

Министерство образования и науки Российской Федерации  
Федеральное государственное автономное образовательное учреждение высшего  
образования  
«Московский физико-технический институт  
(национальный исследовательский университет)»  
Факультет инноваций и высоких технологий  
Кафедра алгоритмов и технологий программирования

# ИНДЕКСИРОВАНИЕ МНОГОМЕРНЫХ ДАННЫХ ПРИ ПОМОЩИ КРИВОЙ МОРТОНА В NOSQL IN-MEMORY СУБД

Выпускная квалификационная работа  
(магистерская работа)

Направление подготовки: 03.03.02 Прикладные математика и информатика

Выполнил:

студент М05-896 группы

\_\_\_\_\_

Бабин Олег Борисович

Научный руководитель:

\_\_\_\_\_

Юхин Кирилл Викторович

Москва 2020

## Оглавление

	Стр.
<b>Введение</b> . . . . .	<b>3</b>
<b>Глава 1. Введение</b> . . . . .	<b>4</b>
1.1 Постановка задачи . . . . .	4
1.2 Математическая модель . . . . .	5
1.3 Оценка полученных результатов . . . . .	7
<b>Глава 2. Анализ предметной области</b> . . . . .	<b>8</b>
2.1 Типы индексов . . . . .	8
2.2 Используемые индексы в различных СУБД . . . . .	18
2.3 Выбор платформы для разработки . . . . .	19
<b>Глава 3. Реализация</b> . . . . .	<b>21</b>
3.1 Bit array . . . . .	21
3.2 Z-order curve . . . . .	23
3.3 Tarantool B-Tree . . . . .	25
3.4 Tarantool index . . . . .	26
3.5 Lua frontend . . . . .	31
<b>Глава 4. Результаты</b> . . . . .	<b>33</b>
4.1 Сравнение с R-Tree. Поиск точек внутри гиперкуба . . . . .	34
4.2 Сравнение с B-Tree. Поиск точек внутри гиперкуба . . . . .	40
4.3 Сравнение с B-Tree. Префиксный поиск . . . . .	41
4.4 Вывод . . . . .	42
<b>Список литературы</b> . . . . .	<b>47</b>

## Введение

Объемы данных, собираемых и анализируемых людьми, постоянно растут. Кроме того, постоянно повышаются требования к скорости чтения/записи этих данных. Сами данные обычно хранятся в структурированном виде, обладая рядом характеристик, по которым необходимо осуществлять их выбор. К решению этой проблемы можно подходить несколькими способами: во-первых, масштабировать систему хранения данных — добавление вычислительных мощностей должно снижать время обработки и поиска. Однако, в реальности такое масштабирование не является выгодным — оно стоит денег: сначала покупка оборудования, затем его поддержка.

## Глава 1. Введение

### 1.1 Постановка задачи

Задачи хранения и обработки большого числа данных встречаются повсеместно. При этом хочется делать это эффективно и быстро. Данная работа охватывает случай, когда область — это не просто интервал на плоской прямой, а некоторая область в многомерном ( $N \geq 1$ ) пространстве.

Представим себе торговую площадку, каждый товар обладает рядом характеристик непрерывных или дискретных, как например на рисунках 1.1.1, 1.1.2, 1.1.3.




	<p><b>Ноутбук Lenovo V130 15</b></p> <p>5.0</p> <p>Покупателям нравится <b>долгое время работы, небольшой вес</b></p> <p>Процессор: Core i3 Объем жесткого диска: 500 ГБ Диагональ экрана: 15.6" Видеокарта: Intel HD Graphics 520 Вес: 1.8 кг</p> <p>28 человек купили этот товар</p>	<p>Есть акции</p> <p><b>от 16 270 Р</b></p> <p>251 предложение от 16 270 Р</p>	<p>Показать всё</p> <p><b>Тип</b></p> <p><input type="checkbox"/> ноутбук <input type="checkbox"/> ноутбук-планшет <input type="checkbox"/> трансформер</p> <p><b>Размер экрана</b></p> <p><input type="checkbox"/> 11"-11.9" <input type="checkbox"/> 12"-12.9" <input type="checkbox"/> 13"-13.9" <input type="checkbox"/> 14"-14.9" <input type="checkbox"/> 15"-15.9" <input type="checkbox"/> 17" и более <input type="checkbox"/> до 11"</p> <p><b>Разрешение экрана</b></p> <p><input type="checkbox"/> 1920x1080 <input type="checkbox"/> 1366x768 <input type="checkbox"/> 1600x900 <input type="checkbox"/> 3840x2160 <input type="checkbox"/> 2560x1600</p> <p>Показать всё</p> <p><b>Процессор</b></p> <p><input type="checkbox"/> Core i5 <input type="checkbox"/> Core i7 <input type="checkbox"/> Core i3 <input type="checkbox"/> Pentium <input type="checkbox"/> Celeron</p> <p>Показать всё</p>
<p>Выбор покупателей</p> 	<p><b>Ноутбук Xiaomi Mi Notebook Air 13.3" 2018</b></p> <p>5.0 7 отзывов</p> <p>Покупателям нравится <b>экран, мощный процессор</b></p> <p>Объем жесткого диска: 256 ГБ Диагональ экрана: 13.3" Видеокарта: NVIDIA GeForce MX150 Вес: 1.3 кг Оптический привод: DVD нет</p> <p>410 человек купили этот товар</p>	<p><b>от 54 400 Р</b></p> <p>30 предложений от 54 400 Р</p>	
<p>Выбор покупателей</p> 	<p><b>Ноутбук Xiaomi Mi Gaming Laptop</b></p> <p>5.0 3 отзыва</p> <p>Покупателям нравится <b>мощный процессор, экран</b></p> <p>Диагональ экрана: 15.6" Видеокарта: NVIDIA GeForce GTX 1060 Вес: 2.7 кг Оптический привод: DVD нет 4G LTE: нет</p> <p>398 человек купили этот товар</p>	<p><b>от 69 000 Р</b></p> <p>18 предложений от 69 000 Р</p>	

Рисунок 1.1.1 — Продажа электроники

Правильно организованная работа с данными увеличивает скорость выполнения запросов и позволяет экономить на оборудовании. Для этого стоит выбрать правильную структуру данных для хранения. Далее будут рассмотрены типы индексов, используемые в современных СУБД и проведено их сравнение.

## Легковые автомобили

Все Новые С пробегом Сохранить поиск

Марка Модель Поколение +

Кузов Коробка Двигатель Привод Объем от, л до

Год от до Пробег от, км до Цена от, ₽ до

Все параметры Показать 52 532 предложения

Audi	2675	Ford	2956	LADA (BA3)	3780	Opel	1989
BMW	4061	Hyundai	2276	Mercedes-Benz	4293	Toyota	2056
Chevrolet	1910	Kia	2339	Nissan	2693	Volkswagen	3432

до 50 000 ₽ до 100 000 ₽ до 150 000 ₽ до 200 000 ₽ до 250 000 ₽ до 300 000 ₽ до 350 000 ₽ до 400 000 ₽ < >

Рисунок 1.1.2 — Продажа автомобилей

## 1.2 Математическая модель

Сформулируем математическую модель, а также характеристики, на которых будет сделан акцент в данной работе.

В базовом случае интересующий нас объект — это хранилище данных 1.2.1, обрабатывающее пользовательские запросы. Запросы могут быть нескольких типов: создание, чтение, обновление и удаление записей. В зависимости от типа запроса запрашивающая сторона (ей может быть пользователь или какая-то другая система) должна получить некоторый результат: данные, удовлетворяющие условию запроса, если это чтение, и «подтверждение» факта, что данный запрос успешно выполнен. Важна также корректность результатов наших запросов — если наш запрос содержит некоторые условия, то при его работе не должны возвращаться, изменяться и удаляться записи, не удовлетворяющие данному условию. Также мы не должны иметь побочных эффектов в виде появления записей, не создаваемых напрямую с помощью запросов или не предусмотренных внутренней задокументированной логикой работы данной базы данных.

Квартира

Комнаты

Цена

+ Еще фильтры

Найти

Город

Метро

Район

Шоссе

Все

Новостройка

Вторичка

Сохранить мой поиск

Опубликовано

За месяц

Цена

За всю площадь

Общая площадь (м²)

От

До

Площадь кухни (м²)

От

До

Жилая площадь (м²)

От

До

Материал здания

☐ Кирпич
☐ Блоки
☐ Монолит
☐ Панель
☐ Дерево

Год постройки

С

По

Дома под снос

Неважно

Этаж

От

До

Не последний

Этажей в доме

От

До

До метро (по карте)

-

Продавец

Любой

Агент/агентство

Собственник

Показать только

С фотографиями

Добавить ключевое слово

Например: окна во двор, консьерж, название ЖК или адрес

Исключить ключевое слово

Например: окна на улицу, требует ремонта или балкона нет

Рисунок 1.1.3 — Продажа недвижимости



Рисунок 1.2.1 — Схематическое представление системы хранения данных

Конечно, обычно СУБД предлагают более широкий список возможностей: поддержку групп и ролей, разграничение доступа в соответствии с ними, создание и использование пользовательских функций, но это не будет рассмотрено, поскольку не имеет непосредственного отношения к рассматриваемой задаче.

Не имеет особой разницы форма запроса и ответа на него — предполагаем, что существует некоторый протокол, который и используется при общении

с данной СУБД при этом на коммуникацию между системами тратится сравнимое и меньшее время, чем на выполнение запроса.

Получаемые результаты могут достаточно сильно зависеть от платформы, на которой будут запускаться тесты. Более подробное описание среды тестирования, платформы и самих тестов будет приведено в следующих главах.

### 1.3 Оценка полученных результатов

Для оценки полученных результатов будут проведены несколько тестов. Для любой структуры данных ключевыми характеристиками являются скорость чтения и скорость записи.

Для оценки скорости записи будет сгенерировано множество объектов, которые затем должны быть вставлены в нужную структуру данных. Если существует несколько структур, которые предполагается сравнивать, но формат объекта, предназначенного для вставки, у них разный, то должно быть сгенерировано несколько множеств объектов, эквивалентных друг другу, но имеющих формат, совместимый с нужной структурой хранения. В дальнейшем данные точки последовательно вставляются в исследуемую структуру. Время этой вставки измеряется. Каждый эксперимент следует повторять несколько раз для получения усредненных результатов и исключения возможных «выбросов».

Аналогично исследуются запросы на чтение. В этом случае предварительно должны быть подготовлены запросы за данными.

Лучшими характеристиками обладает структура, способная обработать как можно большее число запросов, как на чтение, так и на запись.

Кроме того, есть и третья метрика — память, затрачиваемая на хранение вставленных в неё данных. При этом измеряется количество памяти, потребляемое непосредственно структурой, без учёта самих данных — издержки на хранение в данной структуре.

## Глава 2. Анализ предметной области

### 2.1 Типы индексов

#### B-Tree

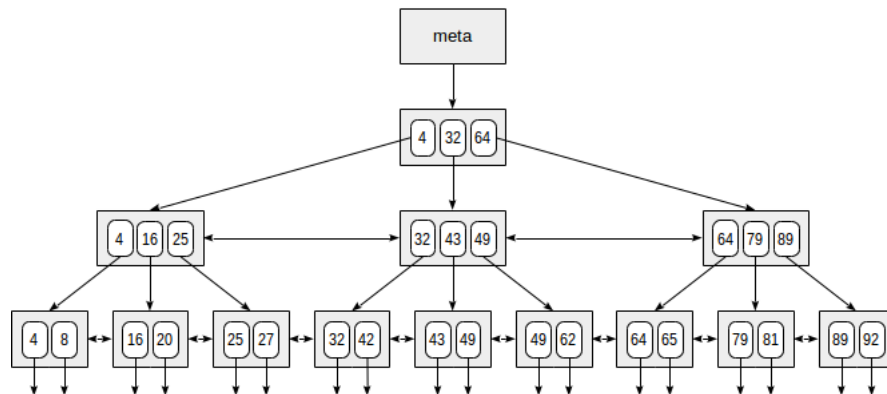


Рисунок 2.1.1 — Схематичный пример индекса по одному полю с целочисленными ключами

Для хранения данных в отсортированном виде обычно используется B-Tree [1]. Чтобы примерно представить себе работу следует вспомнить обычное бинарное дерево (поиск по нему имеет логарифмическую сложность). Однако в данном случае всё устроено сложнее: дерево сбалансировано и сильно ветвистое — каждый узел обычно имеет более двух потомков **2.1.1.**

Элементы данного дерева отсортированы по возрастанию. Что позволяет эффективно выполнять поиск как отдельного значения, так и интервала значений. Ситуация ухудшается, когда необходимо делать поиск по нескольким измерениям. В этом случае мы можем ограничить лишь одно измерение, поиск по другим будет производиться полным перебором.

Тем не менее для большинства задач B-дерево всё-таки является хорошим вариантом. B-Tree можно назвать самым популярным индексом, используемым в большинстве современных СУБД как реляционных, так и нереляционных при этом абсолютно не важно, где именно хранятся данные — в памяти или на диске. Существует много модификаций B-Tree: B+Tree (используется в CouchDB, MongoDB), SB-Tree (OrientDB), B\*-Tree.



В-дерево было предложено ещё в 1970 году для эффективного поиска среди файлов [2], с тех пор появилось большое количество эффективных и компактных реализаций этой структуры. Такую популярность данная структура получила благодаря своей работе с памятью. Если мы хотим прочитать какое-либо значение, то в память/кэш помещается весь блок данных. Что существенно ускоряет скорость чтения, если кроме этого потребуются и соседние значения. Однако это является и недостатком этой структуры — если требуется записать новое или перезаписать существующее значение, будет обновлен весь блок. Такие «паразитные» чтения в литературе, посвящённой хранению на диске, называется *read amplification*, а «паразитные» записи — *write amplification*. Формально, *amplification factor*, то есть коэффициент умножения, вычисляется как отношение размера фактически прочитанных (или записанных) данных к реально необходимому (или изменённому) размеру. В случае В-дерева порядок этого коэффициента — десятки и сотни.

## Hash

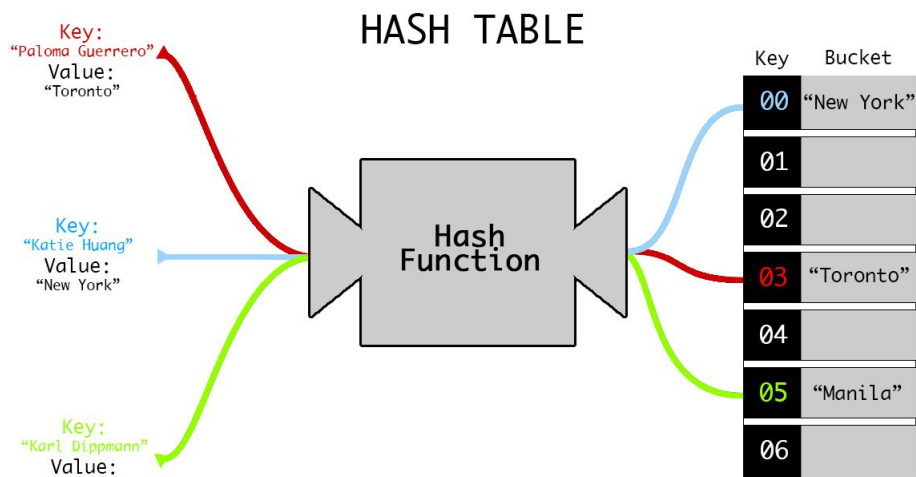


Рисунок 2.1.2 — Схематический пример организации работы hash-индекса

Hash-индекс работает не с индексируемыми ключами, а с их хэшами. Идея хэширования состоит в том, чтобы значению любого типа данных сопоставить некоторую битовую последовательность фиксированной длины. Функцию, осуществляющую такое преобразование, называют хэш-функцией. Вычисленное

значение указывает на некоторую область, хранящую нужную запись. Доступ к этой записи может быть получен за константное время —  $O(1)$  (рис. 2.1.2).

