ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(ФГБОУ ВО «ВГУ»)

На правах рукописи

Потапов Данила Романович

РАЗРАБОТКА И ИССЛЕДОВАНИЕ МЕТОДА И АЛГОРИТМОВ АДАПТАЦИИ АССОЦИАТИВНОГО КОНТЕЙНЕРА ДАННЫХ

Специальность 05.13.17 – Теоретические основы информатики

Диссертация
на соискание ученой степени
кандидата физико-математических наук

Научный руководитель:
доктор физико-математических наук,
профессор Артемов Михаил Анатольевич

Воронеж – 2021

Оглавление

Введение 6

Актуальность работы 6

Степень разработанности темы 7

Цель и задачи исследования 8

Методология и методы исследования 9

Научная новизна 9

Степень достоверности результатов 10

Теоретическая и практическая значимость 10

Апробация работы 10

Публикации 10

Личный вклад автора 11

Область исследования 11

На защиту выносятся 11

Структура и объём работы 12

Глава 1. Анализ существующих методов построения и оптимизации контейнеров данных «ключ-значение» 13

1.1. Анализ существующих методов построения контейнеров одномерных данных «ключ-значение» для использования в самоадаптирующихся контейнерах данных 13

1.1.1. Отсортированные ассоциативные контейнеры 17

1.1.2. Древовидные структуры 19

1.1.3. Кучи 22

1.1.4. Хеширование 23

1.2. Обзор методов построения контейнеров многомерных данных 24

1.2.1. Иерархическое представление многомерной информации 27

1.2.1.1. R-дерево 27

1.2.1.2. BSP-дерево 29

1.2.2. Многомерное хеширование 30

1.2.3. Кривые, заполняющие пространство 30

1.2.4. Многомерные методы доступа для больших размерностей 32

1.3. Анализ применения существующих ассоциативных структур данных в самоадаптирующемся контейнере данных 33

1.4. Обзор существующих алгоритмов кэширования 37

1.5. Выводы 42

Глава 2. Разработка и исследование способа задания самоадаптирующегося контейнера данных 44

2.1. Структурная модель самоадаптирующегося контейнера данных 46

2.2. Анализ условий адаптации самоадаптирующихся контейнеров данных ... 48

2.2.1. Объем информации 48

2.2.2. Количество запросов 50

2.2.3. Архитектура информационной системы 51

2.3. Способ задания ассоциативного контейнера данных 52

2.4. Способ задания нагрузки на ассоциативный контейнер данных 53

2.5. Моделирование процесса применения нагрузки на контейнер данных 55

2.6. Способ задания самоадаптирующегося контейнера данных 56

2.7. Критерий выбора оптимального контейнера 57

2.8. Анализ оптимального размера кэша по временному критерию для нагрузки, сгенерированной по нормальному распределению 58

2.9. Модель нагрузки из смеси нормальных распределений 60

2.10. Способ задания адаптивного кэширующего контейнера данных с использованием интервального статистического ряда 61

2.11. Выводы 64

Глава 3. Анализ и разработка алгоритмов модулей самоадаптирующегося контейнера с использованием кэша 65

3.1. Алгоритм кэширующего модуля самоадаптирующегося контейнера 65

3.2. Алгоритм работы модуля определения параметров сложной нагрузки на самоадаптирующиеся контейнеры данных 75

3.3. Алгоритм работы адаптивного кэширующего контейнера данных с использованием интервального статистического ряда 78

3.4. Выводы 88

Глава 4. Вычислительные эксперименты 89

4.1. Исследование эффективности применения кэша в зависимости от среднеквадратичного отклонения и соотношения скоростей хранилищ для нормального распределения 89

4.1.1. Описание условий тестирования 89

4.1.2. Вычислительный эксперимент и анализ 91

4.2. Исследование эффективности модуля определения параметров сложной нагрузки на самоадаптирующиеся контейнеры данных 102

4.2.1. Описание условий тестирования 102

4.2.2. Эксперименты 102

4.2.3. Анализ времени выполнения 107

4.3. Исследование эффективности адаптивного кэширующего контейнера данных с использованием интервального статистического ряда 110

4.3.1. Описание условий тестирования 110

4.3.2. Вычислительный эксперимент 112

4.4. Выводы 119

Заключение 121

Список литературы 122

Введение

Актуальность работы

Во многих задачах обработки и хранения информации явно или неявно требуются эффективно работающие контейнеры данных (структуры данных) [1]. Существует много различных типов таких контейнеров, однако каждый из них является оптимальным только для ограниченного числа задач. При этом зачастую условия задачи меняются в процессе работы системы и, следовательно, для максимально эффективной работы системы периодически необходима смена типа контейнера. Таким образом, возникает необходимость создания самоадаптирующегося ассоциативного контейнера данных, который меняет логику своей работы в зависимости от нагрузки. Например, при большом количестве операций поиска контейнер работает на основе одной структуры данных, при большом количестве вставок – на основе другой, при определенном соотношении операций вставки, поиска и удаления – на основе третьей и т.д. Помимо обеспечения максимальной скорости работы должна использоваться та структура данных, которая занимает минимум дополнительной памяти.

Одним из очевидных случаев использования самоадаптирующихся контейнеров являются мобильные устройства, такие как планшетные ПК, смартфоны, мобильные телефоны, ноутбуки и др. Такие устройства в силу своих размеров имеют ограниченные ресурсы, существенно уступающие стационарным компьютерам [2]. Ограниченными являются в первую очередь оперативная память и производительность процессора. Поэтому и появляется необходимость в адаптивных контейнерах данных. При изменяющейся нагрузке выбирается контейнер, оптимальный либо с точки зрения быстродействия, либо – минимальных затрат оперативной памяти.

Еще одним примером необходимости использования адаптивного контейнера данных является индекс базы данных. Данный объект используется для повышения эффективности поиска в базе данных, а, следовательно, для этого применяются структуры данных, оптимизированные для поиска. Однако

перестроение таких контейнеров при большом количестве вставок требует много времени [3]. Таким образом, использование различных структур данных в зависимости от нагрузки позволит увеличить эффективность индексов в базах данных.

Однако наибольший прирост производительности при использовании контейнеров, которые изменяют свое поведение при изменении условий работы или нагрузки, можно получить в условиях Highload (высоконагруженные системы) [4], NoSQL (различные реализации хранилищ баз данных, которые существенно отличаются от реляционных и имеют возможность линейно масштабироваться) [5] и Bigdata (структурированные и неструктурированные данные огромных объемов и методы работы с ними) [6], т. к. именно в этих областях присутствует большое количество данных и большие нагрузки. В этом случае помимо основного контейнера зачастую используется такой метод оптимизации как кэширование. Таким образом, помимо смены основного контейнера, самоадаптирующийся контейнер данных может изменять такие параметры кэша как алгоритм вытеснения или размер кэша.

Данное исследование является актуальным, так как в настоящее время не существует универсального контейнера и алгоритма кэширования, в связи с чем существует необходимость динамически выбирать оптимальную структуру данных и алгоритм кэширования для каждого конкретного случая на основе различных критериев [3].

Степень разработанности темы

Проблемой оптимальной структуры данных занимались многие авторы. Алгоритмы и реализации одномерных контейнеров описаны в работах Г. Адельсон-Вельского, Е. Ландиса, Aragon C. R., Seidel R., Fredman M. L., Tarjan R. E. и др. Многомерные структуры данных описаны в исследованиях Гулакова В.К., Трубакова А.О., Kriegel H., Seeger B., Bentley J. L., Six H.-W., Bentley J.L. и многих других авторов.

В этих исследованиях рассматриваются различные структуры данных, но ни в одном из исследований не рассмотрен универсальный контейнер данных, который был бы максимально эффективен в любых условиях и для любых нагрузок. Следовательно, разработка и реализация методов и алгоритмов самоадаптирующегося контейнера данных, является весьма актуальной задачей [3]. Решению этой актуальной задачи посвящена данная диссертация.

Цель и задачи исследования

Цель данной работы состоит в разработке самоадаптирующегося контейнера данных. Для достижения этой цели необходимо решить следующие задачи:

1. Разработать метод адаптации ассоциативного контейнера данных.
2. Определить область применения существующих контейнеров данных и алгоритмов кэширования для реализации самоадаптирующегося контейнера данных.
3. Исследовать зависимость эффективности кэша от среднеквадратичного отклонения и соотношения скоростей хранилищ для нормального распределения [7].
4. Разработать и реализовать алгоритм определения параметров сложной нагрузки на самоадаптирующиеся контейнеры данных [8].
5. Разработать и реализовать алгоритм адаптивного кэширующего контейнера данных с использованием интервального статистического ряда.
6. Провести вычислительные эксперименты, подтверждающие эффективность предложенных алгоритмов.

Объект исследования – структуры данных и алгоритмы доступа к данным.

Предмет исследования – структура данных, которая изменяет логику работы в зависимости от нагрузки и условий работы [7].

Методология и методы исследования

В качестве теоретической и методологической основы диссертационного исследования использованы методы дискретной математики, системного анализа, математической статистики, теория разработки программного обеспечения.

Научная новизна

В диссертационной работе получены следующие результаты, отличающиеся научной новизной:

1. Предложен метод адаптации ассоциативного контейнера данных, позволяющий изменять структуру данных основного хранилища данных, алгоритм кэширования и размер кэша в зависимости от условий работы и нагрузки на контейнер [8].
2. Выявлена зависимость размера кэширующего контейнера от соотношения между скоростями работы кэша и основного хранилища [8] и среднеквадратичного отклонения ключей при нормальном распределении, позволяющая определить минимальный по времени работы размер кэша.
3. Разработан алгоритм определения параметров сложной нагрузки, состоящей из смеси гауссовских распределений, отличающийся применением EM-алгоритма с использованием буфера.
4. Создан алгоритм адаптивного кэширующего контейнера данных с использованием интервального статистического ряда, позволяющий повысить процент попаданий в кэш.
5. На основе разработанных алгоритмов реализованы программные комплексы, позволяющие провести исследования их эффективности.

Степень достоверности результатов

Научные положения, теоретические выводы и практические рекомендации обоснованы корректным использованием математического аппарата и подтверждены вычислительными экспериментами на ЭВМ.

Теоретическая и практическая значимость

Теоретическая значимость заключается в создании метода и алгоритмов самоадаптирующегося контейнера данных и некоторых его модулей, а также в выявлении зависимости оптимального размера кэша от условий работы и распределения ключей. Практическая значимость заключается в программных реализациях разработанных в диссертации алгоритмов.

Апробация работы

Результаты работы были представлены в форме докладов на следующих конференциях: XX Международной конференции «Информатика: проблемы, методы, технологии», IPMT-2020 (Воронеж, 2020 г.), международных научно-технических конференциях «Актуальные проблемы прикладной математики, информатики и механики» (Воронеж, 2018 – 2019 гг.), XVI Международной научно-методической конференции: «Информатика: проблемы, методология, технологии» (Воронеж, 2016 г.), «Математическое и компьютерное моделирование, информационные технологии управления (МКМИТУ-2016)» (Воронеж, 2016 г.).

Публикации

Основные результаты работы опубликованы в 12 печатных работах, 5 из которых опубликованы в рекомендуемых ВАК РФ рецензируемых научных изданиях. Кроме того, есть публикация за рубежом, проиндексированная в Scopus. Также автором было получено свидетельство о государственной регистрации

программы для ЭВМ. Из совместных работ в диссертацию вошли только результаты, принадлежащие лично диссертанту.

Личный вклад автора

Все основные результаты, составившие диссертацию, получены автором лично. Соавторы публикаций по теме диссертации участвовали в обсуждении постановочной части решаемых задач и результатов вычислений по разработанным автором программам расчётов. В диссертации отсутствуют заимствованные материалы и результаты других авторов исследований, а когда по логике изложений такие сведения оказываются необходимыми, то они снабжены соответствующими ссылками на авторов и литературные источники.

Область исследования

Диссертационная работа соответствует следующим пунктам паспорта специальности 05.13.17 – Теоретические основы информатики:

1. Исследование, в том числе с помощью средств вычислительной техники, информационных процессов, информационных потребностей коллективных и индивидуальных пользователей.

2. Исследование информационных структур, разработка и анализ моделей информационных процессов и структур.

5. Разработка и исследование моделей и алгоритмов анализа данных, обнаружения закономерностей в данных и их извлечениях разработка и исследование методов и алгоритмов анализа текста, устной речи и изображений.

На защиту выносятся

1. Метод адаптации ассоциативного контейнера, позволяющий изменять структуру данных основного хранилища данных, алгоритм кэширования и размер кэша в зависимости от условий работы и нагрузки на контейнер.

2. Результат исследования зависимости оптимального размера кэша от соотношения скоростей хранилищ и среднеквадратичного отклонения ключей в нагрузке, определенной по нормальному распределению.
3. Алгоритм работы модуля определения параметров сложной нагрузки [8] на контейнер, состоящей из смеси нормальных распределений.
4. Алгоритм работы адаптивного кэширующего контейнера данных с использованием интервального статистического ряда.
5. Программный комплекс, реализующий разработанные алгоритмы.

Структура и объём работы

Материал работы изложен на 137 страницах машинописного текста. Работа состоит из введения, четырех глав, выводов, списка литературы, содержит 33 рисунка и 27 таблиц. Библиография включает 151 наименование.

Глава 1. Анализ существующих методов построения и оптимизации контейнеров данных «ключ-значение»

1.1. Анализ существующих методов построения контейнеров одномерных данных «ключ-значение» для использования в самоадаптирующихся контейнерах данных

Методы построения контейнеров данных «ключ-значение» можно разделить на две категории. Одна группа методов подразумевает использование некоторого глобального упорядочения (числового или лексикографического) [9]. Ключи хранятся в отсортированном состоянии, а для поиска используется алгоритм бинарного поиска. Контейнеры, полученные данным методом, называются отсортированными ассоциативными контейнерами. Примерами таких контейнеров являются различные деревья.

Второй группой методов является хеширование, а контейнеры, полученные данным методом, называются хешированными ассоциативными контейнерами. В отличии от отсортированных ассоциативных контейнеров такие структуры данных не хранят данные в упорядоченном виде, и позволяют проводить поиск только на равенство ключей. Примерами таких контейнеров являются различные вариации хеш-таблицы.

На рисунке 1.1 изображена классификация одномерных ассоциативных контейнеров, проанализированных в данной части главы [10].

