btree.pdf (дата обращения: 17.04.2021).

[29] The Cargo Book. Режим доступа: https://doc.rust-lang.org/cargo/ (дата обращения: 21.07.2020).

[30] Rust Language Server (RLS). Режим доступа: https://github.com/rust-lang/rls (дата обращения: 21.07.2020).

[31] Rustbook. What is Ownership? Режим доступа: https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html (дата обращения: 20.07.2020).

[32] VIM the editor. Режим доступа: https://www.vim.org/ (дата обращения: 20.07.2020).

[33] VimAwesome. Режим доступа: https://vimawesome.com/ (дата обращения: 20.07.2020).

[34] Manjaro – enjoy the simplicity [Электронный ресурс]. Режим доступа: https://manjaro.org/ (дата обращения: 21.09.2020).

[35] Русская информация об ОС Linux [Электронный ресурс]. Режим доступа: https://www.linux.org.ru/ (дата обращения: 21.09.2020).

[36] Процессор Intel® Core™ i7-8550U [Электронный ресурс]. Режим доступа: https://ark.intel.com/content/www/ru/ru/ark/products/122589/intel-core-i7-8550u-processor-8m-cache-up-to-4-00-ghz.html (дата обращения: 21.09.2020).

[37] Bob. Режим доступа https://github.com/qoollo/bob (дата обращения 08.05.2021).