Ключевым отличием от, например, B-Tree является отсутствие возможности поиска в интервале (близкие значения обычно имеют различные хэши). Кроме того отсутствует возможность выборки по префиксу, поскольку хэш вычисляется от полного ключа.

Существует класс задач, для которых важна скорость доступа к данным по простому ключу. Это привело к появлению так называемых key-value баз данных, например, Redis, Riak, memcached.

## LSM-Tree

Ещё одним типом дерева, наравне с B-Tree, предназначенным для хранения данных является LSM-Tree — *Log-structured merge-tree* [3]. В отличие от B-Tree, которое можно использовать как для хранения в памяти, так и на диске, LSM-Tree предназначено для хранения данных именно на диске. Разделение на части, хранящиеся в памяти и на диске заложено в саму архитектуру данной структуры. Все операции вставки делаются в L0 (уровень, хранящийся в оперативной памяти), как только место там заканчивается, данные начинают сбрасываться на диск (рис. 2.1.3).

Ещё одним ключевым отличием является то, что в узлах дерева хранятся не сами данные, а операции с ними (рис. 2.1.4).

LSM-деревья работают быстрее для частых вставок и редких чтений, в отличие от B-деревьев, иными словами write amplification LSM-деревьев меньше, но read amplification выше. LSM-деревья стали довольно популярны в последнее время, это связано с тем, что стоимость внешней памяти стала меньше, при этом популярность приобретают SSD-диски, обладающие более высокой скоростью чтения по сравнению с устаревающими HDD-дисками.

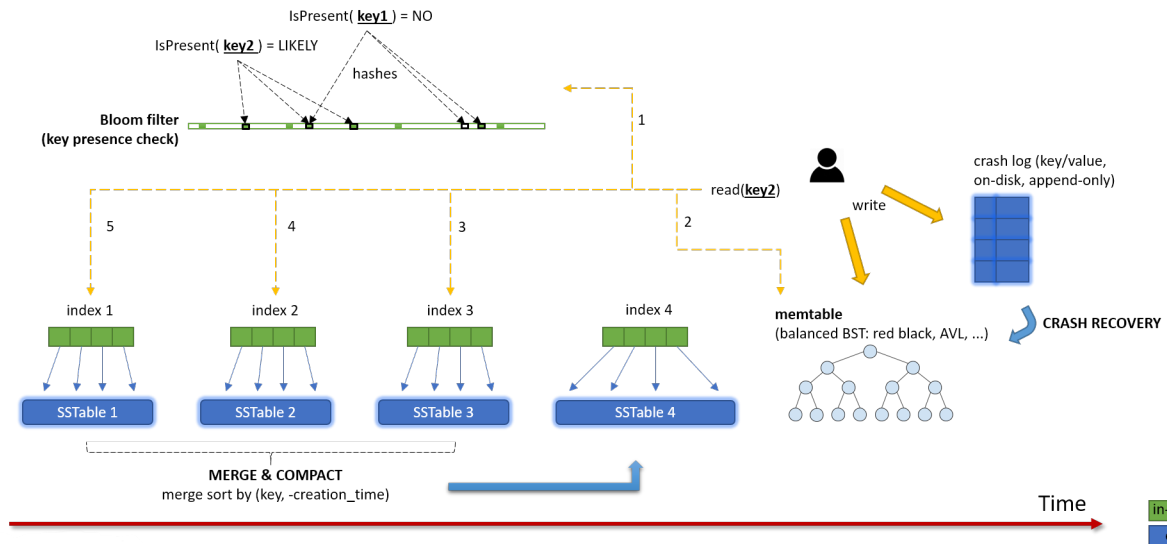


Рисунок 2.1.3 — Схематическое представление LSM-Tree

key	lsn	Op code	Value
1	176	REPLACE	2018-05-07 15:00:01
1	53	INSERT	2017-12-31 23:59:01
2	174	REPLACE	2018-05-06 00:00:00
3	175	REPLACE	2018-05-07 09:04:19
3	9	REPLACE	2017-01-01 19:25:43
3	7	INSERT	2017-01-01 19:22:16
4	173	DELETE	
4	168	INSERT	2018-05-05 07:40:01

Рисунок 2.1.4 — Хранение операций над данными, а не самих данных

## Inverted index

Индекс, использующаяся для полнотекстового поиска. Содержит список всех уникальных слов и ссылки на документы, в которых эти слова встретились (см. рис 2.1.5).

Полнотекстовые запросы выполняют лингвистический поиск в текстовых данных путем обработки слов и фраз в соответствии с правилами конкретного языка: разбиение на слова, отсекаание окончаний, выбор однокоренных слов и т.д. Отдельными задачами при полнотекстовом поиске являются ранжирование результатов запроса и исключение ненужных слов.

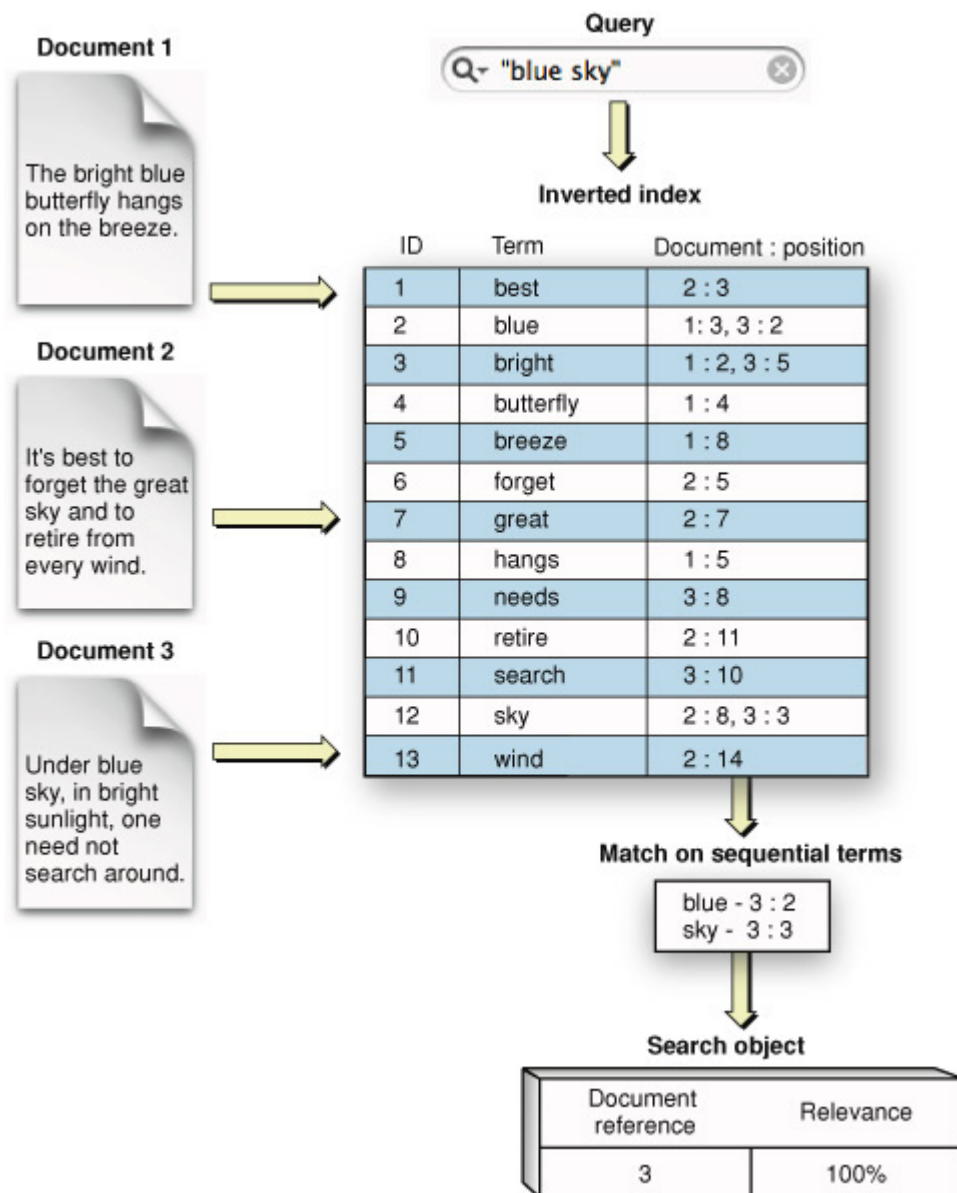


Рисунок 2.1.5 — Хранение документов в обратном индексе

Реализации полнотекстового поиска варьируются в различных СУБД. Инвертированный индекс используется в Microsoft SQL Server, MySQL, OrientDB и поисковом движке Elasticsearch.

Обобщением данного типа индекса является *GIN* (Generalized Inverted Index), реализованный в PostgreSQL. Кроме полнотекстового поиска, является подходящим для индексирования массивов и JSON. Обобщенным он называется, потому что операция над индексируемым объектом задается отдельно в отличие от, например, B-Tree, где все операции сравнения уже заданы. В качестве операции могут использоваться такие как «содержит», «пересекается», «содержится».

Количество текстовой информации, окружающей нас огромно: новости, книги, письма и т.д. Для индексирования содержимого этот тип индекса является подходящим. Однако работа с текстом не входит в поставленную задачу.

## Пространственные индексы

Большинство современных СУБД имеют типы, предназначенные для работы с пространственными типами данных: точки, прямые, окружности и другие геометрические объекты. Для данных объектов используются свои стратегии индексирования.

Известными решениями является использование пространственной сетки (spatial grid), дерева квадрантов (quadtree) и R-Tree.

Данные индексы используются графовыми базами данных (Neo4j, AllegroGrath), однако существуют специальные дополнения и расширения для известных СУБД, но предназначенные для обработки исключительно пространственной информации, например, PostGIS [4], Oracle Spatial [5].

## R-Tree

R-дерево было предложено в 1984 году Антонином Гуттманом [6], и предназначается для поиска среди многомерных данных.

Перед описанием работы данной структуры следует ввести понятие ограничивающего прямоугольника англ. *bounding box*. Эти прямоугольники являются N-мерными и содержат в себе либо данные, либо другие ограничивающие прямоугольники. С точки зрения хранения каждый прямоугольник хранится как набор координат той же размерности, что и объект им ограничиваемый. Структура R-tree не накладывает никаких ограничений на размер данного объекта, однако по соображениям производительности полезно выбирать прямоугольник минимального размера англ. *minimum bounding box*. Так, например, минимальный ограничивающий прямоугольник для окружности на

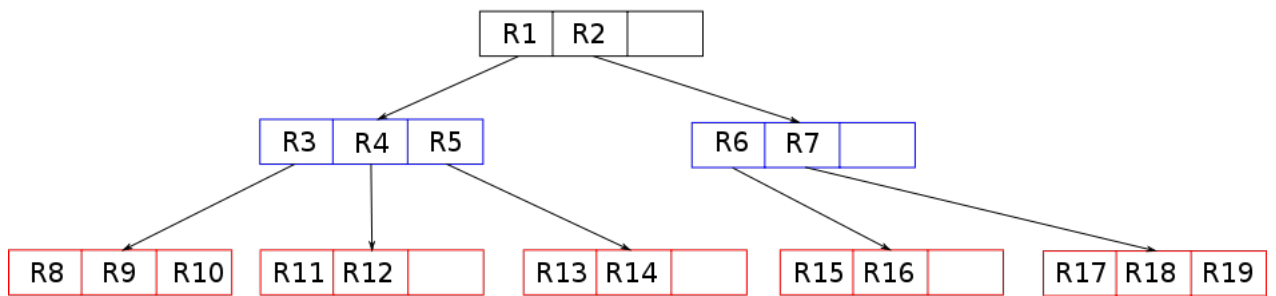
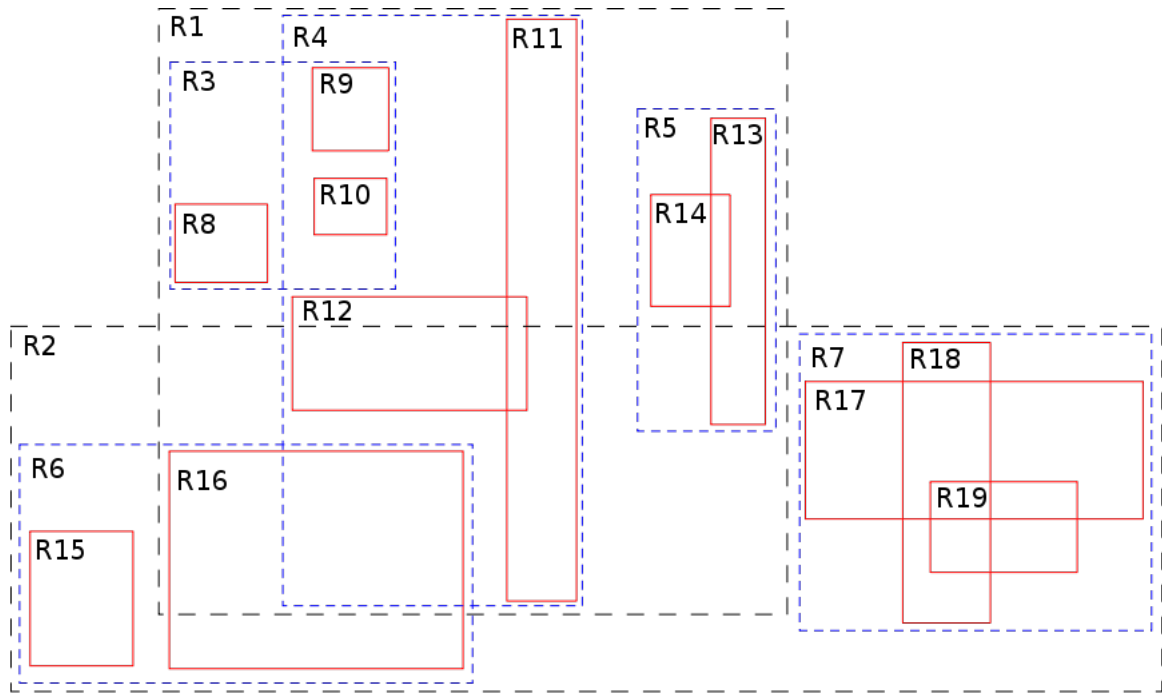


Рисунок 2.1.6 — Пример организации хранения данных в R-Tree

плоскости имеет сторону равную диаметру этой окружности. Аналогично для трехмерной сферы, это куб со стороной, равной диаметру сферы.