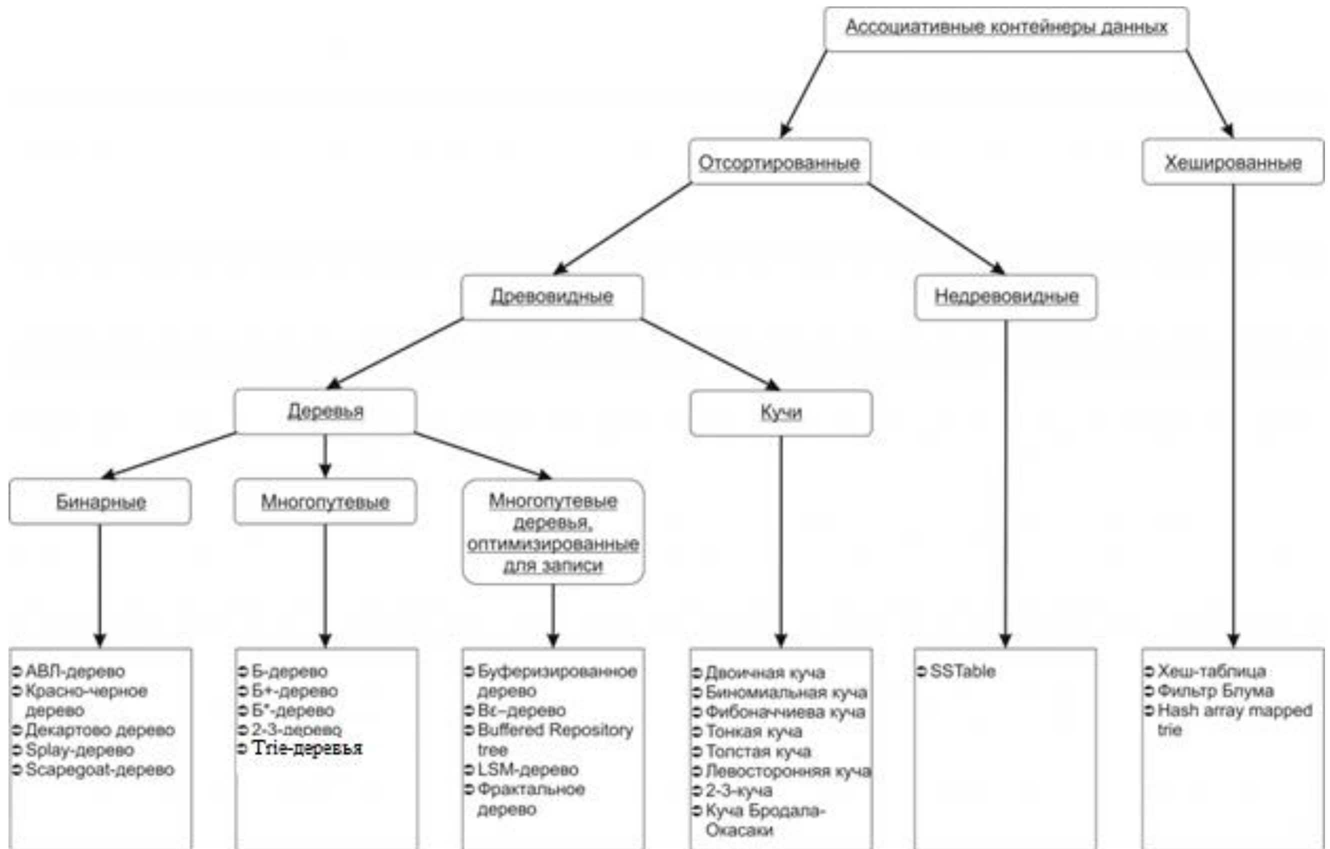

Рис. 1.1. Классификация одномерных ассоциативных контейнеров

Одномерные ассоциативные структуры данных используются для решения большого количества задач в операционных системах и СУБД, алгоритмах обработки, хранения и сортировки данных и многих других алгоритмах и системах. В таблице 1.1 отображены основные области использования этих контейнеров на практике.

Таблица 1.1 – Области применения основных одномерных контейнеров данных

Контейнер	**Области использования**
SSTable	NoSQL СУБД (Cassandra, HBase и LevelDB).

АВЛ-дерево	Высокоэффективные алгоритмы сортировки и поиска (подсистема управления памятью ОС Linux), шахматные програмы, искуственный интеллект.
Красно-черное дерево	Высокоэффективные алгоритмы сортировки и поиска (ядро ОС Linux, файловая система ext3), реализация set и map.
Декартово дерево	Сбор статистики по большому числу параметров.
Splay-дерево	Эффективная обработка часто повторяющихся запросов (сетевые маршрутизаторы, кеш, ОС Windows NT, gcc компилятор и GNU C++ библиотека).
Scapegoat-дерево	Работа с многомерными объектами (k-d дерево, квадродерево, ортогональные запросы).
B+ дерево	Файловые системы для хранения каталогов и индексирования метаданных (NTFS, BeFS и другие), реляционные СУБД – в качестве индекса (Oracle, SQLite и другие), NoSQL базы данных – для доступа к данным (CouchDB).
B* дерево	Файловые системы для хранения каталогов и индексирования метаданных (HFS+, Microsoft's NTFS, AIX (jfs2), btrfs, Ext4), реляционные СУБД – в качестве индекса (PostgreSQL, MySQL), NoSQL базы данных – в качестве индекса (MongoDB).
2-3-4-дерево	Хранение словарей во внутренней памяти.
Буферизированное дерево	Очередь с приоритетами, range и segment деревья, алгоритм сортировки.
Bε–дерево	СУБД TokuDB и файловая система BetrFS.
Buffered Repository tree	Обход графа в ширину и глубину, СУБД TokuDB.

LSM-дерево	Хранение данных в СУБД (BigTable, HBase, LevelDB, MongoDB, SQLite4, RocksDB, WiredTiger, Apache Cassandra и InfluxDB, Tarantool).
Фрактальное дерево	Индексы в СУБД TokuDB и TokuMX, файловые системы TokuFS и BetrFS.
Куча	Пирамидальная сортировка, приоритетная очередь (библиотеки Java, C++, PHP, Python и др., планировщики задач в различных ОС, сетевые маршрутизаторы), алгоритмы на графах (алгоритм Дейкстры, Хаффмана и Прайма), поиск минимума/максимума.
Хеш-таблица	Реализация ассоциативного массива и set (Java, Perl, Ruby, Python, PHP и др.), индексы и хранение данных в СУБД (MySQL,Oracle,MS SQL, Memcached, MemcacheDB, REDIS, Tarantool и многие другие), кэши.
Фильтр Блума	Прокси сервер Squid, браузер Chrome, проверка существования записи в СУБД (BigTable, Apache HBase, Apache Cassandra, Postgresql, Oracle, RocksDB), криптовалюта Bitcoin, веб-сервер Akamai, программы проверки орфографии.
Hash array mapped trie	Реализация персистентных структур данных (Scala, Clojure, Haskell и другие), проверка целостности в файловых системах (IPFS, Btrfs,ZFS), протоколы (Bittorent, Dat, Apache Wave), криптовалюты Bitcoin и Ethereum, СУБД (Apache Cassandra, Riak, Dynamo) [10].

Далее представлен обзор и анализ существующих контейнеров одномерных данных «ключ-значение» таких как SSTable, различные бинарные и многопутевые деревья, деревья с использованием буфера, LSM дерево и фрактальное дерево, основные реализации кучи, хеш-таблица, фильтр Блума и HAMT.

1.1.1. Отсортированные ассоциативные контейнеры

В отсортированном ассоциативном контейнере все ключи отсортированы в некотором порядке [11]. Простейшим примером такого контейнера является Sorted String Table (SSTable) [12]. Этот контейнер является одним из самых популярных для хранения, обработки и обмена большими наборами данных. Он используется в таких известных NoSQL СУБД как Cassandra [13], HBase и LevelDB. Данный контейнер хорошо подходит для ситуаций, когда необходимо обработать гигабайты или даже терабайты информации и выполнить на них последовательность из заданий Map-Reduce [14]. В этом случае из-за размера входных данных затраты на операции чтения и записи будут преобладать над затратами на выполнение полезной работы. Таким образом, использование случайного чтения/записи крайне неэффективно; вместо этого можно использовать потоковое чтение и запись, что позволяет значительно уменьшить расходы на операции чтения/записи. SSTable позволяет использовать такое чтение, за счет своего внутреннего устройства. Данный контейнер представляет собой последовательность отсортированных по ключу пар «ключ-значение». Следовательно, временная сложность основных операций над этим контейнером соответствует временной сложности отсортированного массива. SSTable позволяет хранить дубликаты и не имеет ограничений на размер ключей и значений. Кроме того, для данного контейнера можно обеспечить доступ по ключу за константное время. Для этого необходимо создать отсортированный индекс, последовательно считывая данные из контейнера. Индекс будет упорядочен по ключу и будет содержать смещения до соответствующих значений, как показано на рисунке 1.2.

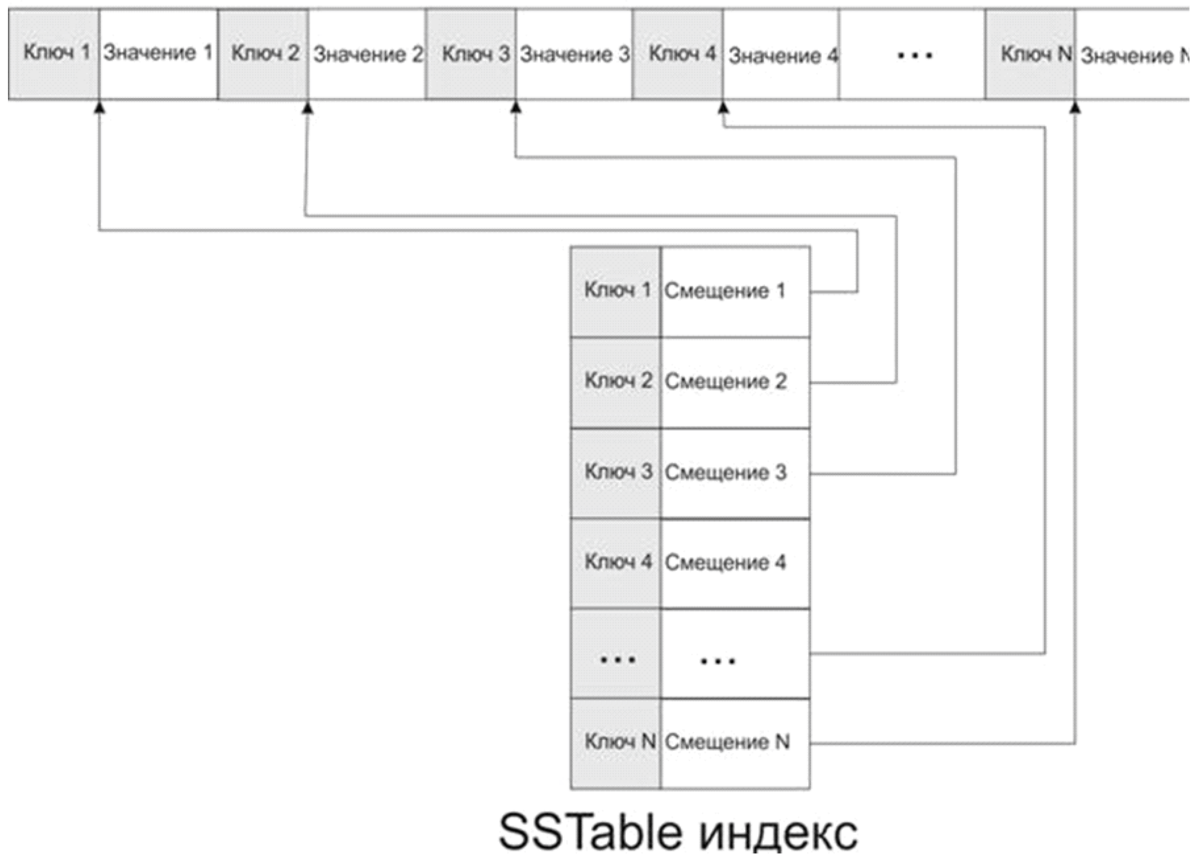

Рис. 1.2. SSTable

Главное преимущество SSTable состоит в простоте и эффективности хранения и обработки большого объема данных, но у данного контейнера существует и значительный минус – любое изменение контейнера (удаление или вставка элемента) будет требовать больших затрат. Это связано с тем, что SSTable представляет собой отсортированный по ключу массив и для его изменения необходимо перестроить весь контейнер.

Таким образом, SSTable лучше всего использовать для хранения статических индексов, так как для извлечения с диска можно последовательно считывать данные, установив головку диска всего один раз. Достаточно быстро происходит и случайное чтение. Следовательно, любая операция по изменению данного контейнера является слишком длительной и дорогостоящей, особенно, когда SSTable располагается не в оперативной памяти. Поэтому обычно в NoSQL СУБД SSTable, которые располагаются на диске, являются неизменяемыми.

Для решения основной проблемы SSTable (крайне неэффективной случайной записи) был разработан другой контейнер – LSM (Log-Structured Merge) дерево.

При использовании данного контейнера в самоадаптирующихся контейнерах [1] необходимо учитывать, что перестроение в SSTable из других контейнеров в основном требует больших затрат (особенно если SSTable располагается на диске), так как данные в памяти или на диске должны быть упорядочены по ключу. Данный контейнер следует использовать в самоадаптирующихся контейнерах только тогда, когда известно, что длительное время контейнер будет использоваться только для частого чтения больших данных, а, следовательно, преимущества последовательного чтения перекроют затраты на построение.

Помимо SSTable существует другая группа отсортированных ассоциативных контейнеров – деревья поиска [10].

1.1.2. Древовидные структуры

Древовидная структура данных [15] – это динамическая связанная структура, в которой связи между элементами не линейны как в списке, а похожи на ветви дерева. Существует две категории данных структур, которые различны по методам построения и обработки. Первая – это *деревья*, вторая – *кучи*. Кроме того, деревья различают по другим атрибутам:

1. Сбалансированность. Дерево может быть:
    a. вырожденным;
    b. идеально сбалансированным;
    c. сбалансированным;
    d. несбалансированным и невырожденным.
2. Количество ветвей. Дерево может быть:
    a. двоичным;
    b. многопутевым, когда количество ветвей больше двух [10].

1.1.2.1. Бинарные деревья

Самым простым деревом поиска является бинарное (двоичное) дерево [16]. Данные деревья являются одними из самых популярных в связи с простой реализацией и достаточно высокой эффективностью. Главным достоинством бинарного дерева является возможность реализовать высокоэффективные алгоритмы сортировки и поиска, построенные на его основе. Кроме того, это дерево используется при реализации контейнеров set и map в c++ [17] и TreeSet и TreeMap в Java [18].