R-дерево — иерархическая структура, хранящая информацию об ограничивающих прямоугольниках. Каждая вершина имеет переменное количество элементов (не более некоторого заранее заданного максимума). Каждый элемент нелистовой вершины хранит два поля данных: способ идентификации дочерней вершины и ограничивающий прямоугольник, охватывающий все элементы этой дочерней вершины. Все хранимые кортежи хранятся на одном уровне глубины, таким образом, дерево идеально сбалансировано. Алгоритмы вставки и удаления используют эти ограничивающие прямоугольники для обеспечения того, чтобы «близкорасположенные» объекты были помещены в одну листовую вершину. В частности, новый объект попадёт в ту листовую вершину,

для которой потребуется наименьшее расширение её ограничивающего прямоугольника. Каждый элемент листовой вершины хранит два поля данных: способ идентификации данных, описывающих объект, и ограничивающий прямоугольник этого объекта. Аналогично, алгоритмы поиска (например, пересечение, включение, окрестности) используют ограничивающие прямоугольники для принятия решения о необходимости поиска в дочерней вершине.

Подобно В-дереву, при попадании в узел мы проверяем, в какой узел следует двигаться дальше, чтобы найти данные, соответствующие запросу. При этом из-за того, что прямоугольники могут пересекаться в одну придется проверить несколько путей перед тем, как будет найден нужный узел. В худшем случае придется обойти всё дерево, в лучшем объект будет вложен в корневой узел.

Один из типичных примеров применения R-деревьев — индексирование географических данных. При этом никаких ограничений на физические характеристики данная структура не ставит.

Данный тип индекса поддерживается некоторыми движками СУБД MariaDB (SPATIAL INDEX), PostgreSQL (RTREE), Oracle и др.

Существуют модификации R-дерева, например, R\*-Tree [7] и R+Tree. Обобщением R-дерева является *GiST (The Generalized Search Tree)* — обобщенное дерево поиска. Реализовано в PostgreSQL и подобно GIN поддерживает индексирование произвольной информации (геоданные, тексты, изображения и т.д.) с использованием операций «принадлежит», «содержит», «совпадает», «соответствует».

### **Z-order curve (Кривая Мортон)**

Данная кривая была предложена Гаем Макдональдом Мортон в 1966 году в работе [8], посвященной хранению и обработке данных, получаемых в рамках проекта «Canada Land Inventory». Однако на тот момент не ставилось цели выполнять сложные поисковые запросы. Необходимо было лишь находить нужный файл, представляемый парой координат.

В последствии данный метод нашел своё применение и при работе с базами данных. Индекс, на основе данной кривой, используется для хранения многомерных данных в одномерной структуре. К каждому значению

$(x_1, \dots, x_n)$  применяется кодирование Мортонa, заключающееся в чередовании двоичных цифр координатных значений, полученный результат называется кривой Z-порядка (англ. *Z-order curve*). Для хранения полученных значений можно использовать любую известную структуру данных: бинарное дерево, хэш-таблицу, B-дерево. В общем случае выбор структуры поиска зависит от задачи.

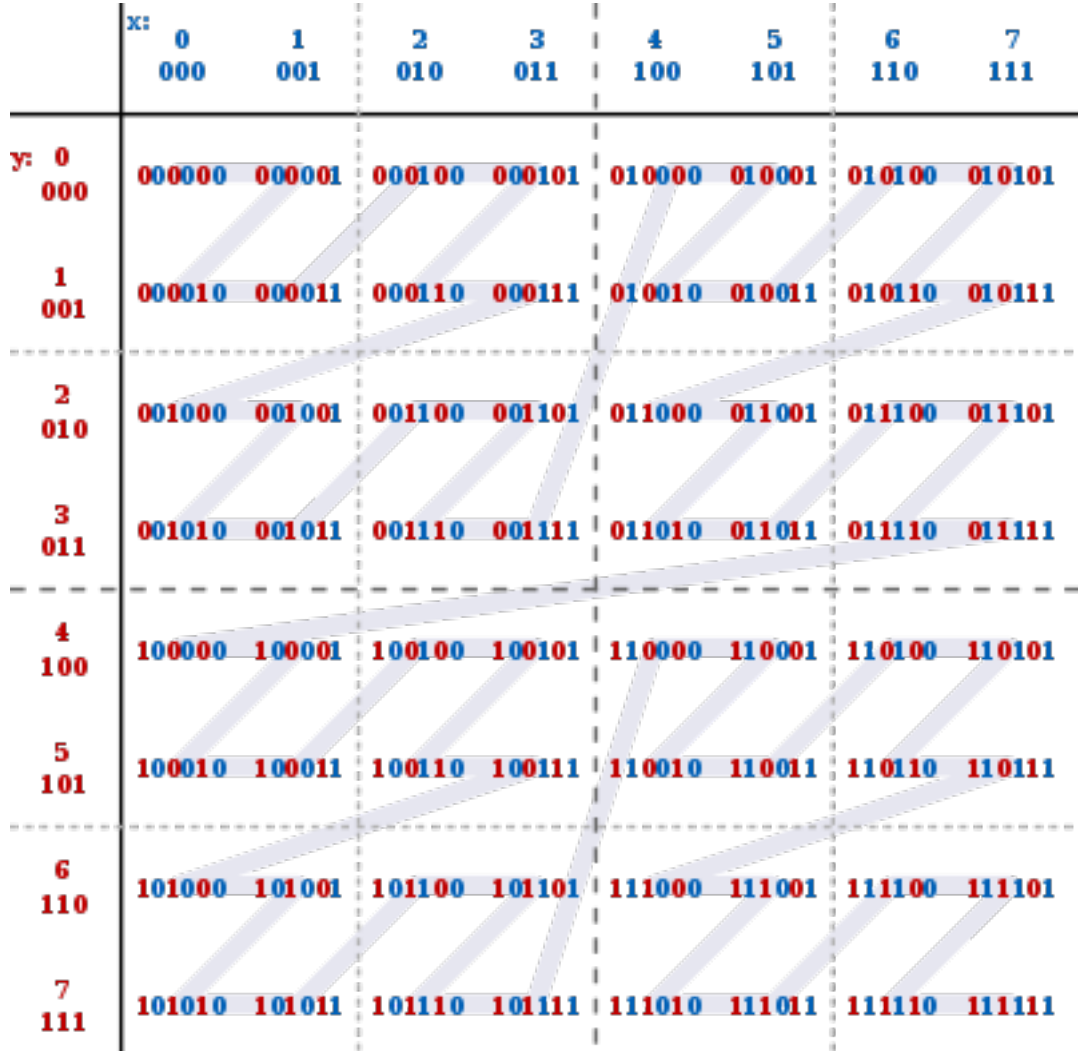


Рисунок 2.1.7 — Построение Z-последовательности

В мире баз данных де-факто стандартом для хранения одномерных данных является B-дерево. Совместное использование кривой Мортонa и B-дерева позволяет эффективно производить поиск по интервалам значений, однако часть возвращаемого результата может и не находиться в указанном интервале (рис. 2.1.8), поэтому при запросе приходится применять дополнительные механизмы для фильтрации данных. Эти аспекты будут изложены в секции 3.2.

В литературе использование Z-order curve совместно с B-Tree часто называют «UB-Tree» (Universal B-Tree) [9–11].



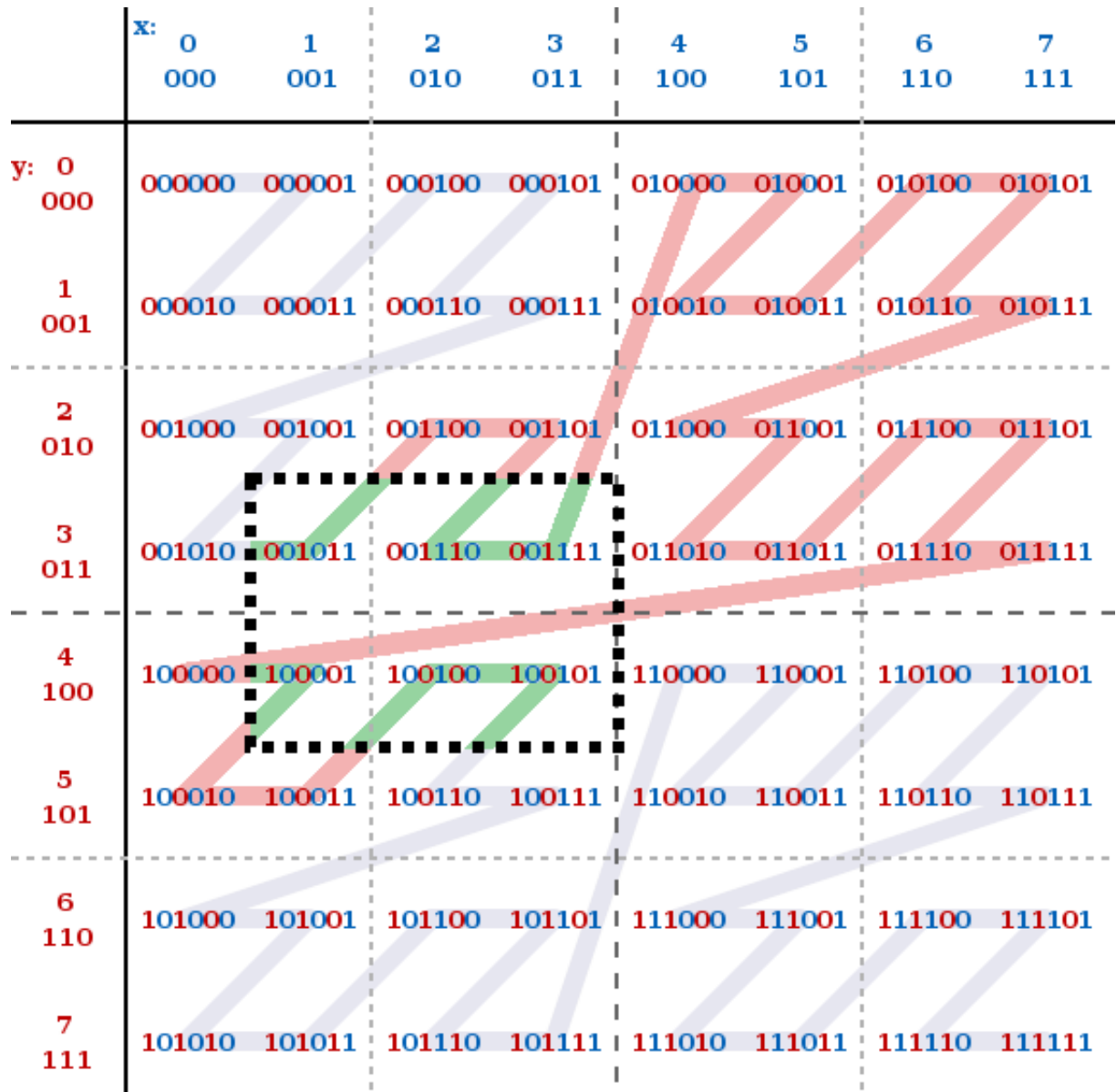


Рисунок 2.1.8 — Поиск значений в интервале

При реализации Z-адрес рассматривается исключительно как битовая последовательность. Это значит, что единственное ограничение на тип ключа — возможность упорядочивания при переходе к двоичному представлению. В итоге доступные типы ограничиваются не только целыми числами, но и числами с плавающей точкой, строками, временными метками...

Данный тип индексирования используется в TransBase[9], Accumulo, HBase [12], DynamoDB[13; 14].

Стоит отметить, что данное преобразование не является единственным для отображения многомерных данных в одномерные. Семейство таких кривых называется «кривые, заполняющие пространство» [15]. Могут использоваться

также другие кривые [10], например, Гильберта [16] или Пеано. Однако Z-последовательность гораздо проще для вычисления, а также сохраняет локальность точек — точки, находившиеся близко в пространстве будут часто близкими и на плоской кривой.

## Индексы с использованием машинного обучения

Можно выделить несколько подходов, которые могут быть использованы для поиска информации и выделения закономерностей в больших массивах данных — Latent Semantic Indexing (LSI) и Hidden Markov Model (HMM). Данные варианты хоть и являются интересными и полезными в некоторых сферах, но примеров их использования в каких-либо СУБД нет.

### 2.2 Используемые индексы в различных СУБД

СУБД	Индексы
PostgreSQL	B-Tree, R-Tree, Hash, GiST, SP-GiST, GIN, RUM, BRIN, Bloom
MySQL/MariaDB	B-Tree, Hash, R-Tree, Inverted Index
Oracle	B-Tree, B-Tree-cluster, Hash-cluster, Reverse key, Bitmap
MongoDB	B-Tree, Geohash, Text index, Hash
OrientDB	SB-Tree, Hash, Lucene Fulltext, Lucene Spatial
MemSQL	SkipList, Hash, Columnstore

## 2.3 Выбор платформы для разработки

В качестве платформы, для которой будет реализован индекс на основе кривой Мортонa была выбрана СУБД Tarantool — СУБД с открытым исходным кодом и активно разрабатываемая в настоящее время.

Предлагаемое решение достаточно просто может быть встроено в существующую кодовую базу — около 200 тысяч строк на языке C и 10 тысяч строк на языке Lua (без учета тестов и зависимостей) [17].

Основные особенности СУБД Tarantool:

- Удовлетворение принципу ACID
- Наличие двух движков для хранения данных на диске (vinyl) и в оперативной памяти (memtx)
- Хранение данных в формате MessagePack
- Обработка транзакций в одном потоке
- Не только база данных, но и сервер приложений

Реализовываемый индекс будет работать с движком Memtx — все данные будут храниться в оперативной памяти. Часть уже существующих решений может быть переиспользована, например, B+\*-Tree (BPS, в терминологии Tarantool).

Система типов сочетает в себе особенности языка Lua 5.1 (LuaJIT 2.1 beta 3) [18; 19], которые предоставляются пользователю для взаимодействия с базой данных и сервером приложений, а также формата MessagePack [20], в котором данные хранятся и обрабатываются.

Типы, доступные из Lua-интерфейса: "string"(строковый тип), "number"(double-precision floating-point), "boolean"(логический тип), "table"(ассоциативный массив), "nil"(литерал, обозначающий отсутствие значения), а так же пользовательские типы данных, представленные такими Lua/LuaJIT типами как "userdata"и "cdata".

Если по тем или иным причинам пользователь не хочет работать с Lua-интерфейсом, он может взаимодействовать с СУБД по бинарному протоколу из любого другого языка программирования. Обмен данными в этом случае будет производиться именно в формате MessagePack.

Кроме того присутствует система типов Tarantool — типы, указываемые пользователем при создании индексов или задании формата кортежа

(возможности указывать тип элемента, ожидаемого для вставки). Части из этих типов можно взаимоднозначно сопоставить MessagePack тип, например, "string" boolean другие могут являться подмножеством типов, например, "number"—тип, объединяющий "unsigned integer" "signed integer" и "double".

При реализации необходимо будет решить несколько инженерных задач:

- Разработка пользовательского интерфейса для работы с индексом;
- Встраивание решения в существующую кодовую базу;
- Адаптация решения под существующую систему типов.

Элементарной единицей является кортеж (tuple), аналог строки в реляционных базах данных. Однако в отличие от реляционных БД кортеж может иметь произвольную длину и содержать произвольные типы. Строгая типизация требуется только для индексируемых полей.

## Глава 3. Реализация

Реализацию индекса можно декомпозировать на несколько частей [21]:

- Битовый массив (bit array)
- Логика z-order curve
- Встраивание в движок БД
- Lua-frontend

### 3.1 Bit array

Как описано в предыдущих пунктах, индексируемое значение получается в результате перемешивания битов указанных индексируемых полей. Данную структуру будем называть битовый массив. При реализации прототипа было решено найти и использовать готовую реализацию на языке C. Был выбран проект <https://github.com/noporpoise/BitArray>, который удовлетворял функциональным потребностям, однако имел достаточно небольшое покрытие тестами, содержал дефекты, которые пришлось исправлять — <https://github.com/noporpoise/BitArray/pull/15>. Также возникали вопросы к производительности и оптимальности данного решения.

С учетом того, что для хранения и представления большинства типов используется 64 бита, размер массива всегда будет кратен 64 элементам. А именно  $N * 64$ , где  $N$  — количество частей в индексе. Приведенная выше реализация содержала достаточно большое число проверок и занимала больше памяти из-за того, что являлось массивом общего назначения с возможностью динамического изменения размера. Для достижения максимальной эффективности пришлось реализовать свой битовый массив постоянного размера и более оптимально работающий с памятью (для выделения памяти использовалось 2 системных вызова, один — выделение структуры со служебными полями, другой сам массив). В полученной реализации создание массива — одна аллокация. Служебное поле "количество элементов в массиве" также потеряло смысл и было удалено поскольку вычисляется как  $N * 64$ .

Следующий шаг в оптимизации — «векторизация». Большинство современных процессоров поддерживает так называемые векторные инструкции, служащие для обработки массивов данных (SIMD — Single Instruction Multiple Data). По словам разработчиков использование таких инструкций позволяет получать прирост производительности до 30%. О том, что какой-то цикл может быть векторизован можно с помощью директивы `#pragma simd`, однако это не дает гарантий, что цикл будет векторизован, компилятор может и проигнорировать данную директиву, к тому же большинство современных компиляторов могут автоматически находить векторизуемые циклы. Посмотреть за тем, векторизуется цикл или нет, можно с помощью специальных опций компилятора. Циклы для операций AND и OR были векторизованы для компиляторов GCC v7.4.0 и Apple Clang v11.0.0.

Частая вставка данных в индекс предполагает частые выделения памяти, что при наивном подходе может привести к большому количеству системных вызовов «`syscall`», являющихся достаточно медленными операциями. Гораздо оптимальнее использовать специализированные аллокатеры — сущности, занимающиеся выделением/освобождением памяти с минимальным количеством системных вызовов. Специфика задачи — выделение большого количества одинаковых по размеру участков памяти. Под эту задачу оптимизирован уже имеющийся в Tarantool аллокатер — `mempool` «memory pool».

## Bit interleaving

Поскольку одной из ключевых операций является перемешивание битов, эта функциональность была добавлена для битового массива.

Вычисление производится не тривиальным образом, а с помощью так называемых `lookup`-таблиц. Таблица ставит в соответствие любому числу от 0 до 255 значение, вычисляемое по правилу  $\sum i = 0^7(k_i * 2^{i*n})$ , где  $n$  - число массивов, которые должны быть перемешаны,  $k_i$  - значение  $i$ -ого бита. Для каждой части в зависимости от её номера  $m$  хранится сдвинутая на  $m$  разрядов копия такой `lookup`-таблицы.

Соответственно вычисление  $z$ -адреса происходит за  $8 * n$  обращений, сдвигов и применения логической операции `OR`, где  $n$  - число массивов. Стоит

обратить внимание, что lookup-таблицы вычислены для октетов, а не больших размеров. Использование для  $n = 16/32/64$  значений затруднено, поскольку расходы памяти на хранение этой таблицы растут экспоненциально как  $2^n$ .

### 3.2 Z-order curve

Изначально информация о существовании индекса на основе кривой z-порядка была получена из статей Amazon [13; 14], данные статьи являются скорее инструкцией для пользователей, как создать индекс на основе кривой Мортон для удовлетворения пользовательских потребностей с минимальными накладными расходами (z-order curve не является встроенной функциональностью DynamoDB).

Основные идеи, которые можно было бы почерпнуть из статьи — алгоритмы работы с кривой z-порядка и работа с различными типами данных — необходимо найти функцию, которая для каждого типа данных возвращает битовую строку. Для различных значений битовые строки должны быть лексикографически упорядочены. Например, рассмотрим знаковые целые числа  $-1$  и  $2$ . Для них выполняется следующее неравенство  $-1 < 2$ . Однако в бинарном представлении  $-1$  — это  $11111111$ , а  $2$  —  $00000010$ . Лексикографический порядок нарушен. Для восстановления предлагается инвертировать старший бит, тогда значения для  $-1$  —  $01111111$  и  $10000010$  для  $2$ .

Также стоит отметить, что целесообразным использование данной структуры является при запросах, обладающих следующими свойствами:

- **Ограничение каждой размерности.** Распространена практика, когда часть параметров запроса не задается, а остается открытой, однако в данном случае это может серьезно влиять на производительность.
- **Высокая селективность.** Границы, устанавливаемые при запросе должны исключать большие объемы данных. Для неравномерно распределенных логических значений пространство поиска может быть сильно увеличено.

После вычисления z-адреса для каждого проиндексированного значения и размещения его в B-дереве мы получаем структуру, из которой хотелось бы выбирать значения, соответствующие нашим запросам. Самое первое,

что может определить пользователь - нижнюю и верхнюю границы поиска (*lower* и *upper bound*). Рассмотрим пример для случая двух измерений. Мы хотим сделать выборку в прямоугольнике  $[x_{min}; x_{max}]$  и  $[y_{min}; y_{max}]$  — границами будут  $z\_address(x_{min}, y_{min})$  и  $z\_address(x_{max}, y_{max})$ . В интервал между двумя этими значениями может попадать достаточно большое число значений, не принадлежащих заданному прямоугольнику. Перед тем как объяснять алгоритм итерации по данной структуре следует ввести 2 ключевые функции —  $is\_relevant(lower\_bound, upper\_bound, z\_address)$ , которая проверяет  $z\_address$  на принадлежность заданному прямоугольнику, и  $next\_jump\_in(lower\_bound, upper\_bound, z\_address)$ , возвращающая для  $z$ -адреса за пределами прямоугольника первое по порядку значение, попадающее в прямоугольник. С учетом вышесказанного итерация выглядит следующим образом. Мы начинаем с некоторого  $z\_address$  значения (большего или равного  $lower\_bound$ ), проверяем его принадлежность прямоугольнику с помощью  $is\_relevant$ , если функция вернула *true*, то это значение возвращается пользователю, иначе с помощью  $next\_jump\_in$  переходим к следующему подходящему значению, пока не выйдем за границу  $upper\_bound$ .

Описание двух, используемых выше алгоритмов, можно найти в [9; 11; 22]. С некоторыми изменениями и модификациями они были реализованы в рамках дипломной работы. Например, функция  $is\_relevant$  может рассматриваться как часть функции  $next\_jump\_in$  и практически без изменений извлечена в отдельную функцию. Однако такой наивный подход неоптимален, функцию можно упростить, добавить дополнительные оптимизации и применить эвристики, способные ускорить данные функции. Во-первых, там, где можно было отказаться от массивов в пользу битовых масок, это было сделано. Как было сказано, ключ — это массив 64-битных чисел, однако если индексируемые числа небольшие, то большая часть битов будет нулевой. Приведенные алгоритмы проверки и поиска работают за  $O(N)$ , где  $N$  — длина ключа. При этом если биты с одинаковым порядковым номером  $zvalue$ ,  $lower\_bound$  и  $upper\_bound$  могут быть безболезненно пропущены, это же справедливо и для более крупных единиц, например, байтов. Данная эвристика для размерности 3 и чисел из интервала  $[0; 255]$  дала выигрыш 15 — 20%.

Отдельно стоит отметить более рациональный подход к используемым типам данных. Использование типов как можно меньшей размерности, скажем,



`uint8_t` вместо `uint64_t` дало существенный прирост производительности, сократив время работы функций практически в 2 раза.

### 3.3 Tarantool B-Tree

Как было сказано, Tarantool поддерживает индекс на основе B-Tree. Если быть максимально точным, то B<sup>+</sup>-Tree. Листья таких деревьев имеют блочную структуру, а сами деревья являются сильно ветвящимися. «+» в названии обозначает, что на самом нижнем уровне блоки связаны между собой (рис. 3.3.1), что позволяет не подниматься на уровень выше для перехода из одного блока в другой при итерации. «\*» — улучшение связанное с заполняемостью блоков. При достижении определенного размера блок разделяется на два новых. В случае B\*-дерева блоки заполнены как минимум на 2/3 вместо 1/2 [23].

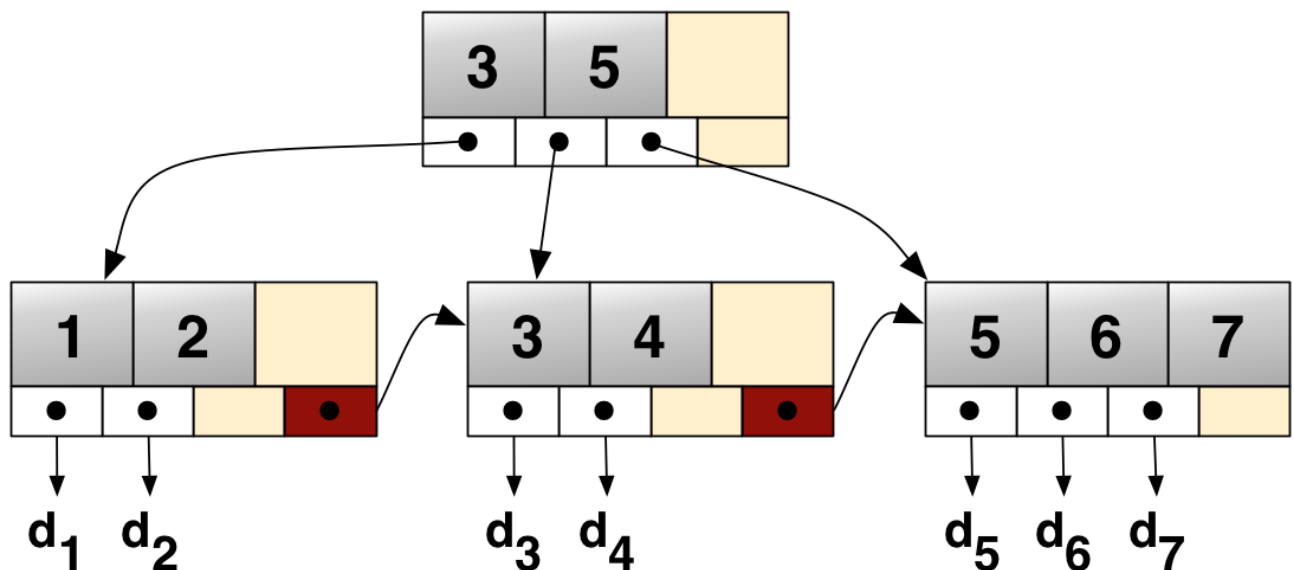


Рисунок 3.3.1 — Связи блоков в B<sup>+</sup>-Tree

Индекс на основе кривой Z-порядка предполагает хранение z-адресов и указателей на элементы спейса внутри B-дерева. В связи с тем, что данная структура уже была реализована, а также много лет уже эксплуатируется в промышленных средах было принято решение не писать новую свою, а использовать уже имеющуюся. Тем более, что данная структура является шаблонной,

позволяет разработчику выбирать структуру какого вида необходимо хранить и как сравнивать ключи.

Расскажем про возможности данной структуры.

В-дерево поддерживает вставку, замену, удаление и поиск элемента по ключу. Вычислительная сложность поиска и модификации —  $\log_B N$ . Поддерживаются также итераторы, предоставляющие последовательный доступ к элементам в порядке возрастания или убывания ключа. Итерация имеет константную сложность.

Среди основных возможностей стоит выделить следующие. Во-первых, дерево является достаточно компактным. Его размер пропорционален размеру структур, хранящихся внутри, и в худшем случае накладные расходы не превышают 60% от содержимого. На практике же данное значение составляет от 20% до 40%.

Во-вторых, данная структура оптимальна при работе с кэшем центрального процессора. Поиск в дереве сводится к поиску в блоках, где  $H$  — высота дерева, пропорциональная отношению  $\log N / \log K$ , где  $N$  — размер дерева, а  $K$  — среднее число элементов в блоке. Таким образом для блоков размером в 512 байт и размере элемента в 8 байт размер дерева, состоящем из миллиона элементов будет иметь высоту 4, а для миллиарда — 6.

Также стоит упомянуть о свойстве итераторов. Вставка нового элемента ломает существующий итератор, однако это не приводит к каким-либо техническим проблемам. Если значение попало в ещё не просканированную область, то оно будет просканировано. Если оно попало в уже просканированную область, то никакого влияния на имеющийся итератор данная вставка не окажет.

### 3.4 Tarantool index

В терминах ООП — индекс это класс, обладающий некоторым набором функций и реализующий интерфейс к некоторой структуре данных, по индексируемым полям вычисляется ключ поиска, а элемент это обычно ключ и значение, являющееся указателем на кортеж.

Перечислим методы, которые должен предоставлять интерфейс нашего индекса: Алгоритм чтения:

- `destroy` — удаление индекса;
- `update_def` — функция вызываемая при обновлении описания индекса, не требующем перестроения этого индекса;
- `depends_on_pk` — является ли описание индекса, зависящим от первичного ключа;
- `index_def_change_requires_rebuild` — возвращает `true/false`, требуется ли перестройка индекса при обновлении определения индекса;
- `size` — количество элементов, которые содержит индекс;
- `bsize` — размер индекса в байтах, не включающий в себя размер кортежей;
- `min/max` — минимальный/максимальный элементы;
- `random` — случайный элемент;
- `count` — количество элементов с определенным ключом;
- `get` — получение к элемента по полному ключу;
- `replace` — функция, отвечающая за вставку, удаление и обновление элементов. Принимает старый и новый кортежи, а также `mode` — режим, который разрешает/запрещает вставку неуникальных элементов.
- `create_iterator` — создание итератора, поиск элемента по ключу.
- `begin_build` — функция, вызываемая до начала построения индекса;
- `reserve` — проверка на то, что для построения индекса достаточно памяти;
- `build_next` - добавление кортежа в индекс;
- `end_build` - функция, вызываемая после завершения построения индекса.

Часть из этих функций может не быть реализовано. Например, для `rtree` нет понятие максимального и минимального элементов, а для хэш-таблицы вычисление этих значений является трудоемкой операцией. Подобные функции заменяются generic-версиями, поддерживающими интерфейс, но возвращающими ошибку при попытке использования такой неподдерживаемой функции.

Как было описано в предыдущих частях, вычисленные z-адреса сохраняются в B-дереве. В СУБД Tarantool уже была готовая реализация B+\*-tree, которую было решено переиспользовать.

СУБД Tarantool имеет собственную систему типов. Для индекса было выбрано несколько поддерживаемых типов — `unsigned (unsigned integer)`, `integer`

(signed integer), number (double-precision floating-point number) и string (только префиксный поиск по первым 64 битам). Это достаточно сильно отличает данный индекс от уже существующего R-Tree, предназначенного для работы с числами с плавающей точкой.

Как было сказано, для корректной работы нужно научиться преобразовывать значения определенного типа к некоторым битовым лексикографически упорядоченным словам. Все указанные типы используют 64 бита для хранения, соответствующие им значения решено было хранить как unsigned integer размером 64 бита.

Для unsigned integer преобразование является тривиальным, поскольку в битовые представления и так лексикографически упорядочены.

У знаковых целых чисел (signed integers) старший бит является меткой знака. То есть с точки зрения бинарного представления любое отрицательное число больше любого положительного. Получить значение, удовлетворяющее нужным критериям можно с помощью инверсии старшего бита.