Бинарные деревья могут быть вырожденными, сбалансированными, идеально сбалансированными или не относится ни к одной из этих категорий. На практике обычно используются сбалансированные деревья, так как вырожденные деревья превращаются в список, а идеально сбалансированные трудоемки в построении, и балансировка в них происходит достаточно часто.

Существует несколько реализаций сбалансированного дерева [31]:

1. *АВЛ-дерево* [16];
2. *Красно-черное дерево* [19];
3. *Декартово дерево* [20];
4. *Splay-дерево* [21];
5. *Scapegoat-дерево* [22].

1.1.2.2. Многопутевые деревья

Помимо бинарных деревьев существуют и деревья с количеством ветвей больше двух. Такие деревья называются *многопутевыми* или *сильноветвящимися*. Самым распространенным является B-дерево и его различные модификации [23]. B-дерево используется в системах баз данных (индексы во многих современных СУБД) и файловых системах.

У данного дерева существует большое количество вариаций. Ниже представлены самые известные из них.

1. *B+-дерево*, также известное как дерево Байера-Баума [24].

2. *B*-дерево* [25].

3. *2-3-дерево* [16].

Другим известным многопутевым деревом является префиксное дерево и его модификации. *Префиксное дерево* (бор, луч, нагруженное дерево, англ. Trie) [26] хранит ассоциативный массив, элементы которого в качестве ключей имеют строки. Каждый узел хранит некоторый символ, в корне хранится пустая строка. При этом ключи не хранятся явно в дереве, а вычисляются путем конкатенации символов по пути от корня к листам, в которых и хранятся значения, соответствующие искомым ключам. У данного дерева существует большое количество модификаций [27] сжатые префиксные деревья (компактное префиксное дерево, англ. Patricia tree, radix tree, crit bit tree), тернарные деревья поиска (англ. Ternary Search Trees(TST)), Level Path Compressed(LPC) tries, Burst tries, Array Mapped Trie (AMT), Hash Array Mapped Trie (HAMT) и др [10].

1.1.2.3. Многопутевые деревья, оптимизированные для записи

На протяжении долгого времени B-дерево не имело альтернатив среди структур данных для хранения во внешней памяти. Но в последнее время ситуация стала меняться из-за необходимости обработки постоянно растущих объемов данных. Для достижения этой цели появились структуры данных, оптимизированные специально для операций записи. Некоторые из этих структур частично уступают B-дереву в скорости поиска, но позволяют более эффективно осуществлять вставку и удаление. Существует несколько типов таких структур данных:

- деревья, использующие буфер:
  - *Буферизированное дерево* [28] – (a, b)-дерево (это сбалансированное сильно ветвящееся дерево, где каждый узел имеет степень O(m), а не B) с коэффициентами a = m/4, b = m, построенное над множеством O(n) листьев, каждый из которых содержит O(B) элементов;
  - *Bε–деревья* [29] представляют собой класс деревьев, которые отличаются от буферизированных деревьев методом поиска данных и

наличием параметра ε. При ε = 1 получается обычное B-дерево (размер буфера равен 0), а при ε = 0 получается модификация буферизированного дерева – Buffered Repository tree [30].

- LSM [31] (и его модификации BLSM-дерево [32] и Diff-Index [33]) и фрактальные деревья [10] [34].

1.1.3. Кучи

Другой древовидной структурой данных является *куча* [35]. В отличии от деревьев куча упорядочена по уровням – значения элементов следующего уровня всегда больше (или меньше), чем значения предыдущего. Таким образом корнем кучи является максимальный (минимальный) элемент всей кучи. Данная структура данных реализует следующие операции:

1. Поиск минимума/максимума;
2. Удаление минимума/максимума;
3. Увеличение/уменьшение значения ключа;
4. Добавление элемента в кучу;
5. Слияние двух куч с образованием новой, содержащей все элементы обеих куч.

Куча широко используется для различных задач. На основе данной структуры данных реализована очередь с приоритетом и пирамидальная сортировка, различные алгоритмы поиска и алгоритмы на графах [36].

Существует несколько вариантов кучи. Основные перечислены далее. Сравнение различных вариантов кучи по временной сложности представлено в таблице 1.2.

Таблица 1.2 – Временные сложности куч

Операция	Двоичная	Биномиальная	Фибоначчиева	2-3 куча	Бродал	Левосторонняя куча
найти минимум	Θ(1)	Θ(log *n*)	Θ(1)	Θ(1)*	Θ(1)	Θ(1)
удалить минимум	Θ(log *n*)	Θ(log *n*)	Θ(log *n*)*	Θ(log *n*)*	Θ(log *n*)	Θ(log *n*)
добавить	Θ(log *n*)	O(log *n*)	Θ(1)	Θ(1)*	Θ(1)	Θ(log *n*)
уменьшить ключ	Θ(log *n*)	Θ(log *n*)	Θ(1)*	Θ(1)*	Θ(1)	Θ(log *n*)
слияние	Θ(*n*)	Θ(log *n*)**	Θ(1)	Θ(1)*	Θ(1)	Θ(log *n*)

(*) Амортизационное время, (**) n — размер наибольшей кучи

1. *Двоичная куча* [35], также известная как *пирамида*;
2. *Биномиальная куча* [35];
3. *Фибоначчиева куча* [37];
4. *Тонкая куча* (англ. Thin heap) [38];
5. *Толстая куча* (англ. Thick heap, Fat heap) [38];
6. *Левосторонняя куча* (англ. leftist heap) [39];
7. *2-3-куча* (2-3 heap) [40];
8. *Куча Бродала-Окасаки* (англ. Brodal's and Okasaki's Priority Queue) [41].

1.1.4. Хеширование

Другим методом построения ассоциативных контейнеров является *хеширование* [19]. Идея хеширования используется во многих контейнерах данных, но самым используемым считается *хеш-таблица*. Этот контейнер реализует ассоциативный массив, причем индексы в нем определяются с использованием функции хеширования. Основная идея данного метода

построения заключается в том, чтобы сузить множество ключей, т. е. ключи, полученные в результате применения функции хеширования, имеют меньше уникальных значений, чем исходные ключи. Следовательно, хеш-коды могут совпадать при различных входных данных согласно принципу Дирихле. Ситуация, которая возникает при этом, называется *коллизией*. Коллизии значительно ухудшают производительность хеш-таблицы. Поэтому было создано несколько методов для борьбы с ними:

1. *Метод цепочек.*
2. *Линейное разрешение коллизий (метод открытой адресации).*
3. *Двойное хеширование.*
4. *Хеширование кукушки* [42].
5. *Идеальное хеширование* [43].
6. *Хеширование «Классики» (Hopscotch hashing)* [44].

Хеш-таблица является самым распространенным контейнером, использующим хеширование, но помимо нее существует несколько других хешированных ассоциативных контейнеров. Ниже перечислены наиболее известные:

1. *Фильтр Блума* (англ. Bloom filter) [45];
2. *Hash array mapped trie* (HAMT) [46].

1.2. Обзор методов построения контейнеров многомерных данных

Большинство объектов в реальном мире являются многомерными или обладают большим количеством параметров, которые также приводятся к многомерным величинам. До недавнего времени обработка многомерных объектов осуществлялась только путем их трансформации к одномерным. При этом такой переход значительно уменьшает эффективность обработки и анализа. Однако в последнее время данная область стала бурно развиваться и появилось большое количество различных алгоритмов и контейнеров многомерных данных [47]. Рост

заинтересованности в обработке таких данных связан с несколькими областями науки и производства.

Многомерные контейнеры и алгоритмы обработки данных используются:

1) для работы с многомерными векторами. Такие вектора применяются, например, в физике для вычислений, связанных с многомерными полями (в вычислительной гидродинамике, электромагнетизме), в биологии — для поиска неполных совпадений в белковых и ДНК-последовательностях. Помимо указанных областей, данные вектора широко применяются в экономике, политологии, медицине, технике.

2) в компьютерной графике и анимации для отсечения невидимых частей, трассировки лучей, обнаружения столкновений и других задач.

3) для работы с пространственными данными, например, с географическими координатами. В первую очередь такие данные используются в географических информационных системах [48] [49].

4) в мультимедийных базах данных, системах виртуальной и дополненной реальности для поиска похожих изображений, поиска контуров по известной базе, различного анализа и других задач [50].

Помимо более сложной структуры самих данных, многомерная структура данных отличается от одномерной необходимостью поддержки дополнительных запросов [51]. Стандартными запросами при работе с многомерными объектами считаются:

- запрос объекта, который полностью совпадает с заданным (exact-match query);
- запрос объекта, содержащего заданную точку (point query);
- запрос объектов, которые имеют хотя бы одну точку в заданном регионе (range query);
- запрос объектов, которые имеют пересечение с заданной областью (intersection/overlap query);

- запрос объекта, который полностью содержит в себе заданный объект (enclosure query);
- запрос объектов, которые полностью входят в заданный объект (containment query);
- запрос объектов, которые имеют с заданным общие границы (adjacency query);
- запрос объекта, который находится ближе всего к заданному (nearest neighbor query). Является самым часто используемым запросом для многомерных данных, в силу широкой применимости в различных приложениях;
- запрос на все пары объектов из двух множеств, удовлетворяющих заданному предикату (spatial join).

Существует несколько классификаций методов построения контейнеров многомерных данных [52]. Методы разделяются по:

1) месту расположения:

а) в оперативной памяти;

б) во внешней памяти.

2) методу доступа к данным [53]:

а) точечный метод доступа (PAM). В этом случае объект воспринимается как точка в многомерном пространстве, которая имеет координату по каждой размерности;

б) пространственный метод доступа (SAM). Объект имеет геометрические размеры и не может восприниматься как точка.

2) методу разбиения:

а) разбиение на основе пространства (space driven partitioning);

б) разбиение на основе данных (data driven partitioning).

3) методу построения контейнера [54]:

а) иерархический (различные деревья);

б) хеширование;

в) кривые, заполняющие пространство;

г) большое количество различных гибридных методов.

В данной части будут рассмотрены методы построения контейнеров данных (классифицированные по методу построения, рис. 1.3) и проведен анализ их использования в самоадаптирующихся контейнерах данных [55].

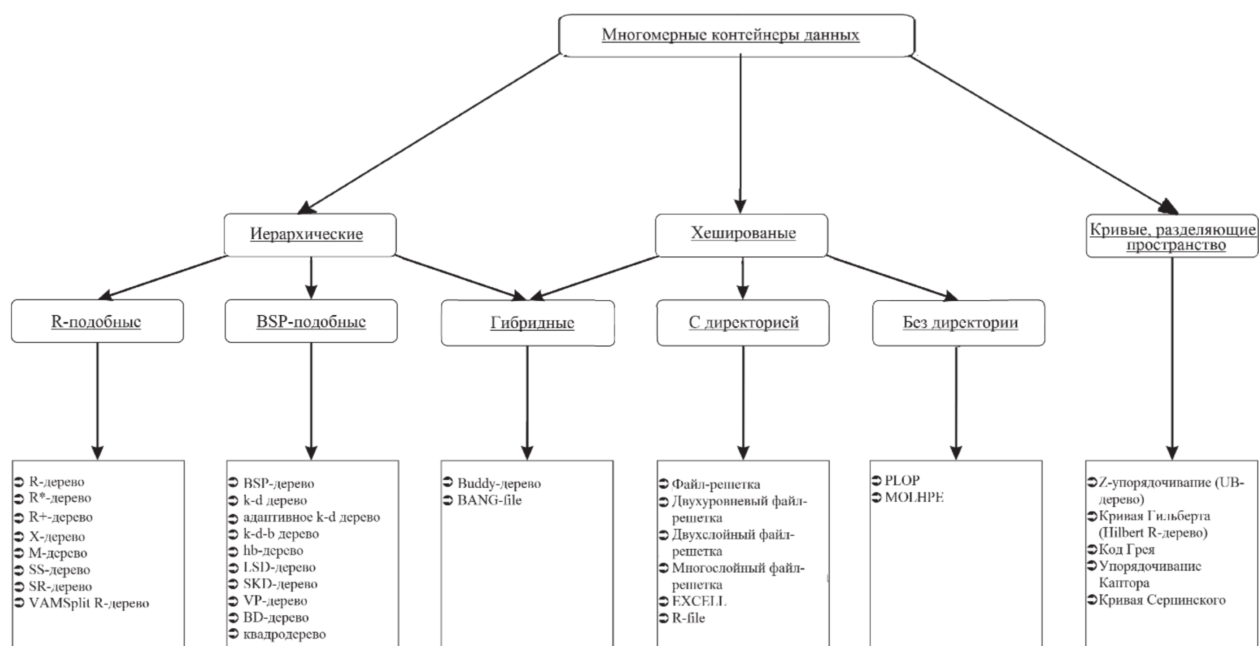

Рис. 1.3. Классификация рассмотренных многомерных контейнеров

1.2.1. Иерархическое представление многомерной информации

Иерархические контейнеры многомерной информации делятся на R-подобные деревья, которые хранят в узлах ограничивающие фигуры (паралеллотопы или сферы), и BSP-подобные деревья, которые разделяют пространство на два и более подпространства гиперплоскостями [55].

1.2.1.1. R-дерево

R-дерево [56] (R — от Region/Rectangle) — аналог B-дерева для многомерных данных. Данный контейнер делит пространство на прямоугольники (для

двумерного пространства) и параллелотопы (для многомерного пространства) таким образом, что области разбиения могут пересекаться и образуют иерархию (рис. 1.4). При этом объекты, которые расположены рядом, оказываются в одном узле дерева.