Поиск по строкам, как было сказано, возможен только префиксный. Первые 8 байт любой строки сохраняются как unsigned integer число, в случае если строка короче, она дополняется нулями в конце. Если используется строка в кодировке ASCII, то значения будут уже лексикографически отсортированы. Кодировка UTF-8 обратно совместима с ASCII, поэтому также может использоваться. Такой подход исключает поддержку collation'ов. Но всё-таки поддержку данного типа было решено оставить, поскольку 8 байт может быть вполне достаточно для многих пользовательских сценариев. Кроме того, данный поиск можно использовать как первичный, и при необходимости пользователь сам может выполнить дополнительную фильтрацию.

Наиболее сложный для рассмотрения вариант — числа с плавающей точкой двойной точности (double). Для того, чтобы разобраться с этим случаем следует обратиться к спецификации IEEE 754, говорящего что “если два числа с плавающей запятой одного и того же формата упорядочены (скажем,  $x < y$ ), то они упорядочиваются таким же образом, когда их биты интерпретируются как целые числа со знаками (sign-magnitude integers)”. Нужное нам преобразование имеет следующий вид: для положительных чисел инвертируется старший бит, для отрицательных инвертируются все биты.

Отдельно стоит отметить *null*-значения. Они не поддерживаются в силу того, что невозможно задать битовое значение, которое бы соответствовало

бы отсутствию любого значения. Однако в пользовательском интерфейсе можно будет использовать такие значения для задания минимально возможного и максимально возможного значения ключа. Но это будет рассмотрено в следующих частях.

В реляционных базах данных индексы могут быть уникальными (индекс может содержать один экземпляр определенного значения) и неуникальными. Для Tarantool это тоже справедливо. Предполагается, что для каждого спейса (аналог реляционной таблицы) определен как минимум один индекс. Самый первый индекс называется первичным и должен быть уникальным. Именно в качестве первичного ключа предлагается использовать *z-order curve* по задумке авторов статьи Amazon [13; 14]. Однако в нашем случае было решено поступить иначе, запретить делать индекс на основе кривой *z*-порядка уникальным. Во-первых, подобное ограничение существенно снижает сферы применения данного индекса, во-вторых, возможен случай, когда первые 8 байт разных строк, совпадают, поэтому введение уникальности могло бы привести к нелогичному для пользователя внешнему поведению — вставка двух разных значений приводит к ошибке уникальности, или ещё хуже, использование операции `"replace"` может привести к неожиданной потере информации — кортеж, с одинаковым проиндексированным префиксом просто был бы заменен новым кортежем.

Для многих структур важно иметь стабильный порядок сортировки. Например, итерация по В-дереву, построенному по неуникальным полям. Допустим у нас есть два кортежа:  $[1, 1]$  и  $[3, 1]$ , и мы строим индекс по второму полю, первое же поля является первичным ключом. Внутри В-дерева ссылки на кортежи будут располагаться именно в приведенном порядке. Если мы вставляем элемент сортировка должна сохраняться. При вставке нового кортежа  $[2, 1]$  он должен будет попасть между уже вставленными. Кроме того, по тем или иным причинам наш индекс может перестраиваться. Это будет всегда происходить, например, при перезапуске СУБД, а также к этому может приводить модификация первичного ключа. Как было сказано, физически хранится лишь первичный индекс, вторичный ключ перестраивается при каждом запуске, причем предсказуемо. Возвращаясь к нашему примеру, мы будем получать следующий порядок:  $[1, 1]$ ,  $[2, 1]$ ,  $[3, 1]$ . Рассмотрим, как это обеспечивается. В Tarantool принято сравнивать неуникальные вторичные ключи с учетом первичного ключа. Полученный ключ является уникальным, и таким образом не

возникает проблем с сортировкой неуникальных значений. Спустимся ещё глубже, к тому, как это устроено внутри. Во-первых, вспомним, что кортеж (tuple) — это массив значений, закодированных в формат MessagePack. Поскольку все данные располагаются в оперативной памяти нет смысла хранить проиндексированные поля отдельно, как это обычно делается в реляционных СУБД [24; 25], можно довольно быстро и эффективно обращаться к кортежам, если их понадобилось сравнить. В определении индекса присутствуют 2 значения: *key\_def* (key definition, определение ключа) — перечень проиндексированных полей и *cmp\_def* — *key\_def*, расширенный первичным ключом, всегда уникальный набор значений. Если какое-то поле уже используется во вторичном ключе оно не будет вновь дописано в конец. *key\_def* — это то описание, которое было задано пользователем при создании данного индекса. Однако, если этот ключ неуникальный, во всех внутренних сравнениях будет использоваться *cmp\_def*.

Подобный подход используется и при работе с кривой z-порядка. Однако вместо сравнения таплов используется сравнение z-адресов. Если вдруг оказалось, что два z-адреса равны, мы начинаем честно сравнивать содержимое кортежей, используя расширенное определение ключа — *cmp\_def*. Подобный подход, как и с B-деревом, позволяет поддерживать стабильный порядок сортировки.

Стоит отметить, что не для всех структур это обеспечивается. Например, при работе с хэш-индексом важна скорость доступа к элементу по ключу. Но пренебрегается возможностью обхода данной структуры. Так вставка очередного элемента может приводить к существенному перестроению хэш-таблицы.

Базово работу с индексом можно описывать как независимые 2 части — чтение и запись. Рассмотрим, как работает индекс на основе кривой Мортонна в этих случаях.

Алгоритм записи следующий:

- Извлечь из полученного кортежа его z-адрес;
- Вставить в дерево элемент — структуру, состоящую из z-адреса и указателя на кортеж. Ключом является z-адрес;
- При спуске по дереву мы сравниваем z-адреса. Если два z-адреса равны, то сравниваем поля кортежа;
- В результате мы находим место, куда и сохраняем нашу структуру.

Алгоритм чтения:

- Извлечь z-адрес из ключа поиска;

- Получить итератор на наименьший элемент с таким ключом — *lower\_bound*;
- Последовательно итерироваться по дереву, проверяя каждый элемент на принадлежность к региону поиска с помощью функции *is\_relevant*;
- В случае выхода за границу вычислить точку вхождения в интервал поиска с помощью функции *next\_jump\_in* и сделать прыжок в эту точку;
- Прекратить поиск, если очередной элемент больше, чем верхняя граница — *upper\_bound*.

### 3.5 Lua frontend

Приводится описание того, как конечный пользователь должен выполнять запрос к базе данных с целью сохранения, поиска, удаления и модификации какого-либо элемента.

```

-- Создание space "test" - аналога таблицы в реляционных БД
local space = box.schema.space.create('test',
  { engine = 'memtx' })
-- Создание первичного ключа
5 local pk = space:create_index('pk',
  { type = 'tree', parts = {{1, 'unsigned'}}})
-- Создание индекса на основе z-order curve
-- Индексируемые поля 2 типа "unsigned" и 3 типа "integer"
local sk = space:create_index('secondary',
10 { type = 'zcurve', parts = {{2, 'unsigned'}, {3, 'integer'}}})
-- Вставка кортежа {1, 2, 3} в space
space:replace({1, 2, 3})
-- Прибавление ко второму полю кортежа единицы
space:update({1}, {{ '+', 2, 1 }})
15 -- Удаление кортежа
space:delete({1})
-- Выборка по всему индексу
sk:select()
sk:select({}, { iterator = 'ALL' })
20 -- Выборка в интервале от [0; 2] и [3; 5]
sk:select({0, 2, 3, 5})
sk:select({0, 2, 3, 5}, { iterator = 'EQ' })

```

```

| sk:select({0, 2, 3, 5}, { iterator = 'GE' })
| -- Выборка значений {5; 6}
25 sk:select({5, 6})
| sk:select({5, 6}, {iterator = 'EQ'})
| -- Выборка значений [5; +inf] и [6; +inf]
| sk:select({5, 6}, {iterator = 'GE'})
| -- Выборка значений [-inf; 5] и [-inf; +inf]
30 sk:select({box.NULL, 5, box.NULL, box.NULL}, {iterator = 'GE'})
| -- Удаление индекса
| sk:drop()

```

Как видно, индекс поддерживает 3 итератора «ALL» — выборка всех данных, «EQ» — выборка данных с указанным ключом и «GE» — выборка данных, больше либо равных указанному ключу.

«box.NULL» — специальный символ, указывающий на отсутствие значения. Если он стоит на нечетном месте, то эквивалентен  $-\infty$ , на чётном —  $+\infty$ .



## Глава 4. Результаты

Было проведено несколько тестов:

- Сравнение с R-Tree — поиск точек внутри гиперкуба размерности от 1 до 20;
- Сравнение с B-Tree — поиск точек внутри гиперкуба размерности от 1 до 20;
- Сравнение с B-Tree — префиксный поиск.

В каждом из тестов отдельно интересуют значения времени, потраченного на вставку тестового датасета, время выполнения запросов и объем потребляемой памяти.

Также будет проверено 3 варианта распределения значений в пространстве. В первом случае запрос будет делаться за данными, распределенными равномерно. Т.е. координаты точек — случайно распределенные числа в некотором интервале  $[V_{min}; V_{max}]$ . Во втором случае запрос должен будет вернуть пустое множество — данных удовлетворяющих запросу нет. Это интересно для понимания следующих вещей — существуют ли чтения в таких случаях и сколько будет отрабатывать запрос, в котором не тратится время на сериализацию/десериализацию данных.

И в последнем случае запрос будет высокоселективным, данные будут сконцентрированы вблизи одной точки, большой вклад во время выполнения запроса начнет давать не только время поиска нужных данных, но и время десериализации результатов перед возвратом клиенту.

Кроме того, важно отметить технические особенности платформы, на которой будут выполняться данные измерения.

Hardware:

MacBookPro

Processor: 2.8 GHz Intel Core i7

RAM: 16 GB 2133 MHz LPDDR3

Опции сборки проекта:

Target: Darwin-x86\_64-Release

Build options: `cmake . -DCMAKE_INSTALL_PREFIX=/usr/local -DENABLE_BACKTRACE=ON`

Compiler: Apple Clang 11.0.0 (clang-1100.0.33.8)

C\_FLAGS: -Wno-unknown-pragmas -fexceptions -funwind-tables -fno-omit-frame-pointer -fno-stack-protector -fno-common -msse2 -std=c11 -Wall -Wextra -Wno-strict-aliasing -Wno-char-subscripts

CXX\_FLAGS: -Wno-unknown-pragmas -fexceptions -funwind-tables -fno-omit-frame-pointer -fno-stack-protector -fno-common -msse2 -std=c++11 -Wall -Wextra -Wno-strict-aliasing -Wno-char-subscripts

## 4.1 Сравнение с R-Tree. Поиск точек внутри гиперкуба

### Описание среды и условий проведения измерений

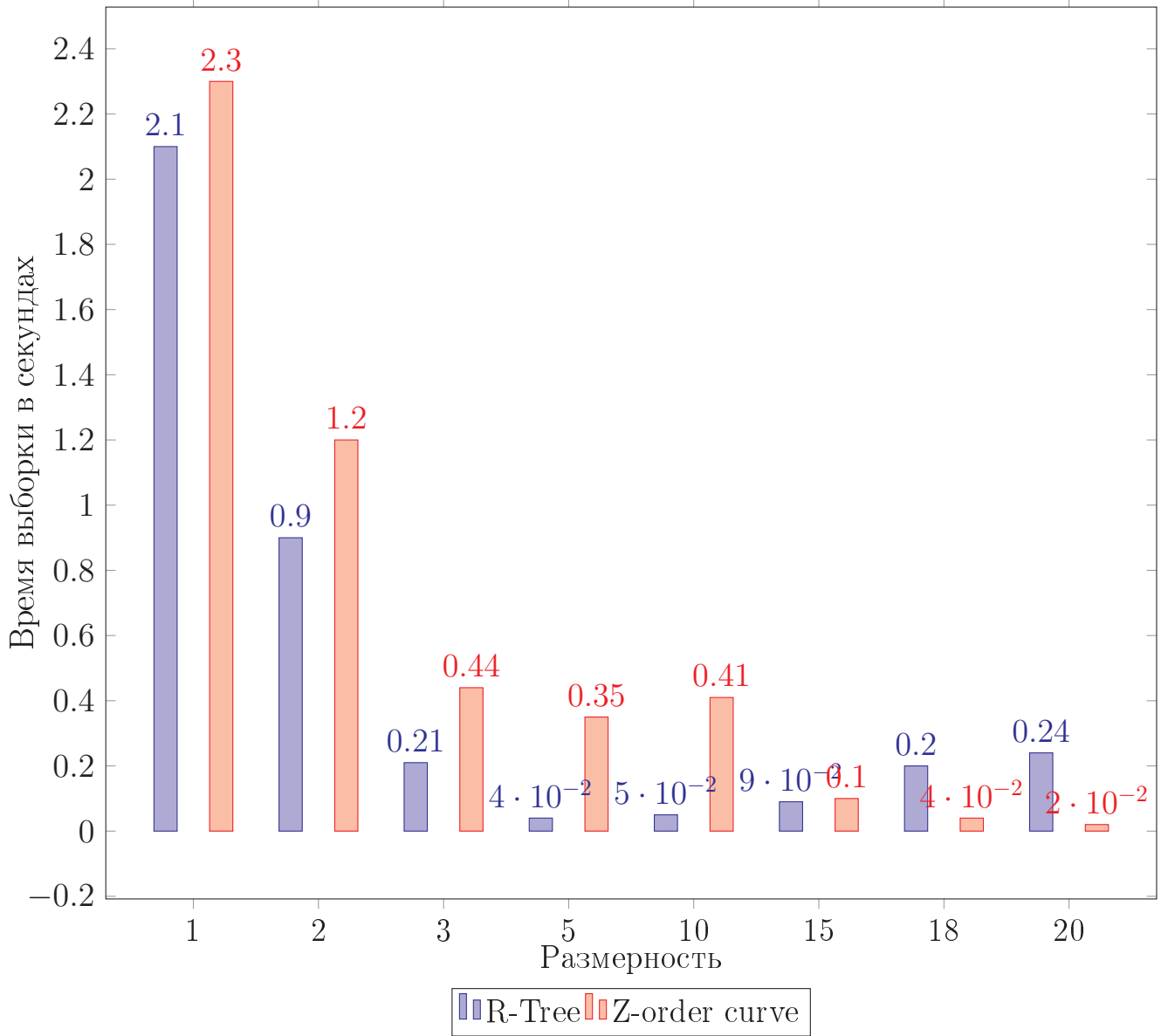
Для каждого из тестов создавались 2 спейса, содержащих 2 индекса — первичный B-Tree и вторичный R-tree или Z-order curve. Последовательно в каждый из спейсов вставлялись приготовленные кортежи, измерялось время вставки и память, которую занимает индекс. Стоит отметить, что для ускорения вставки и исключения факторов, способных её замедлить, были отключены WAL (Write-ahead logs) — запись всех операций в специальный журнал, предотвращающий потерю данных в случае перезагрузки или остановки СУБД. После чего производилось  $N_q$  запросов, и измерялось время, за которое они все были исполнены.

В первом случае равномерного распределения генерировалось  $10^6$  значений в интервале от 0 до  $10^5$ . Область запроса — интервал  $[0.35 \times v_{max}; 0.75 \times v_{max}]$  по каждой из размерностей,  $v_{max}$  — максимально сгенерированное случайное значение. Для усреднения результатов каждый из запросов повторялся 10 раз.

Для тестирования "пустой" выборки ограничивался интервал  $[v_{max}+1; 2 \times v_{max}]$ .

В случае высокоселективной выборки случайным образом для размерности  $D$  генерировались значения  $v_i$  в диапазоне от 0 до  $V = 2.5 \times 10^8$  при этом большая часть значений лежит около 0, ими заполнялись 2 кортежа — для R-Tree (с массивом:  $\{id, \{v_1, v_2, \dots, v_D\}\}$ ) и для Z-Order curve (без массива:  $\{id, v_1, v_2, \dots, v_D\}$ ).  $id$  — первичный ключ, число типа unsigned. Всего  $N_t = 10^6$  значений.

Рисунок 4.1.1 — Сравнение скорости высокоселективной выборки для R-Tree и Z-order curve индексов

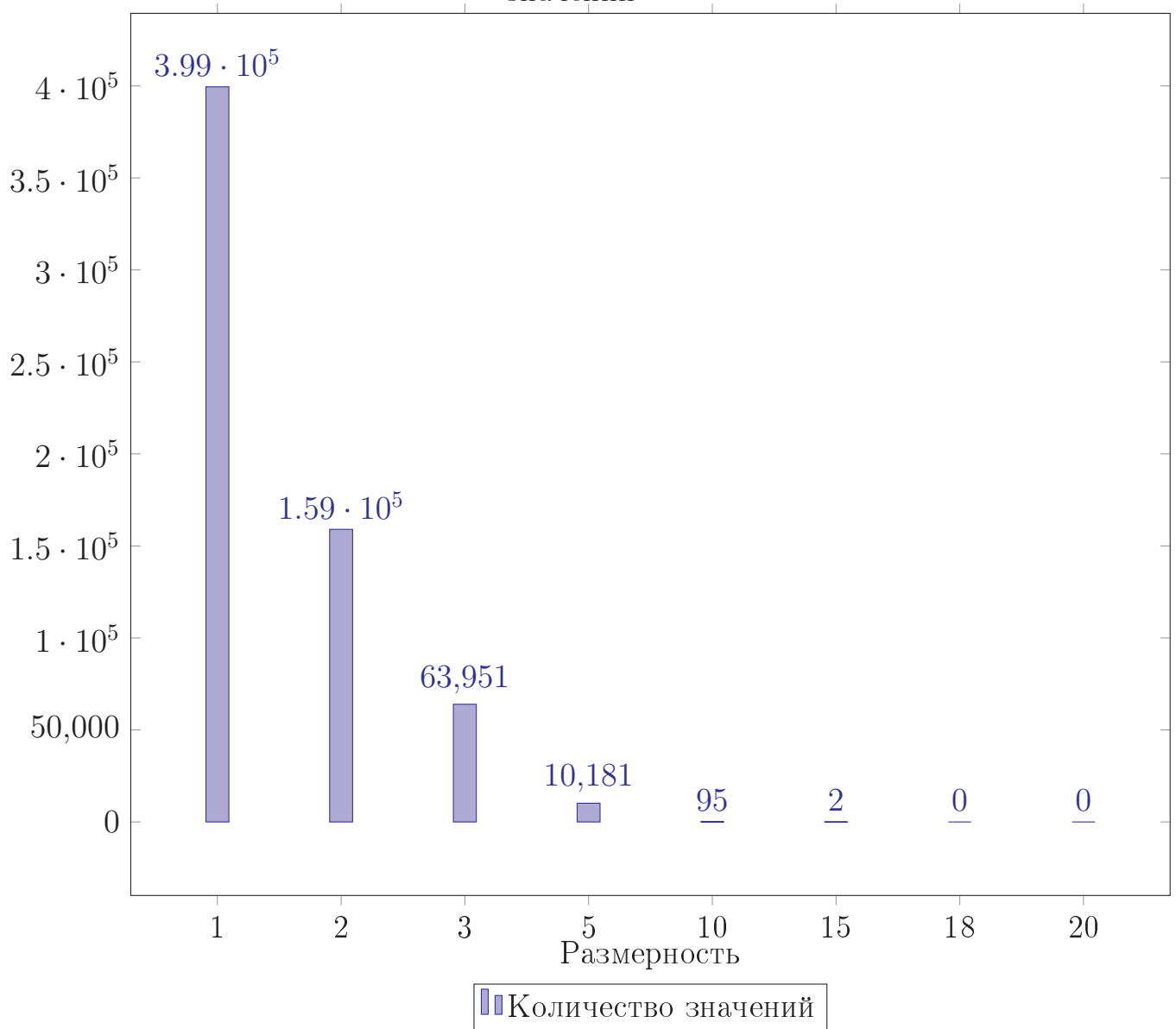


Далее генерировалось 2 числа  $Q_{min}$  и  $Q_{max}$  ( $Q_{min} < Q_{max}$ ) для ограничения области в пространстве  $[Q_{min}; Q_{max}]$  в каждой размерности  $N_q = 10^3$  пар.

## Результаты

Дадим интерпретацию полученным результатам. Во-первых, как видно из рис. 4.1 скорость вставки в Z-order curve индекс выше, чем в R-Tree. Данный

Рисунок 4.1.2 — Селективность запросов в случае равномерно распределенных значений

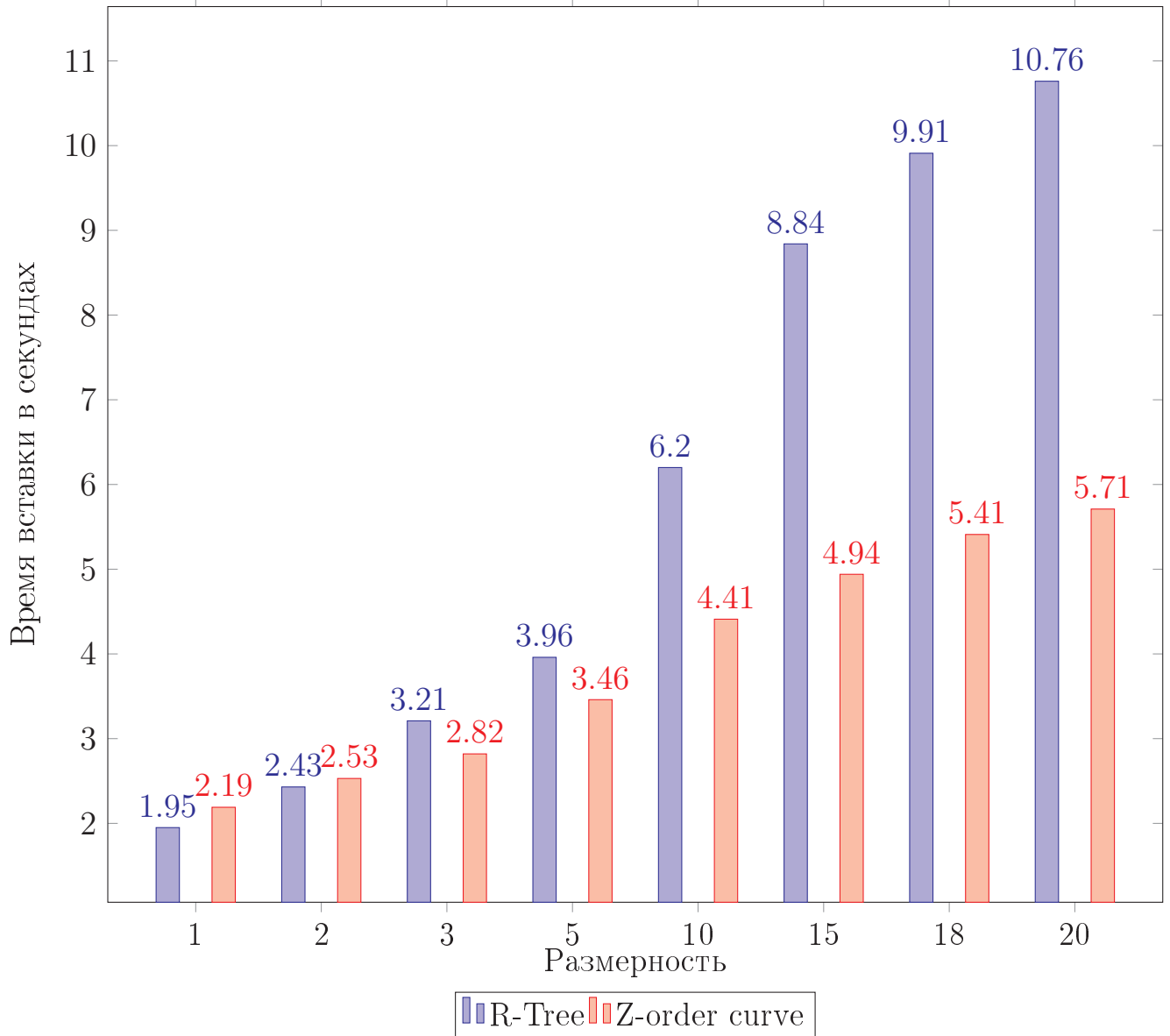


тренд сохраняется и при увеличении размерности. На практике однако результат не особо релевантен, поскольку, как было отмечено выше, тест проводился с выключенным WAL.

С точки зрения расхода памяти оптимальнее Z-order curve (см. рис. 4.1). Причем на больших размерностях практически в три раза. Это уже можно считать релевантным положительным результатом, поскольку объем оперативной памяти обычно ограничен. Но с другой стороны, её стоимость постоянно снижается.

При выполнении запросов, гарантированно возвращающих пустое множество R-Tree и Z-order curve показали себя примерно одинаково (рис. 4.1). Видно,

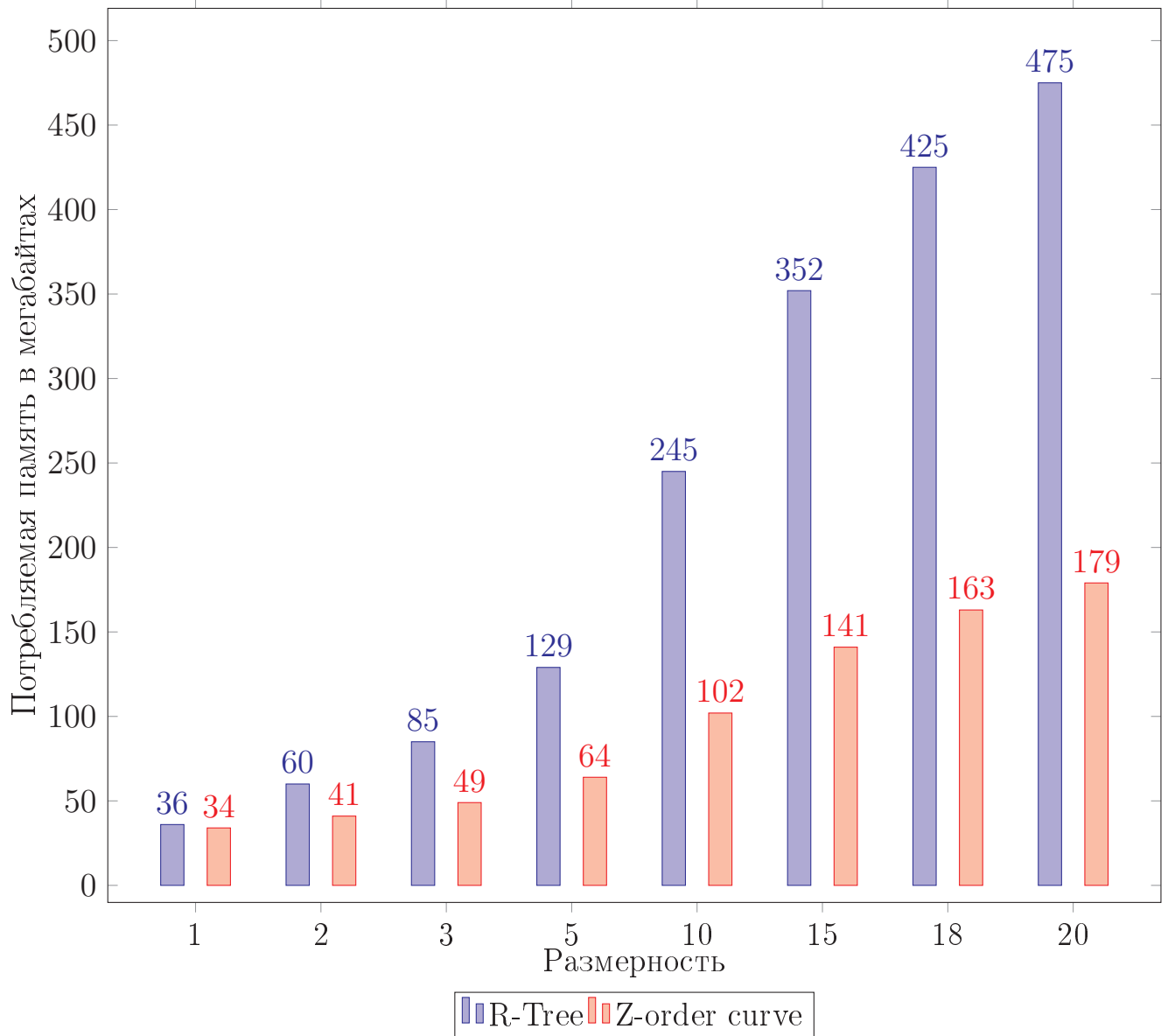
Рисунок 4.1.3 — Сравнение скорости вставки в R-Tree и Z-order curve индексы



что время выполнения не зависит от размерности и нежелательные чтения отсутствуют.