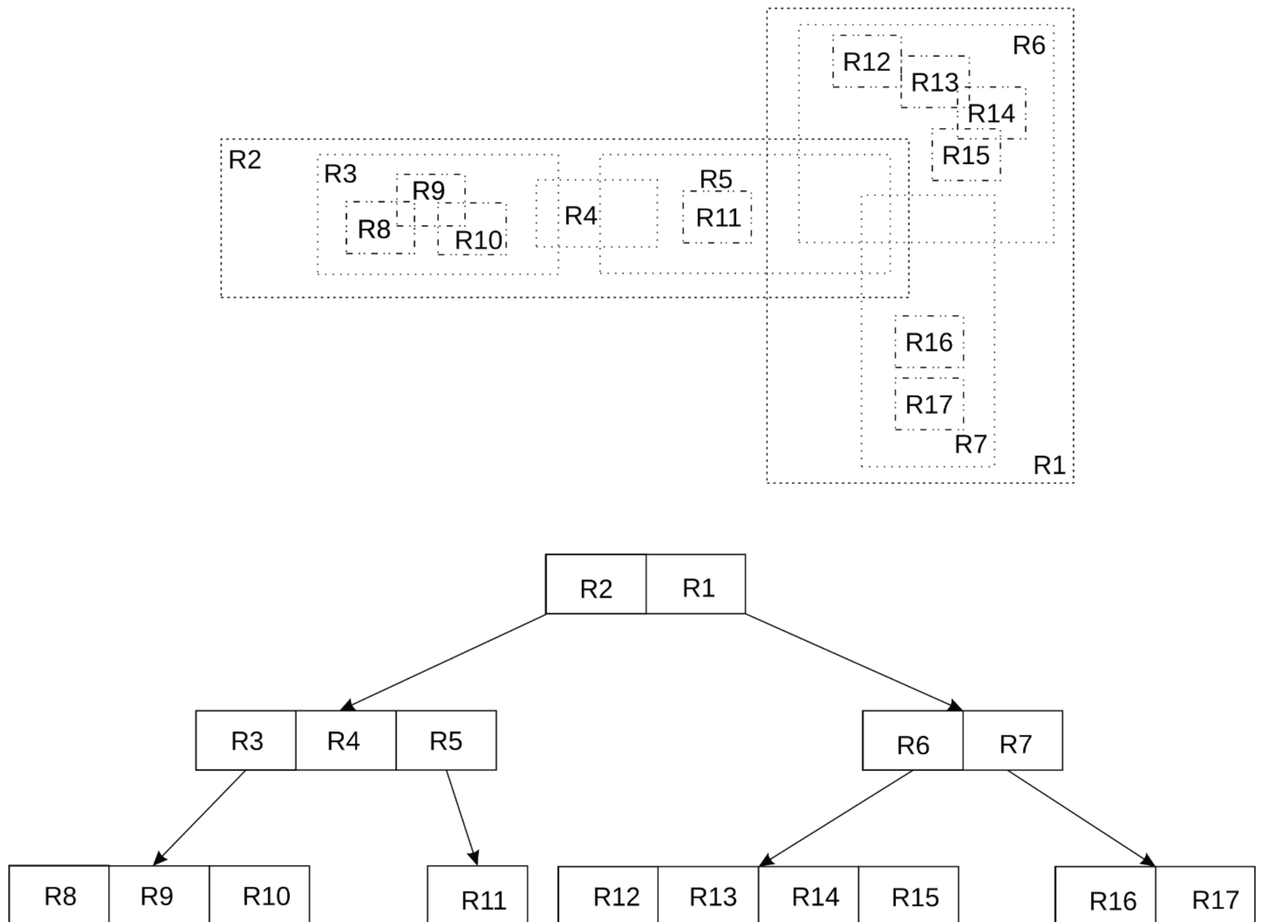

Рис. 1.4. R-дерево

Помимо этого, существуют различные модификации и вариации данного дерева, которые были созданы с целью улучшения производительности R-дерева. Ниже перечислены основные:

- R+ дерево [57];
- R* дерево [58];
- X-дерево [59];
- M-дерево [60];
- SS-дерево [61];
- SR-дерево [62];

- VAMSplit R-tree [63].

1.2.1.2. BSP-дерево

BSP-дерево (от англ. *binary space partition tree*) [64]— древовидная структура данных, где каждая внутренняя вершина содержит разбивающую прямую в двухмерном (гиперплоскость в трехмерном) пространстве, а листья содержат объекты (рис. 1.5). При этом если объект содержится в положительном полупространстве относительно разбивающей плоскости, то он является правым сыном, иначе левым. Таким образом, данное дерево отвечает некоторому двоичному разбиению пространства.

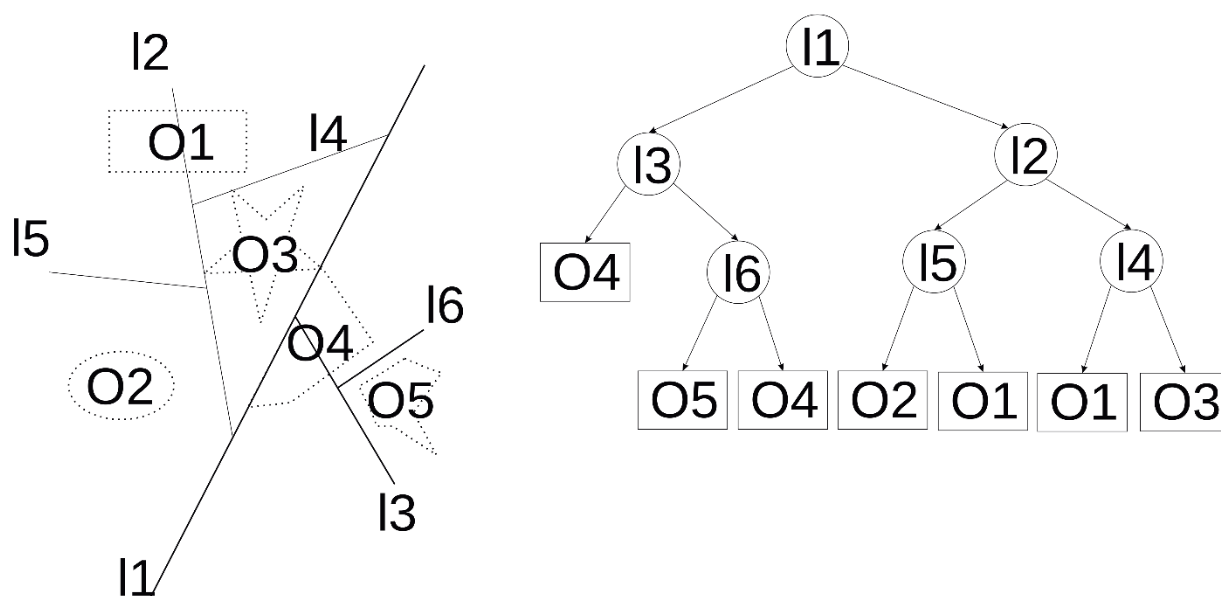

Рис. 1.5. BSP-дерево

Помимо этого, существуют модификации данного дерева, которые определяют то, как заданы разбивающие плоскости. Ниже перечислены основные из них:

- k-d-дерево (англ. *k-d tree*, сокращение от k-мерное дерево) [65] и его модификация адаптивное k-d дерево [66];
- K-D-B-дерево [67];

- hb-дерево [68];
- LSD-дерево [69];
- BIH/SKD-дерево [70];
- дерево квадрантов (англ. *quadtree*) [71];
- октодерево (англ. *octree*) [72];
- VP-дерево [73];
- BD-дерево [74].

1.2.2. Многомерное хеширование

Помимо иерархического подхода к построению многомерных контейнеров данных, применяется и хеширование. Существует несколько структур данных, использующих такой подход:

1. Файл-решетка [75] и его модификации: двухуровневый файл-решетка [75], двойная решетка [76], многослойный файл-решетка [77].
2. EXCELL [75].
3. Многомерное линейное хеширование с сохранением порядка и частичным расширением MOLHPE [78].
4. Многомерное кусочно-линейное хеширование с сохранением порядка PLOP [79].
5. Buddy-дерево [80].
6. BANG-file [81].
7. R-file [82].

1.2.3. Кривые, заполняющие пространство

Еще одним методом построения многомерных контейнеров данных являются кривые, заполняющие пространство. Основная идея состоит в упорядочивании всех точек многомерного пространства. Таким образом многомерные точки трансформируются в одномерные, для которых уже могут применяться одномерные контейнеры. Существует несколько кривых, которые могут

упорядочить многомерные данные. Самые известные изображены на рис. 1.6. На рис. 1.6*а* и *б* изображены очевидные варианты «строка за строкой» и «змейка». На рис. 1.6*в* изображено Z-упорядочивание (ключ Мортона, N-упорядочивание) [83]. На рис. 1.6*г* — кривая Гильберта, рис. 1.6*е* — код Грея [84], рис. 1.6*ж* — упорядочивание Кантора, рис. 1.6*з* — кривая Серпинского. Зачастую для получившейся одномерной информации применяют B-дерево, некоторые контейнеры имеют специальные названия: Hilbert R-дерево [85] использует кривую Гильберта, UB-дерево [86] использует Z-упорядочивание. Алгоритм Geohash [87] использует Z-упорядочивание для преобразования географических координат в строку [55].

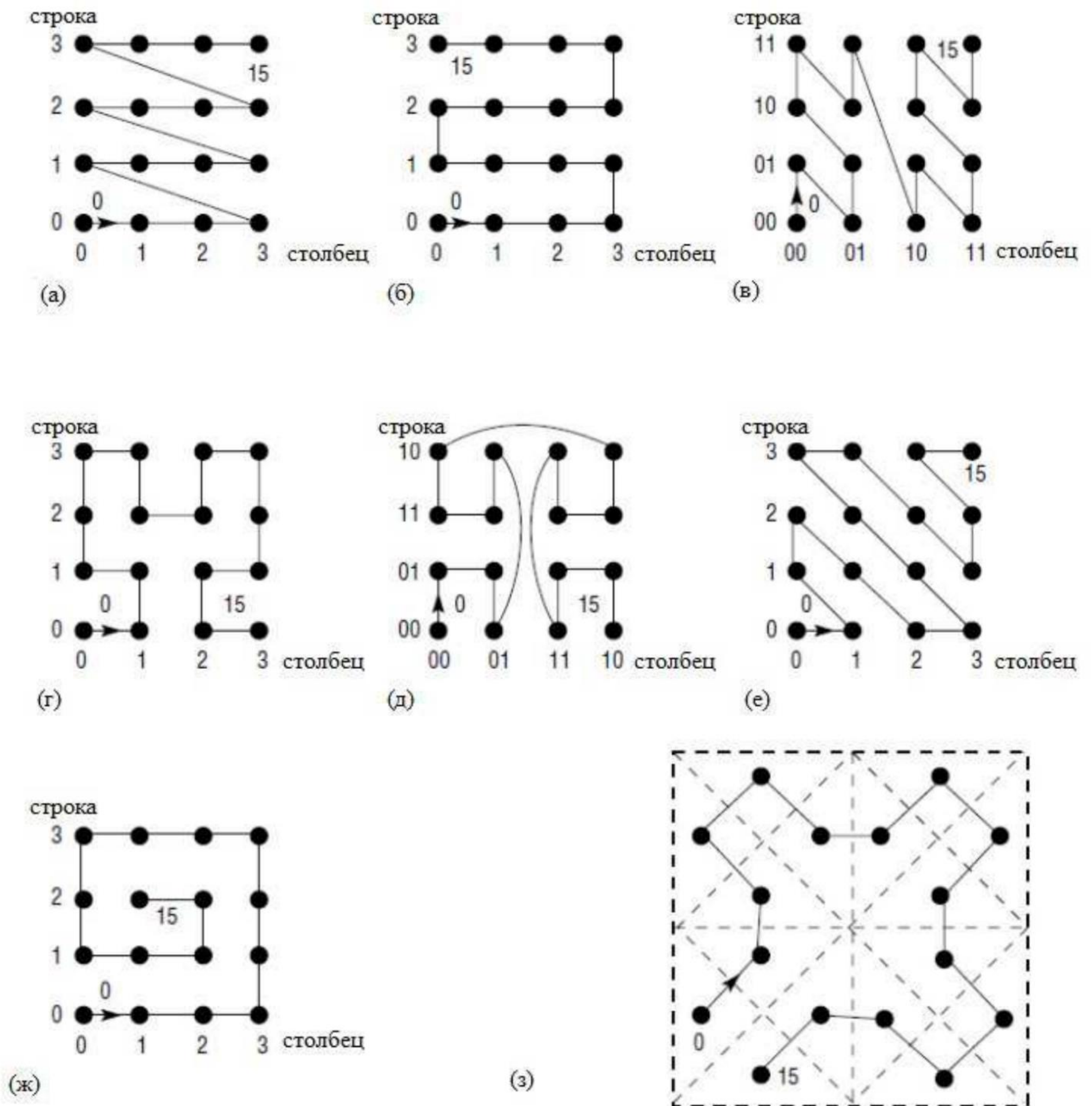

Рис. 1.6. Кривые, заполняющие пространство

1.2.4. Многомерные методы доступа для больших размерностей

Согласно проведенным исследованиям [52] все перечисленные выше методы доступа хорошо работают при размерности меньше 15. Для работы с объектами в пространствах больших размерностей были изобретены новые методы, использующие идею вектора приближения. Такие методы разделяют пространство на прямоугольные ячейки, присваивают каждой ячейке уникальную битовую строку и аппроксимируют вектор данных, который попадает в ячейку по этой

битовой строке. При поиске ближайших соседей последовательно просматриваются ячейки, битовые строки которых приблизительно равны битовой строке искомого значения. К таким контейнерам относятся VA-File, VA+-File, LPC-File, A-Tree, GC-Tree, RA-Blocks, IQ-tree, SA-tree, A-tree, iDistance и т.д. [88] [55].

1.3. Анализ применения существующих ассоциативных структур данных в самоадаптирующемся контейнере данных

Среди множества древовидных структур для использования в самоадаптирующихся ассоциативных контейнерах можно найти оптимальную структуру практически для любого случая [55]. При выборе следует руководствоваться условиями задачи (подробный анализ представлен в разделе 1.1). Основным условием для выбора структуры является объем данных. Можно выделить несколько основных случаев использования контейнеров одномерных данных:

1. В случае, когда данные помещаются в память, можно использовать сбалансированные бинарные деревья, выбор которых необходимо осуществлять на основе многих факторов: сложность построения, дополнительная память, соотношение производительности вставки и удаления.
2. Если имеется необходимость использовать контейнер как очередь с приоритетами или для решения задачи поиска минимума/максимума, то можно использовать кучу (любой вариант в зависимости от потребностей в эффективности и персистентности).
3. В случае, когда данные не помещаются в оперативную память и необходимо использовать внешнюю память, надо использовать другие деревья. Основной структурой данных является B-дерево (или его вариации в зависимости от необходимости последовательного доступа к ключам, дополнительной памяти и сложности реализации). При необходимости большого количества записей лучший выбор –

использование деревьев, оптимизированных на запись (LSM-деревья, фрактальные и буферизированные).