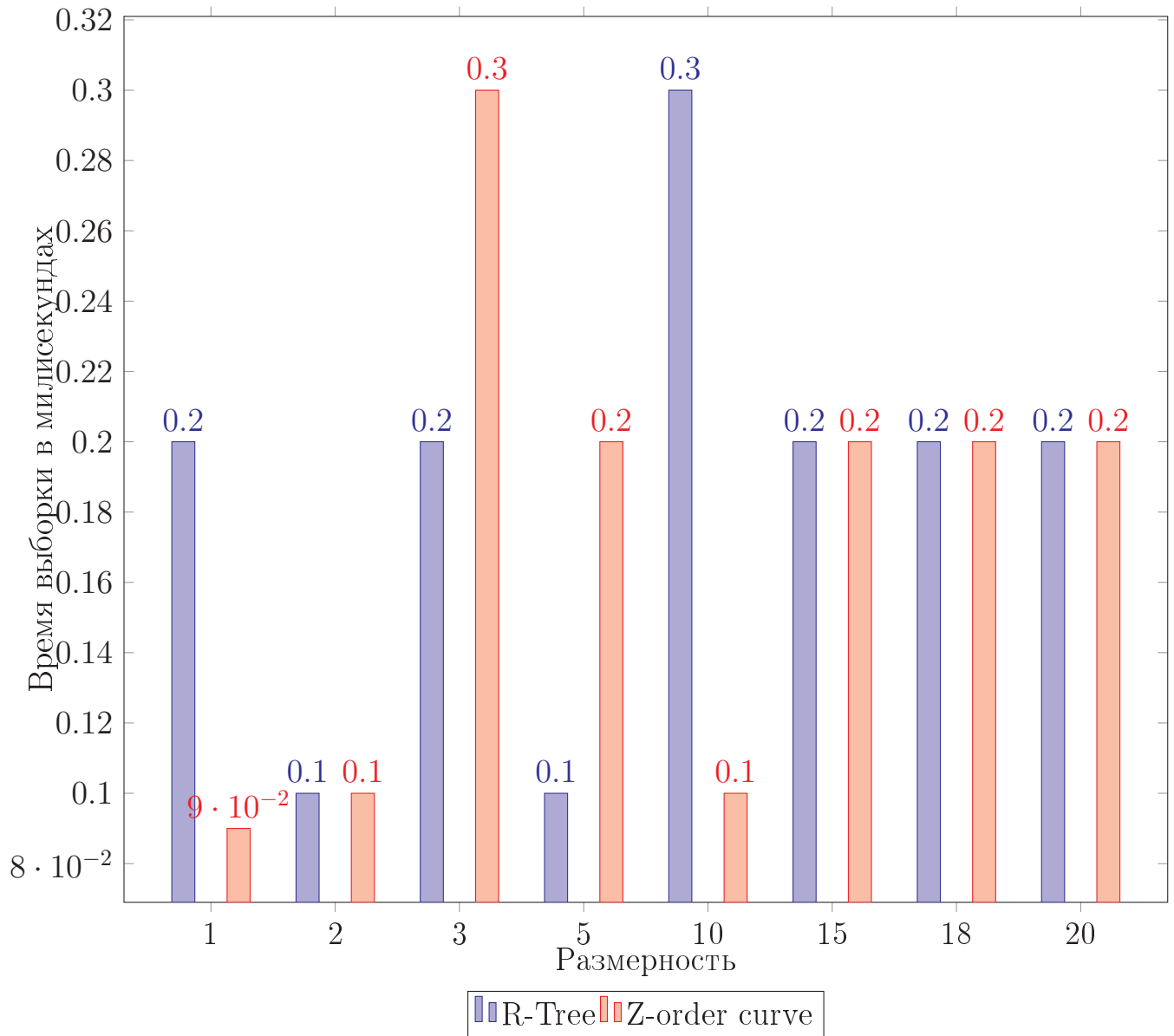
Последний график (рис. 4.1.6), показывающий время выполнения запросов на чтение наиболее интересен. Как видно примерно до размерности 5 время уменьшается, после чего для R-Tree остается практически постоянным, а для Z-order curve начинает драматически возрастать. Дело в том, что при одинаковом количестве точек с увеличением размерности начинает падать селективность наших запросов. Больше размерностей — меньше вероятность того, что координаты точки попадут в интервал между границами запроса. Для R-Tree низкая селективность не критична, поэтому в конце время выполнения запросов практически одинаково. Совершенно иначе обстоит дело с Z-order Curve

Рисунок 4.1.4 — Сравнение памяти, потребляемой индексами



Как только итератор натывается на точку, лежащую вне границ поиска вычисляется Z-адрес следующего вхождения в указанную область. После этого совершается прыжок — спуск по дереву в эту точку. Если данной точки не существует, то мы попадем в следующую по порядку. Данная точка снова может лежать вне области поиска, тогда ситуация будет повторяться вновь и вновь. Это обоснование правой части графика. Слева при высокой селективности большая часть времени может тратиться не на сам поиск, а на то, чтобы упаковать данные и вернуть их клиенту. При работе с внешними коннекторами — это пересылка по сети, при доступе из Lua-приложения это дополнительная работа по сериализации/десериализации Lua-объектов перед возвратом пользователю. Также, если при спуске по R-дереву понять принадлежность точки/области гиперкубу можно за количество операций, зависящих от размерности, то при

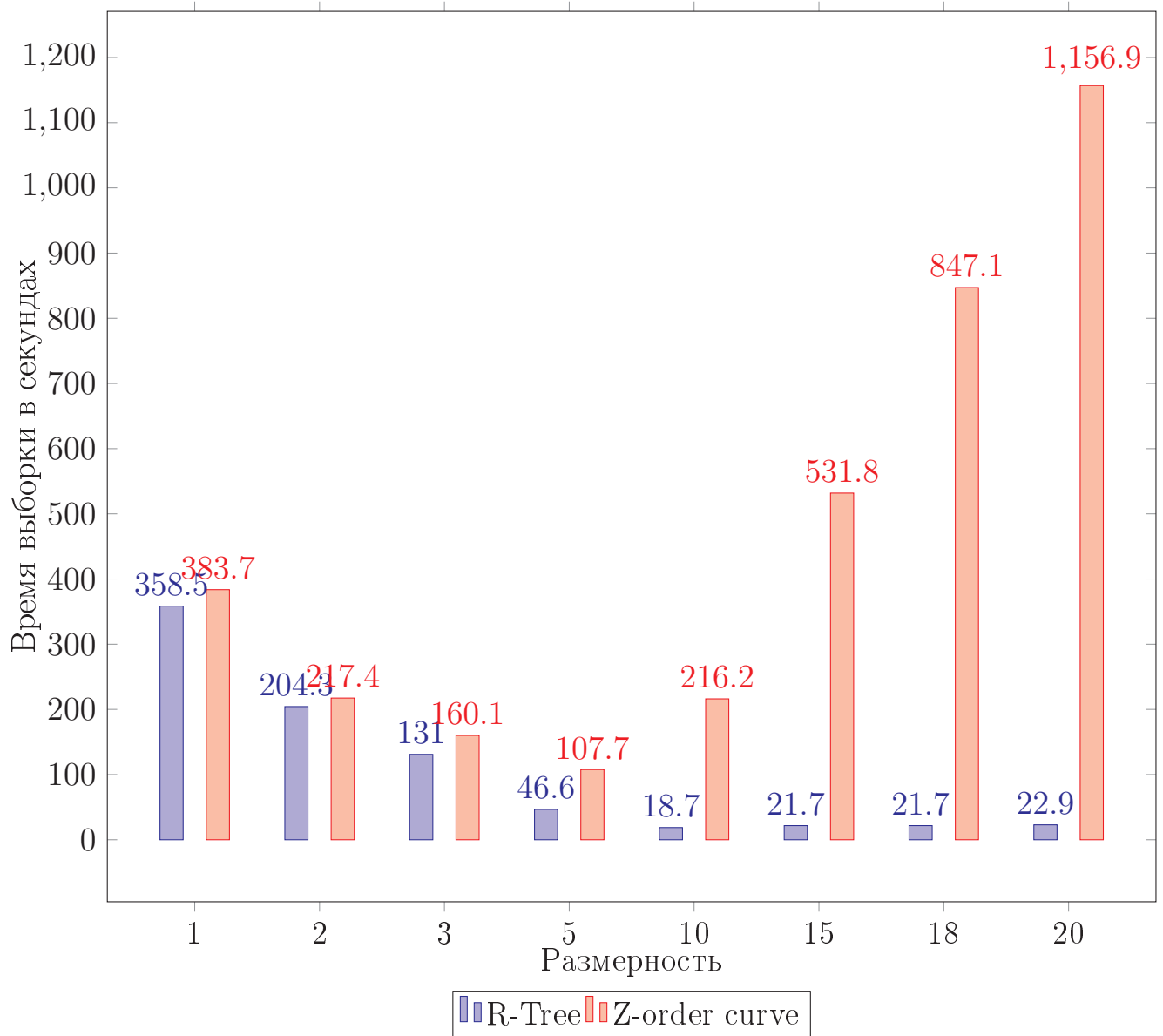
Рисунок 4.1.5 — Сравнение скорости пустой выборки для R-Tree и Z-order curve индексов



работе с Z-адресами алгоритмы линейно зависят от длины этого адреса — размерность, умноженная на достаточно большую константу — 64.

С помощью утилиты "perf top" был проведен анализ функций, потребляющих наибольшее процессорное время. Полученные результаты лишь подтвердили всё вышесказанное (см. рис. 4.1.7).

Рисунок 4.1.6 — Сравнение скорости высокоселективной выборки для R-Tree и Z-order curve индексов



## 4.2 Сравнение с B-Tree. Поиск точек внутри гиперкуба

### Описание среды и условий проведения измерений

В первом случае генерируется  $10^6$  значений равномерно распределенных в интервале  $[0; 10^5]$ . Область запроса — интервал  $[0.35 \times v_{max}; 0.75 \times v_{max}]$  по каждой из размерностей,  $v_{max}$  — максимально сгенерированное случайное значение. Для усреднения результатов каждый из запросов повторялся 10 раз.



Для высокоселективных запросов, как и при сравнении с R-Tree, генерировалось  $N = 10^6$  значений. В том же диапазоне от 0 до  $2.5 \times 10^8$ , однако большая их часть сконцентрирована в районе 0. Значения вставлялись в один спейс, в котором были созданы три индекса — первичный уникальный ключ, B-Tree индекс по первому полю и Z-order curve по второму и всем последующим полям. Далее выбирались точки из гиперкуба в интервале  $[1; v_{max}/4]$ , где  $v_{max}$  — самое большое сгенерированное значение. Чтобы избежать случайных выбросов и усреднить результаты, каждый запрос был сделан 10 раз.

## Результаты и интерпретация

Результаты с нормальным распределением приведены на рисунке 4.2.2. Как и ожидалось, Z-order curve показал себя быстрее, чем B-Tree. Время, которое тратилось на сканирование значений, не входящих в интервал поиска, значительно превышало время, которое затрачивалось на вычисление *next\_intersection\_point* в случае выхода за границу поиска и прыжка к этой точке. При этом заметим, что результаты, полученные для больших размерностей не являются особо показательными, поскольку значений, удовлетворяющих критериям запроса было либо слишком мало, либо совсем не было.

Результаты приведены на рис. 4.2.3

В графиках присутствуют выбросы, но общая тенденция видна — при высокоселективных запросах использование Z-order curve оправдано. Однако с увеличением размерности и уменьшением селективности сканирование начинает быть эффективнее, чем частые спуски по дереву для пропуска интервала, не входящего в область запроса.

## 4.3 Сравнение с B-Tree. Префиксный поиск

Поиск производился по спейсу, заполненную одинаковыми строками. Выполнялся запрос с итератором «EQ» — equal. B-дерево оказалось медленнее

примерно на 15%. Это связано с тем, что существующая реализация не использует дополнительной памяти для хранения проиндексированных значений. Поэтому каждая итерация — это декодинг message pack (см. рис 4.3.1, [20]) строки и затем уже сравнение. В противоположность Z-order curve индекс хранит дополнительно 8 байт на каждое проиндексированное значение. Декодирования на каждой итерации не происходит, что и является причиной подобного прироста производительности.

## 4.4 Вывод

В работе была рассмотрена структура поиска, которая называется кривая Мортонa, позволяющая выполнять запросы за данными, представляющими собой точки в многомерном пространстве, размерностью от 1 до 20. Реализовывалась данная структура на языке «C», далее была встроена в СУБД Tarantool, исходный код которой находится в открытом доступе. Важной целью было также сравнить характеристики данной структуры с уже имеющимися — R-Tree и B-Tree. Основные аспекты — скорость чтения, скорость записи и занимаемая память.

Результаты испытаний показали, что R-Tree быстрее в случае выборки данных особенно при больших размерностях, но является менее компактной и чуть медленнее при вставке.

Использование B-Tree для подобных случаев малоприспособлено, поскольку запрос в общем случае приводит к сканированию большого количества данных, не попадающих в нужный интервал. Однако B-Tree компактно хранит данные и является хорошей базой для реализации многомерных индексов на своей основе — Z-order curve.

Для того, чтобы сделать Z-order curve более пригодным для промышленного использования индексом необходимо работать в сторону улучшения алгоритмов, проверяющих принадлежность точки области поиска, а также нахождения следующей по порядку точки-границы области поиска.

```

27,64% tarantool      [.] z_value_is_relevant
26,61% tarantool      [.] tree_iterator_next
14,44% tarantool      [.] bit_array_get_word
7,49% tarantool       [.] port_c_destroy
5,39% libc-2.27.so    [.] __memmove_avx_unaligned_erms
4,21% tarantool       [.] tuple_to_obuf
3,71% tarantool       [.] port_c_add_tuple
2,70% tarantool       [.] obuf_dup
2,38% tarantool       [.] mslab_alloc
1,52% tarantool       [.] mempool_alloc
1,27% tarantool       [.] mslab_free
0,92% tarantool       [.] box_select
0,57% tarantool       [.] port_c_dump_msgpack_16
0,45% tarantool       [.] iterator_next
0,24% tarantool       [.] memcpy@plt
0,22% tarantool       [.] bit_array_length
0,06% tarantool       [.] slab_get_with_order
0,03% tarantool       [.] slab_put_with_order
0,03% tarantool       [.] ev_run
0,02% tarantool       [.] epoll_poll
0,01% tarantool       [.] mslab_tree_next.part.4
0,01% libc-2.27.so    [.] epoll_wait
0,01% tarantool       [.] bit_array_cmp

```

а) выборка

```

5,15% tarantool      [.] memtx_zcurve_insert.constprop.28
4,09% tarantool      [.] bit_array_cmp
3,80% tarantool      [.] ev_async_send
3,62% tarantool      [.] pipecb
3,14% tarantool      [.] epoll_poll
3,13% libpthread-2.27.so [.] __pthread_mutex_lock
3,12% tarantool      [.] ev_run
2,74% tarantool      [.] memtx_tree_insert
2,44% tarantool      [.] bit_array_interleave
2,15% tarantool      [.] ev_feed_event
2,13% [vdso]          [.] __vdso_clock_gettime
2,12% tarantool      [.] tuple_field_raw_by_path
1,99% tarantool      [.] ev_invoke_pending
1,97% tarantool      [.] tuple_compare_slowpath<false, false, false, false>
1,60% libc-2.27.so    [.] epoll_wait

```

б) вставка

Рисунок 4.1.7 — Потребление процессорного времени

Рисунок 4.2.1 — Сравнение скорости выборки для B-Tree и Z-order curve индексов в случае нормального распределения данных

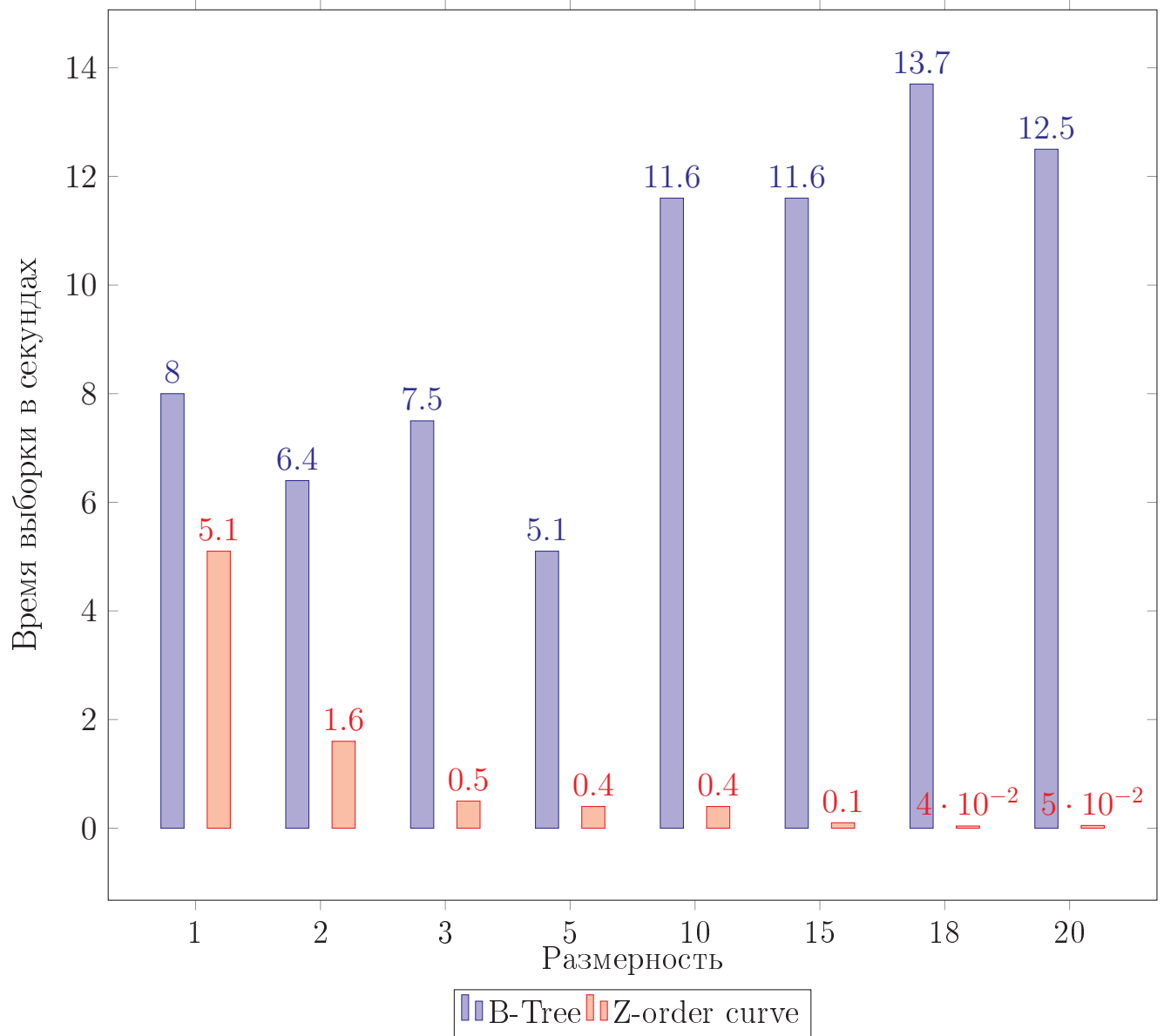


Рисунок 4.2.2 — Селективность тестовых запросов

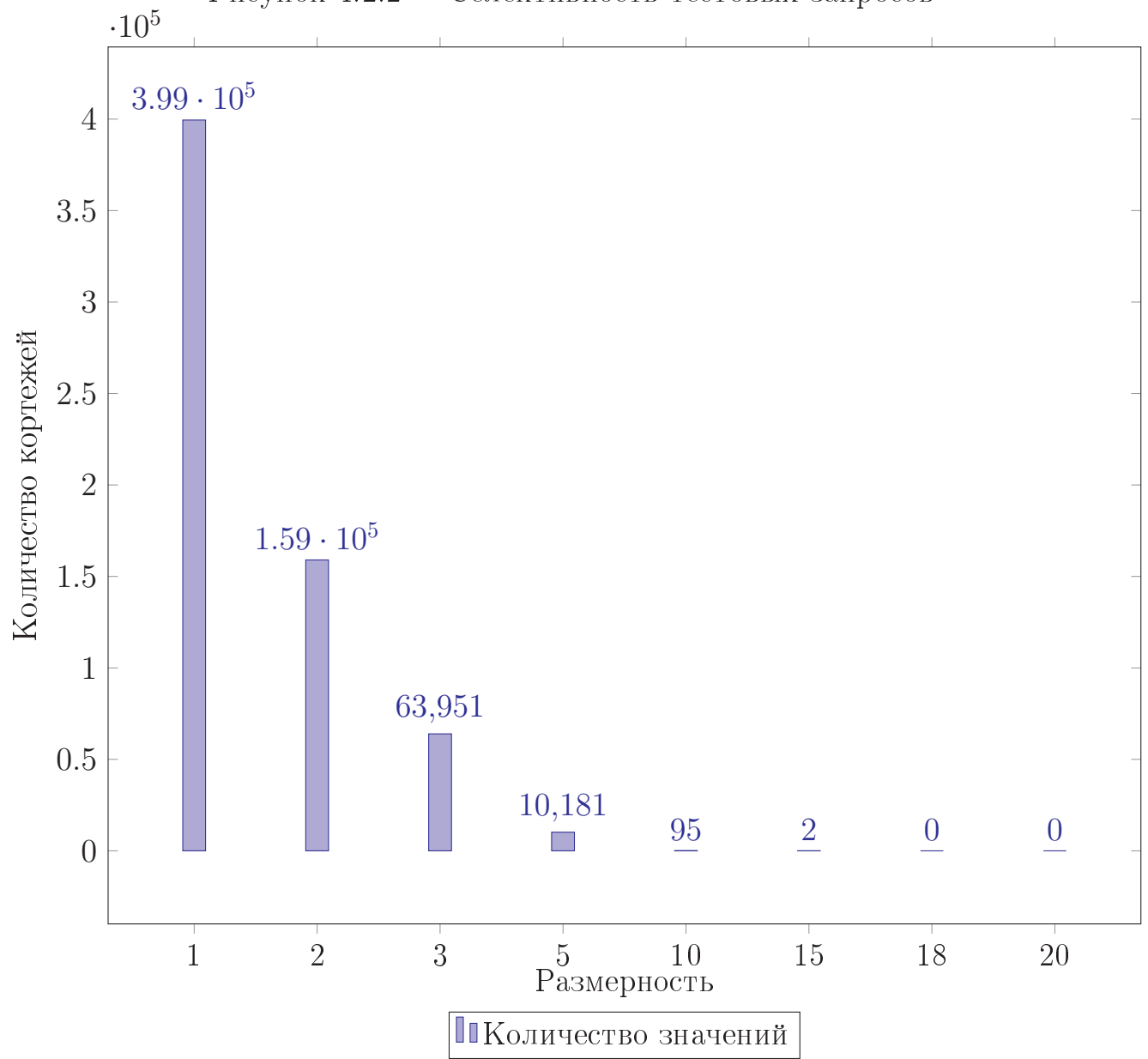


Рисунок 4.2.3 — Сравнение скорости высокоселективной выборки для B-Tree и Z-order curve индексов

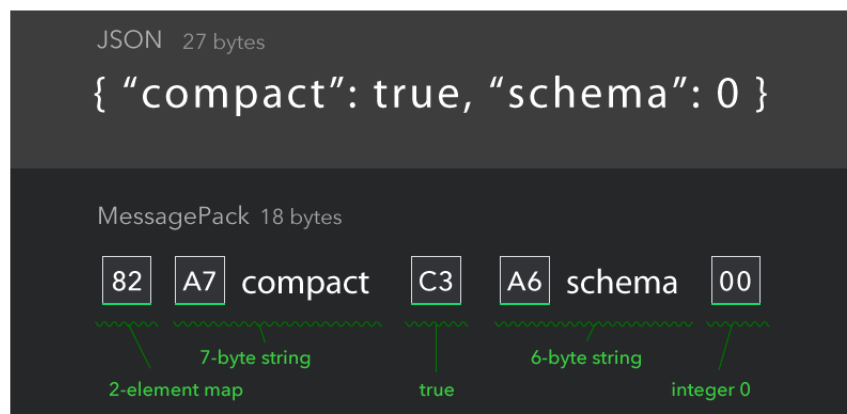
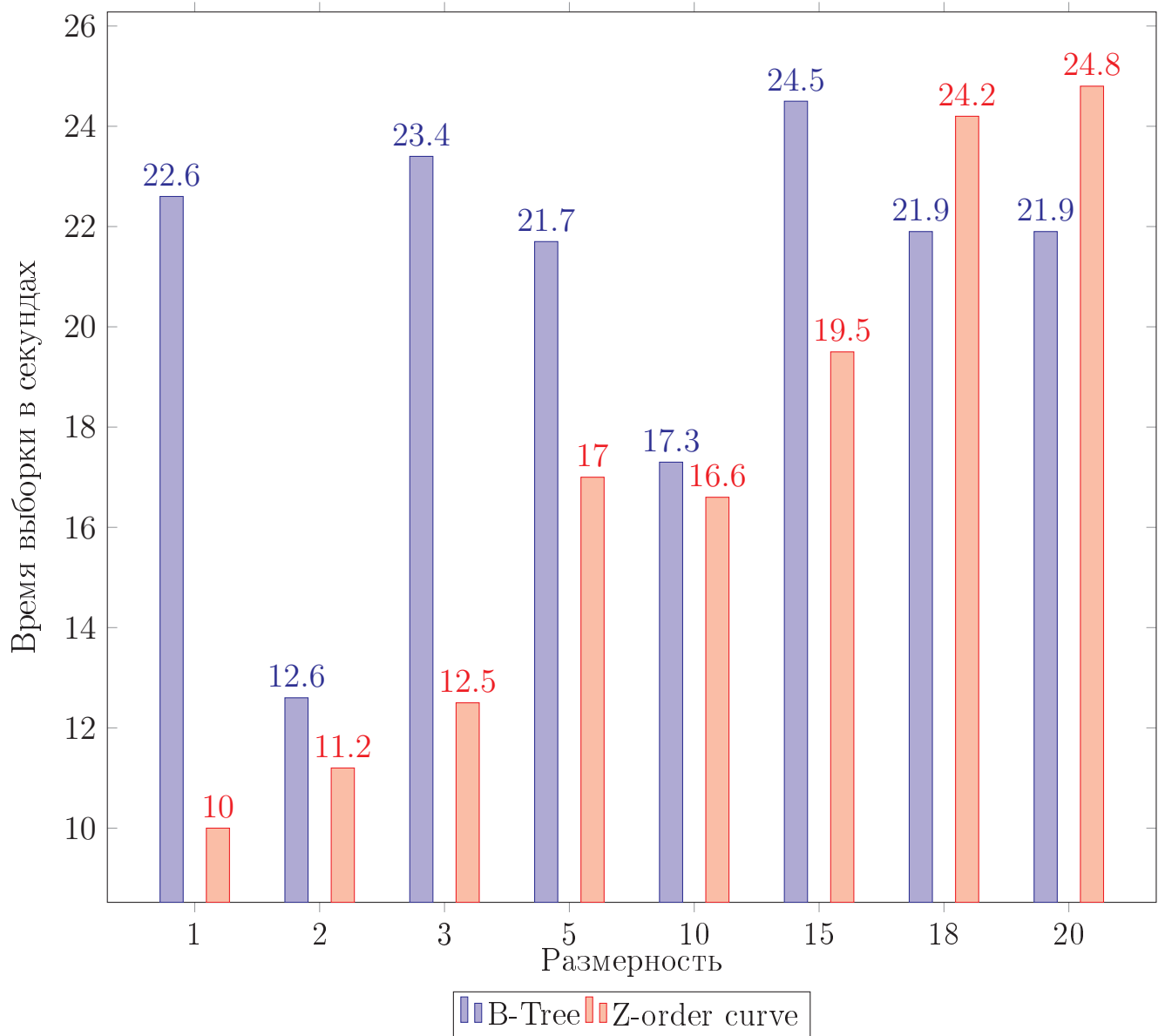


Рисунок 4.3.1 — Сравнение хранения в JSON-формате (plain text). И message pack (binary)

## Список литературы

1. *Comer, D.* Ubiquitous B-tree / D. Comer // ACM Computing Surveys (CSUR). — 1979. — Т. 11, № 2. — С. 121—137.
2. *Bayer, R.* Organization and maintenance of large ordered indexes / R. Bayer, E. McCreight // Software pioneers. — Springer, 2002. — С. 245—262.
3. The log-structured merge-tree (LSM-tree) / P. O’Neil [и др.] // Acta Informatica. — 1996. — Т. 33, № 4. — С. 351—385.
4. *Obe, R.* PostGIS in action / R. Obe, L. Hsu // GEOInformatics. — 2011. — Т. 14, № 8. — С. 30.
5. *Kothuri, R. K. V.* Quadtree and R-tree indexes in oracle spatial: a comparison using GIS data / R. K. V. Kothuri, S. Ravada, D. Abugov // Proceedings of the 2002 ACM SIGMOD international conference on Management of data. — 2002. — С. 546—557.
6. *Guttman, A.* R-trees: A dynamic index structure for spatial searching / A. Guttman // Proceedings of the 1984 ACM SIGMOD international conference on Management of data. — 1984. — С. 47—57.
7. The R\*-tree: an efficient and robust access method for points and rectangles / N. Beckmann [и др.] // Proceedings of the 1990 ACM SIGMOD international conference on Management of data. — 1990. — С. 322—331.
8. *Morton, G. M.* A computer oriented geodetic data base and a new technique in file sequencing / G. M. Morton. — 1966.
9. Integrating the UB-tree into a database system kernel. / F. Ramsak [и др.] // VLDB. Т. 2000. — 2000. — С. 263—272.
10. Processing Relational Queries Using a Multidimensional Access Technique / V. Markl [и др.]. — Ios Press, 1999.
11. *Widhopf-Fenk, R. J.* Advanced Concepts and Applications of the UB-tree : дис. ... канд. / Widhopf-Fenk Robert Josef. — Technische Universität München, 2005.

12. MD-HBase: A scalable multi-dimensional data infrastructure for location aware services / S. Nishimura [и др.] // Mobile Data Management (MDM), 2011 12th IEEE International Conference on. Т. 1. — IEEE. 2011. — С. 7—16.
13. *Slayton, Z.* Z-Order Indexing for Multifaceted Queries in Amazon DynamoDB: Part 1 / Z. Slayton. — 2017. — URL: <https://aws.amazon.com/ru/blogs/database/z-order-indexing-for-multifaceted-queries-in-amazon-dynamodb-part-1/> (дата обр. 07.10.2018).
14. *Slayton, Z.* Z-order indexing for multifaceted queries in Amazon DynamoDB: Part 2 / Z. Slayton. — 2018. — URL: <https://aws.amazon.com/ru/blogs/database/z-order-indexing-for-multifaceted-queries-in-amazon-dynamodb-part-2/> (дата обр. 07.10.2018).
15. *Gaede, V.* Multidimensional access methods / V. Gaede, O. Günther // ACM Computing Surveys (CSUR). — 1998. — Т. 30, № 2. — С. 170—231.
16. *Lawder, J. K.* Querying multi-dimensional data indexed using the Hilbert space-filling curve / J. K. Lawder, P. J. H. King // ACM Sigmod Record. — 2001. — Т. 30, № 1. — С. 19—24.
17. *Danial, A.* CLOC—Count lines of code, 2009 / A. Danial.
18. *Ierusalimschy, R.* Lua 5.1 reference manual / R. Ierusalimschy, L. H. De Figueiredo, W. Celes. — 2006.
19. *Pall, M.* The luajit project / M. Pall. — 2008.
20. *Furuhashi, S.* MessagePack / S. Furuhashi // URL: <https://msgpack.org>. — 2013.
21. *Бабин, О.* Как написать свой индекс в Tarantool / О. Бабин ; Mail.Ru Group. — URL: <https://habr.com/ru/company/mailru/blog/505880/> (дата обр. 08.06.2020).
22. *Prukl, A.* A relational approach to indexing / A. Prukl. — 2007.
23. *Knuth, D.* The Art of Computer Programming: Volume 3: Sorting and Searching / D. Knuth. — Pearson Education, 1998.
24. *Liu, L.-C. H.* Secondary index search / L.-C. H. Liu, K. Yoneda. — 7 24.2001. — US Patent 6,266,660.
25. *Owens, M.* SQLite / M. Owens, G. Allen. — Springer, 2010.