4. У хешированных контейнеров (см. раздел 1.2.4) есть большое преимущество – очень хорошая средняя скорость операций O(1), что гораздо лучше, чем у деревьев O(log n). Однако хешированные контейнеры имеют недостатки, связанные с внутренним устройством:

    a. Невозможно итерировать объекты в порядке увеличения/уменьшения ключей, а следовательно, невозможно эффективно реализовать операции поиска минимума/максимума и хранить данные в упорядоченном виде.
    b. Плохая временная сложность худшего случая. Например, в реализациях хеш-таблиц, которые используют полную перестройку таблицы при изменении размера, вставка и удаление имеют сложность до O(n). Кроме того, большое количество коллизий может привести к тому, что может потребоваться перебор большого количества элементов.

Таким образом, хешированные контейнеры стоит использовать для приложений, где изначально можно предсказать количество элементов, что позволит выделить под таблицу необходимую память и не перестраивать ее впоследствии. Кроме того, хеш-таблицы не стоит использовать в приложениях, где необходима гарантированная средняя стоимость операций. А также они плохо подходят для использования в определенных приложениях для работы со строками таких, как приложения для проверки орфографии (для них лучше подходят деревья и конечные автоматы).

Следовательно, использование хеш-таблиц в самоадаптирующихся контейнерах данных может быть целесообразно в случаях, когда данные долгое время используются для поиска и очень редко для вставки и удаления. Кроме того, необходимо учитывать, что при перестроении

требуется выбрать подходящую хеш-функцию и размер хеш-таблицы, что в общем случае – сложная задача. Фильтр Блума может использоваться в данных контейнерах в первую очередь как дополнительная модификация (для уменьшения количества запросов к самому контейнеру). HAMT может использоваться в самоадаптирующихся контейнерах как альтернатива хеш-таблицам для экономии занимаемого места и во избежание перехеширования таблицы [10].

Для работы с многомерной информацией в самоадаптирующемся контейнере данных необходимо использовать многомерные структуры данных (см. раздел 1.3). Среди большого количества таких структур данных можно выбрать оптимальную практически для любого случая. При выборе необходимо учитывать как общие условия работы контейнера, так и специфические для многомерных контейнеров условия. Можно выделить несколько основных случаев использования контейнеров многомерных данных:

1. При работе с большими размерностями необходимо использовать специальные контейнеры (см. раздел 1.3.4).
2. В случае, когда количество данных невелико и они помещаются в оперативную память, можно использовать k-d-дерево, BSP-дерево, BD-дерево, квадродерево, R-дерево и их различные модификации (см. раздел 1.3.1). Кроме того, существуют специальные (cache conscious) структуры данных, которые учитывают особенности кэшей процессора, что позволяет улучшить производительность. К таким контейнерам относятся CR-дерево [89] и его различные варианты, MR-дерево [90], DR-дерево.
3. Для ситуаций, когда данные долго используются без изменения, необходимо использовать структуры для статических данных, такие как k-d-дерево, Packed Hilbert R-дерево и VAMSplit-деревья.
4. При работе с большими объемами данных, когда оперативной памяти не хватает, необходимо использовать другие контейнеры. Согласно исследованиям [51], лучшими по производительности являются

структуры (при этом в списке не учитываются контейнеры, сравнительных анализов для которых не было опубликовано):

1) buddy (hash) tree [80];
2) cell tree with oversize shelves [91];
3) Hilbert R-tree [85];
4) KD2B-tree [92];
5) PMR-quadtree [93];
6) R+-tree [57];
7) R*-tree [58].

Однако выбор даже среди указанных структур представляется непростой задачей, т.к. на данный момент не существует структуры данных, которая превосходила бы другие во всех условиях. Это связано, с одной стороны, с тем, что количество критериев оптимальности велико, а, следовательно, контейнер, который является по одному параметру лучшим, может оказаться по другому худшим. Например, эффективность структуры по времени и объему занимаемой памяти зависит от данных, выполняемых запросов, во многом от характеристик аппаратного оборудования (например, размер страницы памяти). С другой стороны, контейнер, который хорошо работает при разбивающих гиперплоскостях перпендикулярных осям координат, может показать очень плохие результаты при произвольных осях разбиения. Также стоит учитывать, что точечные методы доступа могут быть крайне неэффективны для работы с пространственными объектами.

Таким образом в общем случае для использования внешней памяти в самоадаптирующихся контейнерах данных следует выбирать контейнер исходя в первую очередь из того, насколько структура данных надежна и проста в реализации. Такой подход к выбору структуры данных осуществляется во многих коммерческих продуктах, например, использование квадродеревьев в SICAD [51] и Smallworld GIS [51], R-деревьев в Informix [51] и Z-упорядочивание в Oracle [51]. Кроме того, на

основе статистики запросов к самоадаптирующемуся контейнеру можно проводить анализ многомерных структур данных и выбирать лучшую на данной нагрузке методом проб и ошибок [55].

1.4. Обзор существующих алгоритмов кэширования

Самоадаптирующиеся контейнеры данных могут быть реализованы путем использования различных структур данных в зависимости от нагрузки и условий работы контейнера (см. разделы 1.1, 1.2, 1.3) [7]. Однако наиболее эффективно такие контейнеры могут проявить себя в условиях большого объема данных. В таком случае часть данных хранится в более быстрой памяти, а другая часть – в более медленной памяти или на удаленном источнике. Для приложений, где данные статичны или количество запросов на получение данных намного превышает количество запросов на добавление элементов в хранилище данных, наиболее оптимальным будет использование кэша [94].

Кэш представляет собой набор записей «ключ-значение», расположенный в памяти с большей скоростью доступа, предназначенный для ускорения обращения к данным, расположенным в хранилище данных с меньшей скоростью доступа. Кэширование является эффективным благодаря свойствам локальности ссылок (принципам локальности). Выделяют принцип временной и пространственной локальности. Принцип временной локальности состоит в том, что если в какой-то момент времени конкретная ячейка памяти была запрошена, то вполне вероятно, что на ту же ячейку будут ссылаться снова в ближайшем будущем. Принцип пространственной локальности означает, что если была запрошена некоторая ячейка памяти, то в течении некоторого небольшого промежутка времени с большой вероятностью будет обращение к одной из близлежащих ячеек. Таким образом, используя данные свойства локальности, кэш позволяет хранить данные, к которым скорее всего будет произведено обращение в ближайшем будущем, что позволяет избежать дополнительных обращений к медленной памяти [95].

Кэширование применяется как на аппаратном уровне (в процессорах [96], жестких дисках [97]), так и на программном уровне [98] (в операционных системах,

на сетевом уровне (в сетях доставки контента (CDN), системах DNS, сессиях), интернет-приложениях, базах данных, мобильных приложениях, сервисах кэширования в оперативной памяти (Redis [99] [100], Memcached [101] [102])) [7]. Помимо выигрыша в производительности при чтении данных, кэш может использоваться и для ускорения записи.

Однако основной целью кэширования является оптимизация запросов на получение данных, поэтому далее в данной работе рассматривается использование кэша только для чтения. В этом случае данные попадают в кэш при промахе кэша. *Промахом кэша* называется ситуация, когда в кэше не был найден элемент с запрошенным ключом и такой элемент загружается из основного хранилища.

Ситуация, когда элемент с запрошенным ключом присутствует в кэше называется *попаданием в кэш*. Процентное соотношение попаданий в кэш к общему числу запросов называется *уровнем попаданий (hit ratio)* или *коэффициентом попаданий* в кэш.

На практике размер кэша имеет ограничения, и, следовательно, не все необходимые элементы могут быть добавлены в кэш. В таком случае при промахе кэша необходимо решить, нужно ли загружать в кэш запрошенный элемент и какой элемент кэша должен быть замещен. Для решения этой задачи существует большое количество различных *алгоритмов кэширования (политик замещения)*. Эффективность таких алгоритмов в первую очередь оценивается по коэффициенту попаданий в кэш и зависит от многих факторов (размер кэша, программно-аппаратные условия, объем основного хранилища). Можно выделить несколько основных алгоритмов:

1. *LRU* (Least Recently Used) [103]. При использовании данного алгоритма замещается элемент, к которому дольше всего не было обращений. Данный алгоритм является наиболее используемым в силу своей эффективности близкой к оптимальной. Существует несколько реализаций данного алгоритма, однако все они обладают плохой производительностью. Для того чтобы улучшить производительность

и эффективность LRU алгоритма были изобретены различные варианты данного алгоритма:

a. *TLRU* (Time aware least recently used) [104].
b. *PLRU* (Pseudo-LRU) [105] – это целое семейство алгоритмов кэширования, которые улучшают эффективность LRU алгоритма. Наиболее популярными алгоритмами данного семейства являются Tree-PLRU [97] и Bit-PLRU [106].
c. *LRU-K* [107] [108].
d. *ARC* (Adaptive Replacement Cache) [109].
e. *2Q* (две очереди) [110].
f. *SLRU* (сегментированный LRU) [105].
g. *MQ* (Multi queue) [111].
h. *BIP* (Bimodal Insertion Policy) [112].
i. *DIP* (Dynamic Insertion Policy) [112].

2. *MRU* (Most Recently Used) [113] – в отличии от LRU вытесняется последний использованный элемент. Согласно исследованиям [] такая политика замещения хорошо проявляет себя при случайных схемах обращения и повторяющихся сканированиях над большими данными.
3. *LIRS* (Low Inter-reference Recency Set) [114]. Данная политика учитывает две характеристики для каждого элемента кэша x: IRR(x) (Inter-Reference Recency) – количество элементов, к которым обращались между двумя последними обращениями к элементу x и R(x) (Recency) – количество уникальных обращений к кэшу после последнего обращения к элементу x. Весь кэш разделяется на два раздела: первый раздел (порядка 99% от размера кэша) хранит элементы с небольшим расстоянием повторного использования LIR (Low IRR) и второй раздел, который хранит элементы с большим расстоянием повторного использования HIR (High IRR). При этом HIR элемент может быть нерезидентным, то есть хранить только метаданные (IRR и R). Последние запрошенные элементы хранятся в

очереди S, это могут быть как LIR, так и HIR элементы, при этом в конце очереди всегда располагается LIR элемент. Все резидентные HIR элементы хранятся в очереди Q.

Пока LIR раздел не заполнен все запрошенные блоки получают LIR статус, после этого все элементы, запрошенные первый раз или которых нет в очереди S (были запрошены слишком давно), получают статус HIR блока и помещаются в голову очереди S. При запросе блока, находящегося в кэше возможны три ситуации:

a. Запрошен элемент со статусом LIR. В этом случае элемент помещается в голову очереди S. Если этот элемент был в конце очереди S, то все HIR элементы очереди до предыдущего LIR элемента будут удалены из очереди S.
b. Запрошен резидентный HIR элемент. Элемент помещается в голову очереди S. При этом если он уже был в очереди S, то статус меняется на LIR и удаляется из очереди Q, а LIR элемент, который был в конце списка S помещается в конец очереди Q со статусом HIR и все HIR элементы очереди до предыдущего LIR элемента будут удалены из очереди S. Если запрошенного элемента не было в очереди S, то элемент перемещается в конец очереди Q.
c. Запрошен нерезидентный HIR элемент. Это означает, что элемента нет в кэше. В этом случае удаляется резидентный HIR элемент, находящийся в голове очереди Q (он становится нерезидентным) и в голову очереди S загружается запрошенный элемент. При этом если запрошенный элемент уже был в очереди S, то его статус меняется на LIR, а LIR элемент, который был в конце списка S помещается в конец очереди Q со статусом HIR и все HIR элементы очереди до предыдущего LIR элемента будут удалены из очереди S. Если элемента не было в очереди S, то элемент добавляется в конец очереди Q.

4. LFU (Least frequently used) [115]. Данный алгоритм подсчитывает сколько раз был запрошен каждый элемент кэша и вытеснен будет элемент, к которому обращались меньше всего раз. У данного алгоритма существует несколько модификаций:
    a. LFRU (Least frequent recently used) [116].
    b. LFUDA (LFU with dynamic aging) [117].
5. *RR* (Random replacement) [113].
6. *LIFO* (Last in first out) [113].
7. *FIFO* (First in first out) [113].
8. CLOCK [118] и его модификации GCLOCK [118], Clock-Pro [119] и CAR (Clock with Adaptive Replacement) [120].
9. NFU (Not frequently used) [121].
10. NRU (Not recently used) [94].

Помимо перечисленных выше общих алгоритмов кэширования, применимых в разных условиях и системах, существуют алгоритмы, специально разработанные для определенных устройств и приложений. Это специализированные политики замещения, применяемые в процессорах (семейство PLRU алгоритмов и их модификаций, кэш прямого отображения, N-канальная ассоциативность, H-NMRU и другие) и жестких дисках (Clean first LRU (CFLRU), Flash Aware Buffer policy (FAB), DUal LOcality (DULO), Block Padding Least Recently Used (BPLRU), Large Block CLOCK (LB-CLOCK), Lazy Adaptive Replacement Cache (LARC) и другие). Также существует большое количество алгоритмов, применяемых в веб-приложениях и распределенных системах Greedy Dual Size (GDS), SIZE и их различные модификации, Weighting Size and Cost Replacement Policy (WSCRP), Least Relative Value (LRV), LNC-R-W3-U, Least Unified Value (LUV), Logistic Regression Algorithm (LR), Size-Adjusted LRU (SLRU), Hybrid и Mix политики, алгоритмы с использованием нечеткой логики и искусственного интеллекта и др. Кроме того, используется политики замещения для кэширования мультимедиа

(Chunk-based Caching algorithm (CC), Quality-based video Caching algorithm (QC)), Pareto-based Least Frequently Used algorithm (PLFU), Pareto-based Least Recently Used algorithm (PLRU), Segment-Based Proxy Caching of Multimedia Streams, xLRU, Cafe Cache, gLRU и другие.

Однако несмотря на многообразие политик кэширования, на данный момент не существует универсального алгоритма, который был бы оптимальным при любой нагрузке и в любых условиях работы контейнера. Следовательно, существует необходимость в адаптивном алгоритме кэширования, который бы реагировал на изменение нагрузки и других параметров [95].

1.5. Выводы

1. Проведен анализ условий, в которых происходит адаптация самоадаптирующихся контейнеров данных.
2. Рассмотрены существующие методы построения одномерных и многомерных ассоциативных контейнеров данных и проанализирована возможность их применения в самоадаптирующихся контейнерах данных.
3. Рассмотрены существующие алгоритмы кэширования.
4. Исходя из выполненного анализа сделан вывод, что не существует универсальной структуры данных, алгоритма кэширования и размера кэша, которые бы обеспечивали максимальную эффективность работы в любых условиях и для любых нагрузок.
5. Можно установить, таким образом, существование необходимости создания контейнера данных, который бы изменял логику своей работы в зависимости от условий работы и нагрузки. Одним из путей решения этой проблемы может быть создание контейнера данных, который выбирает оптимальную структуру данных, алгоритм кэширования и размер кэша для каждого конкретного случая на основе различных критериев [3].
6. Построение такого контейнера является очень сложной и трудоемкой задачей, поэтому данное исследование ограничивается следующими задачами:

I. Разработка общей математической модели самоадаптирующегося контейнера.
II. Анализ областей применения существующих структур данных и алгоритмов кэширования в таком контейнере (разделы 1.4, 1.5).
III. Поиск зависимости размера кэша от условий работы и нагрузки.
IV. Разработка и реализация модуля определения параметров сложной нагрузки на самоадаптирующиеся контейнеры данных.
V. Разработка и реализация адаптивного кэширующего контейнера данных с использованием интервального статистического ряда.

Глава 2. Разработка и исследование способа задания самоадаптирующегося контейнера данных

Постоянное увеличение объемов информации приводит к потребности эффективно организовывать хранение и обработку информации. На сегодняшний день существует много различных решений по хранению данных. У каждого из них есть свои плюсы и минусы. Кроме того, контейнеры (модули хранения информации) можно комбинировать различными способами. Таким образом, количество различных контейнеров, предназначенных для хранения информации и поддерживающих необходимые для работы с этими данными операции, огромно.

Для того чтобы в каждый момент времени структура данных была оптимальна для текущих условий работы и запросов, поступающих в контейнер, необходимо разработать специальную структуру данных, которая бы меняла внутреннюю структуру данных.

Однако наиболее эффективно такие контейнеры могут проявить себя в условиях большого объема данных. В таком случае часть данных хранится в более быстрой памяти, а другая часть — в более медленной памяти или на удаленном источнике. Для приложений, где данные статичны или число запросов на получение данных намного превышает число запросов на добавление элементов в хранилище данных, наиболее оптимальным будет использование кэша. В таком случае помимо смены основного контейнера, самоадаптирующийся контейнер данных может изменять такие параметры кэша как алгоритм вытеснения или размер кэша.

Для решения сформулированных в главе 1 задач необходимо построить математическую модель системы. В данной главе рассматривается построение и исследование способа задания самоадаптирующегося контейнера данных и его различных модулей, элементами которых являются множества и их наборы, а также отображения из одних множеств в другие (функции). Для этого вводится структурная модель самоадаптирующегося контейнера данных, которая описывает элементы адаптивного контейнера и взаимодействие между ними. Кроме того,

необходимо учитывать, какие ограничения могут накладывать задачи, для которых используются эти контейнеры. Для этого необходимо построить модель условий адаптации самоадаптирующихся контейнеров данных. Основными параметрами этой модели являются объем информации, количество и типы запросов, а также архитектура информационной системе.

Для построение математической модели самоадаптирующегося контейнера данные необходимо обозначить способ задания ассоциативного контейнера данных. Для этого вводится множество объектов в контейнере в определенный момент времени, множество операций (отображений) над контейнером и внутреннее состояние контейнера в определенный момент времени. Кроме того, необходим способ задания нагрузки на ассоциативный контейнер данных. Для этого вводится понятие структурной, блочной, последовательной и базовой нагрузки и известное понятие *трасса* представлено в терминах нагрузки. Способ задания самоадаптирующегося контейнера данных задается с помощью множества условий адаптации, множества состояний контейнера, состояния контейнера в текущий момент времени и отображения перехода, которое при изменении нагрузки меняет состояние контейнера.

Также важно оценить эффективность самоадаптирующегося контейнера данных для текущего набора параметров для текущей нагрузки в текущих условиях адаптации. В данной работе основным критерием эффективности является время работы контейнера, которое зависит от времени работы основного хранилища и времени работы кэша.

Для нахождения зависимости размера кэша от условий работы и нагрузки необходимо выбрать ключевые параметры нагрузки и условий работы. В качестве основного параметра условий работы выбрано соотношение скоростей основного хранилища и кэша. Использование кэша имеет смысл при большом объеме данных, следовательно, основное хранилище должно располагаться на диске или удаленном источнике и соотношение скоростей хранилищ является определяющим фактором при выборе размера кэша [7]. В качестве параметра нагрузки было выбрано среднеквадратичное отклонение нормального распределения ключей

нагрузки. Данное распределение было выбрано потому, что нагрузка подвержена влиянию большого количества случайных факторов, следовательно, подчиняется нормальному закону распределения. Такое распределение часто используется для анализа данных. В данной главе аналитически сделаны некоторые предположения, которые были проверены в разделе 4.1 с помощью алгоритма, описанного в разделе 3.1.

Во многих задачах нагрузка представляет собой набор простых нагрузок, так как запросы к хранилищу могут приходить из разных источников или для различных прикладных задач. Таким образом необходимо выявлять параметры простых нагрузок для максимально эффективного использования кэша [8]. Для этого необходима модель нагрузки из смеси нормальных распределений. На основе данной модели разработан алгоритм, описанный в разделе 3.2, результаты тестирования которого представлены в разделе 4.2.

2.1. Структурная модель самоадаптирующегося контейнера данных

В первой главе были рассмотрены различные ассоциативные контейнеры данных и методы кэширования, и сделан вывод о необходимости создания самоадаптирующегося контейнера данных, который позволит использовать наиболее эффективные контейнеры данных и алгоритмы кэширования в зависимости от входящих данных.

Самоадаптирующийся контейнер данных (*SADC*) – ассоциативный контейнер, который меняет логику своей работы в зависимости от нагрузки L (поступающих к нему запросов, см. разделы 2.3 и 2.4) [3] [7]. Кроме того, данный контейнер должен учитывать условия, в которых он работает – условия адаптации C (см. раздел 2.2).

Самоадаптирующийся контейнер данных состоит из:

1. Основного хранилища (ассоциативный контейнер *AS*);
2. Кэширующего ассоциативного контейнера *CS* размера M, реализующего алгоритм кэширования A;

3. Состояния самодаптирующегося контейнера данных в момент времени t q_t это комбинация из состояния ассоциативного контейнера данных *AS* на момент времени *t* и состояния кэширующего контейнера *CS* на момент времени *t*;
4. Модуля адаптации, который реализует некоторую функцию перехода *f* от одного состояния самоадаптирующегося контейнера к другому.

Основные элементы системы самоадаптирующегося контейнера данных изображены на рисунке 2.1.

Рис. 2.1. Структура самоадаптирующегося контейнера данных

Далее в разделе 2.5 описан способ задания самоадаптирующегося контейнера данных с помощью множеств и наборов, описывающий данную структурную модель.

2.2. Анализ условий адаптации самоадаптирующихся контейнеров данных

Для того чтобы построить самоадаптирующийся ассоциативный контейнер данных, необходимо понимать, в каких условиях происходит адаптация. Нужно знать, какие ограничения могут накладывать задачи, для которых используются эти контейнеры. Среди основных условий можно выделить следующие:

1. объем информации V;
2. количество и типы запросов Req;
3. архитектура информационной системы A [3].

Данные условия будут рассмотрены далее более подробно. Таким образом условия адаптации C могут быть записаны в виде:

$$C = \{V, Req, A\} \quad (2.2.1)$$

2.2.1. Объем информации

Объем информации V является основным условием адаптации контейнера, так как это условие определяет, где располагается контейнер. Существует несколько вариантов его расположения (в порядке увеличения объёма информации):

1. данные располагаются в памяти *InMemory*. В первую очередь необходимо понимать, что контейнер использует несколько уровней памяти компьютера [122]. Верхний уровень иерархии памяти – это кэш процессора. Здесь можно использовать особенности кэша. Следующий уровень иерархии памяти – это оперативная память [94]. В данном случае контейнер помещается в размеры оперативной памяти и, следовательно, не требуется дополнительных операций для хранения информации. При перестроении контейнера в оперативной памяти не требуется учитывать

такие дополнительные факторы, как, например, затраты на доступ к диску или передача данных по сети (которые являются гораздо более существенными, чем затраты на многопоточность и время ожидания блокировок). Кроме того, не нужны специальные алгоритмы для уменьшения затрат на операции ввода-вывода. Таким образом, для контейнера можно использовать любые подходящие для данной нагрузки структуры данных.

2. данные располагаются частично в памяти, частично на диске *InMemoryOrDisk*. На практике количество данных может значительно превышать объем оперативной памяти, и поэтому необходимо использовать дополнительную память [20]. В этом случае наиболее часто используются следующие накопители: HDD и SSD — накопитель на базе технологии флэш-памяти [21]. Для HDD диска время доступа к оперативной памяти в 10^4 раз меньше, чем к жесткому диску, а для последних моделей SSD — в десятки раз меньше. Тем не менее, для обоих типов накопителей любая операция чтения или записи замедляет контейнер в тысячи (для SSD – в десятки) раз, и необходимо использовать другие алгоритмы и структуры данных. Основная идея этих алгоритмов заключается в кэшировании данных.
3. данные располагаются на кластере *InCluster* [3]. В случае, когда количество данных огромно (десятки и сотни петабайт) и дискового пространства не хватает или необходимо параллельно обрабатывать [24] эти данные, используется компьютерный кластер [25] [26]. Для выбора оптимального контейнера в случае кластера необходимо учитывать, как организовано хранение данных на нескольких связанных компьютерах [27] [28]. Алгоритм выбора оптимального контейнера должен учитывать, что любое обращение к данным невероятно дорого с точки зрения затрат системы и производительности. Кроме того, оптимальный контейнер предполагает перестроение контейнера данных, что влечет за собой еще большие нагрузки на сеть и файловую систему.

Таким образом объем информации V может быть записан в виде:

$$V \in \{InMemory, InMemoryOrDisk, InCluster\} \quad (2.2.2)$$

2.2.2. Количество запросов

Это второй по важности параметр после объема информации. Возможна ситуация, когда количество данных огромно, а количество запросов к ним невелико. С другой стороны, возможно небольшое количество часто запрашиваемых или часто изменяющихся данных. Для каждого из таких случаев необходимо выбрать свой оптимальный контейнер.

Помимо этого, важно понимать количество и типы запросов. Существует три базовых запроса:

1. Поиск (Select). Операция чтения данных из контейнера.

2. Вставка (Insert). Операция вставки данных.

3. Удаление (Remove). Операция удаления данных.

При этом каждый контейнер данных является оптимальным и работает эффективнее других для определенного соотношения запросов. Например, простой список хорошо подходит для большого количества вставок, но плохо для операций поиска и удаления.

В свою очередь, B-дерево не стоит использовать, если количество операций вставки примерно равному количеству операций поиска при малом количестве удалений. В остальных случаях этот контейнер хорошо подходит для различных соотношений операций вставки, удаления и поиска.

На основании этих соотношений можно понять, в какой момент необходимо перестраивать контейнер, т. е. адаптировать его под новое соотношение запросов вставки, удаления и поиска. Следовательно, необходимо понять, какое соотношение базовых операций является пограничным, т. е. после преодоления которого необходимо менять контейнер данных.

2.2.3. Архитектура информационной системы

Еще одним важным фактором при адаптации является архитектура информационной системы (ИС) [123]. Существует несколько типов ИС:

1. *Desktop.* Настольные (desktop), или локальные ИС, в которых все компоненты находятся на одном компьютере, а, следовательно, контейнер данных располагается на этом же компьютере. В этом случае нет никаких дополнительных затрат памяти и процессорного времени на работу оптимального контейнера.
2. *FileServer.* Файл-серверные ИС [123]. Данная архитектура показывает себя хорошо в современных условиях при небольших объемах данных, но при увеличении числа компьютеров в сети или росте БД [124] происходит резкое падение производительности. Причина заключается в резком увеличении объема данных передаваемых по сети, так как все данные обрабатываются на компьютере пользователя. Например, когда пользователю требуется несколько записей из таблицы, несколько тысяч строк, файл-сервер передаст всю таблицу на компьютере пользователя и только потом СУБД [125] отберет необходимые записи.

   Для адаптации оптимального контейнера – это худший вариант ИС, поскольку при запросе одного элемента из контейнера будут извлекаться все элементы, содержащиеся в контейнере. Кроме того, при любом запросе передается большое количество дополнительной информации, что при большом количестве запросов к контейнеру ведет к избыточным затратам процессорного времени и памяти. При этом стоит помнить, что это сервер и клиент – это различные машины и, следовательно, данные передаются по сети, что существенно ограничивает скорость обмена информацией.

3. *ClientServer.* Клиент-серверные ИС [123]. Сервер обрабатывает запросы клиента и извлекает необходимые клиенту данные из базы данных и возвращает приложению, расположенному на компьютере клиента.

Основное преимущество такой архитектуры заключается в том, что количество передаваемых данных значительно меньше, чем в других архитектурах. Для адаптации контейнера эта архитектура ИС подходит лучше, чем файл-серверная, потому что клиенту передается только та информация, которая им запрашивалась. Но также, как и в файл-серверной архитектуре скорость обмена информацией между клиентом и сервером ограничена скоростью передачи по сети [3].

$$A \in \{Desktop, FileServer, ClientServer\} \quad (2.2.3)$$

2.3. Способ задания ассоциативного контейнера данных

Ассоциативный контейнер данных может быть представлен в виде набора:

$$AS = \langle O_t, OP, s_t \rangle, \quad (2.3.1)$$

где O_t – множество объектов в контейнере в момент времени t, OP – множество операций (отображений) над контейнером, s_t – внутреннее состояние контейнера в момент времени t.

$$O_t \subseteq O, \quad (2.3.2)$$

где O – множество всех возможных объектов, которые могут храниться в контейнере.

$$O_t = \{o^i\},\ i = 1, \dots, n_t, \quad (2.3.3)$$

где o^i – пара «ключ-значение», $n_t \in N$ – количество объектов в контейнере в момент времени t.

$$o^i = \langle k_i, v_i \rangle, \qquad i = 1, \dots, n_t, \quad (2.3.4)$$

где $k_i \in N$ – некоторый идентификатор (также называемый ключом), который однозначно определяет объект o^i, $v_i \in O$ – данные объекта o^i (также называемые значением).

$$s_t \in S, \quad (2.3.5)$$

где S – множество всех внутренних состояний контейнера.

$$OP = \{select, insert, remove\}, \tag{2.3.6}$$

где:

1. $select : N \times S \rightarrow O \times S$ это отображение, которое при получении идентификатора объекта возвращает объект, находящийся в множестве O и изменяет состояние контейнера s_t:

$$select(x, s_t) = \begin{cases} (o^x, s_{t+1}), \ (k_x = x) \wedge x \in К_t \text{ (объект найден в контейнере)} \\ (\emptyset, s_{t+1}), \ x \notin К_t \text{ (объект не найден в контейнере)} \end{cases}, \tag{2.3.7}$$

   где $К_t$ – множество всех ключей объектов из множества O_t, $x \in N$, $К_t \subset N$.

2. $insert: O \times S \times O \rightarrow O \times S$ это отображение, которое добавляет объект к множеству объектов в контейнере:

$$insert(O_t, s_t, x) = (O_t \cup x, s_{t+1}) \tag{2.3.8}$$

3. $remove: O \times S \times O \rightarrow O \times S$ это отображение, которое удаляет объект из множества объектов в контейнере:

$$remove(O_t, s_t, x) = (O_t \setminus x, s_{t+1}) \tag{2.3.9}$$

При этом стоит отметить, что для различных ассоциативных контейнеров внутреннее состояние s_t, и набор операций OP реализовано различными способами.

Пусть SAS – множество всех ассоциативных контейнеров, тогда

$$(OP^i, {s_t}^i)! = (OP^j, {s_t}^j), i = 1 \dots n_{sas}, j = 1 \dots n_{sas}, i! = j \tag{2.3.10}$$

где $n_{sas} = |SAS|$

2.4. Способ задания нагрузки на ассоциативный контейнер данных

Согласно [1] нагрузка представляет собой различные последовательности операций вставки, выборки и удаления в контейнер данных. На рисунке 2.2 представлена классификация нагрузок.

Рис. 2.2. Классификация нагрузок

Нагрузкой L на контейнер данных [7] называется:

$$L \in STL, \tag{2.4.1}$$

где STL – множество структурных нагрузок, определяемое по формуле:

$$STL = \{LB, LS\}, \tag{2.4.2}$$

где LB – блочная нагрузка (элементы в случайном порядке), а LS – последовательная нагрузка (кортеж элементов).

$$LB = \{e_1, \dots, e_n\} \tag{2.4.3}$$

$$LS = (e_1, \dots, e_n)$$

где $n \in N$ – количество элементов в нагрузке, а элементы нагрузки e_i это либо структурные нагрузки, либо базовые и задаются формулой:

$$e_i \in \{BL, STL\}, i = 1, \dots, n, \tag{2.4.4}$$

где BL – базовая нагрузка:

$$BL = (c, type), \tag{2.4.5}$$

где $c \in N$ – количество элементов в нагрузке,

$type \in \{IL, RL, SL\}$ – тип нагрузки (IL – вставка в контейнер, RL – удаление из контейнера, SL – получение элемента контейнера).

Рассмотрим также понятие трассы (англ. *trace*) [126] [97] [106] [109], которое будет использовано далее. Трассой называется детерминированная

последовательность ключей, представляющая собой последовательность обращений с этими ключами к кэшу. В терминах нагрузки трасса T представляет собой:

$$T = L = LS =< BL(c, SL) >, \tag{2.4.6}$$

где c – количество элементов в трассе, а сами элементы трассы определяются отображением:

$$g(c) = \{o_1, \ldots, o_c\}. \tag{2.4.7}$$

2.5. Моделирование процесса применения нагрузки на контейнер данных

Процесс применения нагрузки на контейнер данных может быть представлен в виде набора:

$$AL = (AS, L, g, a), \tag{2.5.1}$$

где AS – контейнер данных, L – нагрузка, g – Отображение для генерации пар «ключ-значение», a – отображение для применения нагрузки L к контейнеру AS.

$$g: N \times R \times R \to O$$
$$g(c) = \{o_1, \ldots, o_c\}, o_i \in O \tag{2.5.2}$$

$$a: O \times S \times STL \to O \times S$$
$$a(O_t, s_t, L) = \tag{2.5.3}$$
$$\begin{cases} a(\ldots a(a(O_t, s_t, e_1), e_2) \ldots, e_n) | e_i \in L, i = 1..n, L \in STL \\ insert(\ldots insert(insert(O_t, s_t, x_1), x_2), \ldots, x_c) \\ \qquad | x_i \in g(c, min, max), L \in BL \wedge (type = IL) \\ remove(\ldots remove(remove(O_t, s_t, x_1), x_2), \ldots, x_c) \\ \qquad | x_i \in g(c, min, max), L \in BL \wedge (type = RL) \\ (O_t, select(O_t, \ldots select(O_t, select(O_t, s_t, x_1.key), \ldots, x_c.key) \\ \qquad | x_i \in g(c, min, max), L \in BL \wedge (type = SL) \end{cases},$$

где O_t – множество объектов ассоциативного контейнера AS, s_t – внутреннее состояние контейнера AS, $\{insert, remove, select\}$ – множество операций контейнера AS.

2.6. Способ задания самоадаптирующегося контейнера данных

На основе описанных выше наборов и множеств был разработан способ задания самоадаптирующегося контейнера:

$$SADC = \langle C, Q, q_t, f \rangle, \quad (2.6.1)$$

где C – множество условий адаптации, Q – множество состояний контейнера, $q_t \in Q$ – состояние контейнера в момент времени t, f – отображение перехода, которое при изменении нагрузки меняет состояние контейнера.

$$Q = \{q_i\}, i = 1..n \quad (2.6.2)$$

$$q_t = \langle AS_t, CS_t \rangle,$$

где $AS_t \in SAS$ – ассоциативный контейнер данных, хранящий данные в момент времени t, CS_t – кэширующий контейнер в момент времени t, n – количество всех возможных состояний контейнера.

$$CS_t = \langle A_t, M_t \rangle, \quad (2.6.3)$$

где $A_t \in A$ – алгоритм кэширования в момент времени t, $M_t \in N$ – размер кэша в момент времени t.

$$f: Q \times STL \times C \to Q \quad (2.6.4)$$

$$f(q_t, L, c) = q_{t+1},$$

Таким образом отображение f определяет, как будет изменяться самоадаптирующийся контейнер в зависимости от нагрузки L в условиях адаптации C. В общем случае может изменяться основной ассоциативный контейнер AS, алгоритм кэширования A и размер кэша M.

2.7. Критерий выбора оптимального контейнера

Для того чтобы реализовать отображение f необходимо определить, как оценивать эффективность самоадаптирующегося контейнера данных, т.е. по какому критерию определять, что текущий набор параметров (AS, A, M) для нагрузки L в условиях адаптации C требует адаптации (где AS - основной ассоциативный контейнер, A - алгоритм кэширования и M - размер кэша).

Эффективность контейнера данных может быть оценена различными параметрами, однако в данной работе основным критерием эффективности является время работы контейнера. Таким образом контейнер является оптимальным, если текущий набор параметров (AS_t, A_t, M_t) доставляет минимум функции времени работы контейнера:

$$t_{SADC}(AS, A, M, L, C) \rightarrow min \quad (2.7.1)$$

Следовательно, для построения оптимального контейнера необходимо решить задачу оптимизации (2.7.1), где L, C – независимые переменные, AS, A, M – зависимые (искомые) переменные, а функция t_{SADC} задана следующим образом:

$$t_{SADC}(AS, A, M, L, C) = t_{AS}(AS, A, M, L, C) + t_{CS}(A, M, L, C), \quad (2.7.2)$$

где $t_{AS}(AS, A, M, L, C)$ – время работы основного хранилища с применением ассоциативного контейнера AS при кэширующем контейнере с алгоритмом кэширования A и размером кэша M для нагрузки L в условиях C, $t_{CS}(A, M, L, C)$ – время работы кэша с алгоритмом кэширования A и размером кэша M для нагрузки L в условиях C.

Следовательно, если время работы при наборе текущих параметров (AS_t, A_t, M_t) для нагрузки L в условиях адаптации C не достигает минимума, то необходима адаптация.

2.8. Анализ оптимального размера кэша по временному критерию для нагрузки, сгенерированной по нормальному распределению

Одним из параметров адаптации является размер кэширующего контейнера *M*. Как указано в формуле (2.7.4) размер кэша зависит от условий адаптации *C* и нагрузки *L*. Исследование эффективности работы кэширующего контейнера проводится для нагрузки, состоящей из операций чтения (трассы *T*). Основным параметром нагрузки, влияющим на размер кэша, является распределение ключей, а среди условий адаптации главным является соотношение скоростей между основным хранилищем и кэшем k.

$$k = \frac{v_{CS}}{v_{AS}}, k \in K \subset R, k > 1 \qquad (2.8.1)$$

Данное соотношение будет больше единицы, т.к. скорость самых современных SSD дисков в разы меньше скорости оперативной памяти, а для HDD дисков и удаленных источников достигает сотен тысяч раз и более (в зависимости от конфигурации компьютера и скорости доступа к удаленному источнику) [7].

Для генерации пар «ключ-значение» в трассе с помощью отображения *g* может быть использовано нормальное распределение. В силу того, что данное распределение часто встречается в природе и технике, а также в силу центральной предельной теоремы, данное распределение является одним из самых распространённых, поэтому зачастую используется для анализа данных. Т. о. трасса может быть сгенерирована как:

$$g(c) = \{o_1, \dots, o_c\}, \qquad (2.8.2)$$

где $o_i = < N(\mu, \sigma^2), v_i >, i = 1..c$.

Следовательно, необходимо найти зависимость размера кэша *M* от среднеквадратичного отклонения σ и соотношения скоростей [7] между хранилищами k, для которой выполняется критерий оптимальности 2.7.1. Зафиксировав алгоритм кэширования A_{const} и основное хранилище AS_{const}, а также взяв в качестве основного параметра трассы *T* среднеквадратичное отклонение σ и в качестве основного параметра условий адаптации *C* соотношение скоростей между хранилищами k, из формулы 2.7.1 получаем:

$$t(AS_{const}, A_{const}, M, \sigma, k) \to min, \quad (2.8.3)$$

$$\text{где } t(AS_{const}, A_{const}, M, \sigma, k) =$$

$$t_{AS}(AS_{const}, A_{const}, M, \sigma, k) + t_{CS}(A_{const}, M, \sigma, k),$$

$0 \leq M \leq M_{max}$, где M_{max} – доступный размер кэша.

Аналитически можно сделать несколько предположений:

Лемма 1. Существуют достаточно небольшие соотношения скоростей хранилищ, такие что использование кэша не имеет смысла (затраты на поиск в кэше больше, чем выгода от его использования) [7]:

$$\exists K_0: \forall M > 0, \forall \sigma > 0, \forall k_0 \in K_0, \quad (2.8.4)$$

$$t(AS_{const}, A_{const}, M, \sigma, k_0) > t(AS_{const}, A_{const}, 0, \sigma, k_0) \Longrightarrow M_{opt} = 0.$$

Лемма 2. Для соотношений скоростей не попадающих под условия леммы 1 существует некоторый размер кэша M_0 такой что накладные расходы на кэш при $0 < M < M_0$ перекрывают выгоду от его использования, далее для размера M_0 эффективность применения кэша равна эффективности без кэша, и только при $M > M_0$ применение кэша становится эффективно.

$$\forall k \notin K_0, \forall \sigma \quad (2.8.5)$$

$$\exists M_0: t(AS_{const}, A_{const}, M_0, \sigma, k_0) = t(AS_{const}, A_{const}, 0, \sigma, k_0)$$

Следствие 1. $M_{max} < M_0 \Rightarrow M_{opt} = 0$.

Лемма 3. Существуют k достаточно большие, такие что использование кэша уменьшает время работы контейнера для любого распределения ключей если позволяет максимально доступный размер кэша. В этом случае, оптимальный размер кэша будет стремиться к тому, чтобы вместить все уникальные элементы трассы T, что позволит уменьшить время обращения к основному хранилищу [7].

Однако в случае нормального распределения согласно математической статистике, в интервал $(-4\sigma, 4\sigma)$ помещается 99,997% [7] всех уникальных ключей, а следовательно оптимальный размер кэша 8σ будет достаточным и дальнейшее увеличение размера кэша не будет давать существенного уменьшения времени работы контейнера.

$$\exists K_b: \forall M < 8\sigma < M_{max}, \forall \sigma > 0, \forall k_b \in K_b, \quad (2.8.6)$$

$$t(AS_{const}, A_{const}, M, \sigma, k_b) > t(AS_{const}, A_{const}, 8\sigma, \sigma, k_b) \Rightarrow M_{opt} = N_T \approx 8\sigma,$$

где N_T – количество уникальных элементов в трассе T.

Лемма 4. $\forall k \in K_b: 8\sigma > M_{max} > M_0 \Rightarrow M_{opt} = M_{max}$. (2.8.7)

Лемма 5. Существует некоторое множество соотношений скоростей хранилищ, для которых не выполняются условия лемм 1 и 3. В таком случае существует некоторая σ_0, для которой при $\sigma < \sigma_0$, оптимальным будет не использовать кэш, а при $\sigma > \sigma_0$ оптимальный размер кэша вычисляется аналогично леммам 3 и 4.

$$\forall k \in K_d = K/(K_0 \cup K_b): M_{opt} = \begin{cases} 0, (\sigma < \sigma_0) \vee (M_{max} < M_0 \wedge \sigma > \sigma_0) \\ N_T \approx 8\sigma, 8\sigma < M_{max} \wedge \sigma > \sigma_0 \\ M_{max}, 8\sigma > M_{max} \wedge \sigma > \sigma_0 \end{cases} \quad (2.8.8)$$

Лемма 6. Основываясь на предыдущих леммах и следствии, можно вывести формулу оптимального размера кэша в зависимости от среднеквадратичного отклонения ключей в трассе и соотношения скоростей хранилищ [7]:

$$M_{opt}(k, \sigma) = \begin{cases} 0, (k \in K_0) \vee (M_{max} < M_0) \vee (\sigma < \sigma_0 \wedge k \in K_d) \\ 8\sigma, (k \in K_b \wedge M_0 < 8\sigma < M_{max}) \vee \\ (k \in K_d \wedge M_0 < 8\sigma < M_{max} \wedge \sigma > \sigma_0) \\ M_{max}, (k \in K_b \wedge 8\sigma > M_{max} > M_0) \vee \\ (k \in K_d \wedge 8\sigma > M_{max} > M_0 \wedge \sigma > \sigma_0) \end{cases} \quad (2.8.9)$$

Таким образом необходимо проверить данные утверждения вычислительным экспериментом и найти множества K_d, K_b, K_0, размер кэша M_0 и значение среднеквадратичного отклонения σ_0, для которых выполняются утверждения, что позволит получить зависимость размера кэша от среднеквадратичного отклонения ключей в трассе и соотношения скоростей кэша и основного хранилища [7] и разработать программную реализацию кэширующего самоадаптирующегося контейнера с динамическим размером кэша.

2.9. Модель нагрузки из смеси нормальных распределений

Однако на практике во многих задачах нагрузка представляет собой набор простых нагрузок, так как запросы к хранилищу могут приходить из разных источников или для различных прикладных задач. Таким образом необходимо

выявлять параметры простых нагрузок для максимально эффективного использования кэша.

Модель данных, в которой учитывается несколько одномерных нормальных распределений с различными весами называется смесью нормальных распределений (Gaussian mixture model - GMM) [127] [128]. Для заданной выборки χ объемом N смесью p(x), состоящей из K компонент $N(\mu,\ \sigma^2)$ с весами π_k называется формула:

$$p(x) = \sum_{k=1}^{K} \pi_k N(\mu_k, \sigma_k^2), \tag{2.9.1}$$

где $\sum_{k=1}^{K} \pi_k = 1$.

Т. о.

$$g(c) = \{o_1, \dots, o_c\}, \tag{2.9.2}$$

где $o_i = < p(x), v_i >, i = 1..c$.

Задача нахождения параметров таких распределений (среднеквадратичное отклонение σ и матожидание μ) сводится к задаче нахождения вектора параметров $\theta = (\pi_1, \dots, \pi_k, \mu_1, \dots, \mu_k, \sigma_1, \dots, \sigma_k)$, при котором функция правдоподобия (2.9.3) достигает максимума [8].

$$\ln p(\chi|\pi, \mu,) = \sum_{n=1}^{N} \ln\left\{\sum_{k=1}^{K} \pi_k N(x_n|\mu_k, \sigma_k^2)\right\} \tag{2.9.3}$$

2.10. Способ задания адаптивного кэширующего контейнера данных с использованием интервального статистического ряда

Пусть $SADC$ – самоадаптирующийся контейнер данных, O_t - множество объектов в контейнере AS_t в момент времени t (основное хранилище), K_t множество всех ключей множества O_t.

$$\min = \min(K_t) \tag{2.10.1}$$

$$\max = \max(K_t)$$

Тогда множество O_t может быть разбито на подмножества O_t^i:

$$O_t = \bigcup_{i=1}^{n_b} O_t^i, \qquad (2.10.2)$$

где K_t^i – множества ключей множеств O_t^i.

$$K_t^i = \{k_t^i\}, \qquad (2.10.3)$$

таких что

$$min^i < k_t^i < max^i, i = 1..n_b, \qquad (2.10.4)$$

$$min^i = min + \frac{(max - \min)}{n_b} \times (i - 1),$$

$$max^i = min + \frac{(max - \min)}{n_b} \times i.$$

Пусть T – это трасса, которая применяется на самоадаптирующийся контейнер данных, тогда её можно рассматривать как совокупность участков длиной $l \in N$ трассы:

$$\mathrm{T} = (T_1, T_2, \dots T_n), \qquad (2.10.5)$$

где $T_1 = (r_1, .. r_l), T_2 = (r_{l+1}, .. r_{2l}), \dots T_n = \left(r_{l*(n-1)+1}, \dots, r_{|T|}\right), n = |T|/l.$

Тогда для каждого участка трассы можно построить интервальный статистический ряд:

$$h(T_j) = \left(hc_1, \dots, hc_{n_b}\right), \qquad (2.10.6)$$

где $hc_i = (min^i, max^i, c_i),\ i = 1..n_b$,

где min^i, max^i – определены по формулам (2.10.4),

$c_i = |\{r : min^i < r < max^i, r \in T_j\}|$ - количество ключей участка трассы T_j, которые попадают в интервал $[min^i; max^i]$.

Алгоритмом кэширования с использованием интервального статистического ряда называется:

$$AH(H, h_0, g, l, n_b), \qquad (2.10.7)$$

где $H = \{h: h = (hc_1, \dots, hc_{n_b}),\ c_1 > c_2 > \dots > c_{n_b}\}$ – множество всех возможных ранжированных по убыванию (по количеству ключей участка трассы c_i) интервальных статистических рядов, $h_0 = (0{,}0 \dots 0) \in H$ – начальное состояние ранжированного интервального статистического ряда, l – параметр кэша, определяющий длину участка трассы, который инициирует процесс загрузки нового состояния кэша, n_b – количество блоков на которые разбивается основное хранилище, g – отображение перехода, которое при получении порядкового номера ключа в текущем участке трассы по текущему ранжированному интервальному статистическому ряду и состоянию кэш-памяти возвращает новый ранжированный интервальный статистический ряд и новое состояние кэш-памяти:

$$g: S_M \times H \times N \to S_M \times H, \tag{2.10.8}$$

где $S_M = \{S | S \subseteq O \wedge |S| \leq M\}$ – множество всех подмножеств множества объектов основного хранилища O, которые могут быть размещены в кэше объёма M.

Пусть $S_t \in S_M$ – состояние кэша в момент времени t, $h_t \in H$ – состояние интервального статистического ряда в момент времени t, $i \in N$ – порядковый номер запрашиваемого ключа в текущем участке трассы T_j, тогда в соответствии с (2.10.8) g – это отображение следующего вида:

$$g(S_t, h_t, i) = \begin{cases} (S_t, h_{t+1}),\ i! = l \text{ (участок трассы еще не закончился)} \\ (S_{t+1}, h_{t+1}),\ i = l \text{ (участок трассы закончился)} \end{cases}, \tag{2.10.9}$$

где l – длина участка трассы (параметр кэша), S_{t+1} – новое состояние кэша, состоящее из элементов основного хранилища с наибольшими частотами попаданий согласно ранжированному интервальному статистическому ряду (т. е. в кэш загружается n_{max} блоков, у которых c_i наибольшее, т.к. интервальный статистический ряд ранжирован по c_i).

$$S_{t+1} = \cup_{i=1}^{n_{max}} O_t^j, \tag{2.10.10}$$

$$\sum_{i=1}^{n_{max}} |O_t^i| \leq M \wedge (K_t^i = \{k_t^i : min^i < k_t^i < max^i\}),$$

$$i = 1..n_{max}, (min^i, max^i) \in h\text{c}_i \subset h_{t+1}$$

где n_{max} $(0 < n_{max} < n_b)$ – максимальное количество блоков основного хранилища, которые можно разместить в кэш-памяти.

Таким образом получаем самоадаптирующийся контейнер данных с кэшем $AH(H, h_0, g, l, n_b)$ с использованием интервального статистического ряда в качестве алгоритма кэширования A.

2.11. Выводы

1. Разработана структурная модель самоадаптирующегося контейнера данных.
2. Предложен способ задания ассоциативного контейнера, нагрузки и процесса ее применения на ассоциативный контейнер.
3. На основе структурной модели разработан способ задания самоадаптирующегося контейнера данных и предложен критерий выбора оптимального контейнера.
4. Разработана модель сложной нагрузки на контейнер, состоящая из смеси гауссовских распределений и способ задания адаптивного кэширующего контейнера данных с использованием интервального статистического ряда.
5. Проведен анализ зависимости оптимального размера кэша по временному критерию для нагрузки, сгенерированной по нормальному распределению. Аналитически выведено несколько утверждений и предложена формула нахождения оптимального размера кэша по среднеквадратичному отклонению ключей в трассе и соотношению скоростей кэша и основного хранилища [7].

Глава 3. Анализ и разработка алгоритмов модулей самоадаптирующегося контейнера с использованием кэша

В данной главе представлены алгоритмы модулей самоадаптирующегося контейнера данных. На основе моделей, описанных в главе 2, разработаны алгоритмы, позволяющие решить задачи, поставленные в главе 1.

Для построения алгоритма кэширующего модуля самоадаптирующегося контейнера данных необходимо выбрать шаблон проектирования. Существует несколько архитектурных шаблонов кэширования. У каждого из них есть свои плюсы и минусы. Выбор среди них осуществляется на основе определенных параметров, описанных в данной главе.

Для решения задачи расщепления смеси нормальных распределений наиболее эффективным алгоритмом является метод Expectation Maximization. Для инициализации данного метода существует несколько алгоритмов инициализации. В данной главе обоснован выбор метода инициализации и описан метод EM.

3.1. Алгоритм кэширующего модуля самоадаптирующегося контейнера

Для реализации самоадаптирующегося контейнера данных, использующего программный кэш необходимо выбрать архитектурный шаблон кэширования. При выборе необходимо учитывать следующие параметры:

1. **Время жизни объектов в кэше**. При использовании кэша может возникнуть необходимость задания времени жизни данных, по истечении которого объекты кэша становятся неактуальными и вычищаются из кэша. Чтобы стратегия кэширования была эффективной необходимо чтобы политика срока жизни объекта соответствовала шаблону доступа к приложениям, использующим данные. Если время жизни объектов в кэше слишком мало, то извлечение данных из основного хранилища и добавление их в кэширующий контейнер будет происходить слишком часто. С другой стороны, при слишком длительном сроке жизни объектов в кэше данные могут стать неактуальными. Необходимо понимать, что

кэш наиболее эффективен в случаях, когда данные часто запрашиваются на чтение и редко записываются.

2. **Удаление данных из кэша**. В большинстве случаев кэш имеет размер значительно меньший чем основное хранилище. При достижении этого размера, часть объектов кэша должно будет быть удалено. Существует много алгоритмов вытеснения объектов из кэша (см. раздел 1.4). Для того чтобы кэш являлся наиболее эффективным необходимо использовать наиболее оптимальную политику кэширования. Однако не всегда целесообразно применять алгоритм вытеснения для каждого отдельного объекта. Например, если извлечение объекта из основного хранилища является трудоемкой операцией, то можно оставить этот объект в кэше путем удаления из кэша более приоритетных согласно политике кэширования, но менее трудоемких для извлечения объектов.
3. **Изначальная инициализация кэша**. Некоторым приложениям необходимы изначальные данные для запуска. Их также можно разместить в кэше. Большая часть архитектурных шаблонов поддерживают такую возможность.
4. **Согласованность**. Некоторые шаблоны кэширования не гарантируют согласованность между основным хранилищем и кэшем. Если изменить объект в основном хранилище извне, то до следующего запроса данного объекта в кэше могут находиться устаревшие данные. Если используется репликация, то при частой синхронизации такая несогласованность может привести к серьезным проблемам.
5. **Локальное (выполняющееся в памяти) кэширование**. Кэш может быть локальным для экземпляра приложения и хранящимся в памяти. Кэш на стороне можно использовать в этой среде, если приложение многократно получает доступ к одним и тем же данным. Тем не менее локальный кэш является закрытым, и поэтому разные экземпляры приложения могут иметь копию одних и тех же кэшированных данных. Эти данные могут быстро стать

несогласованными между кэшами, из-за чего может возникнуть необходимость завершения срока хранения данных, которые находятся в частном кэше, и обновлять их чаще. В таких случаях рекомендуется изучить использование механизма общего или распределенного кэширования.

Существует несколько таких шаблонов:

1. Cache-aside «кэш на стороне» [129]. Самый используемый шаблон кэширования. При таком подходе кэш располагается отдельно и приложение взаимодействует напрямую с кэшем и хранилищем данных. При запросе элемента с заданным ключом в первую очередь проверяется кэш. Если элемент с таким ключом найден в кэше, то объект возвращается приложению из кэша. Если элемент с данным ключом не найден, то объект загружается из базы, возвращается приложению и добавляется в кэш. Схема работы такого подхода изображена на рисунке 3.1.

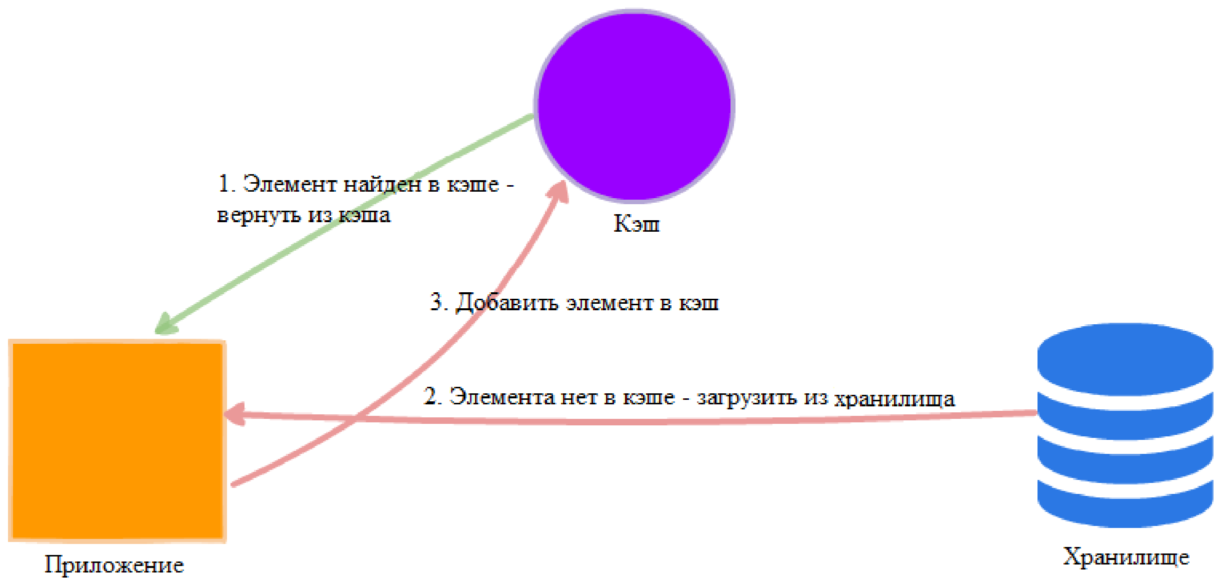

Рис. 3.1. Схема работы шаблона Cache-aside

Данный шаблон хорошо подходит для приложений с большим количеством запросов на чтение. Часто в качестве кэша используется Redis и Memcached.

Преимущества данного подхода:

- Данный подход к кэшированию устойчив к сбоям кэша. Если кэш по какой-то причине будет недоступен, то приложение сможет взаимодействовать с хранилищем напрямую (в данном случае производительность приложения будет значительно снижена).
- Так как происходит загрузка по запросу, в кэше оказываются только запрошенные элементы, что позволяет избежать загрузки в кэш тех данных, которые не были запрошены.

Недостатки данного подхода:

- Каждый промах кэша требует совершить три операции, что может привести к значительным задержкам.
- При использовании данного шаблона данные записываются напрямую в хранилище, что может привести к тому, что в кэше окажутся неактуальные данные. Данный недостаток можно нивелировать путем введение времени жизни объекта (Time to live, TTL) по достижению которого необходимо загружать актуальное состояние объектов из хранилища в кэш.
- При использовании распределенной системы добавление нового узла увеличит задержку, так как кэш изначально будет пустым.

2. Read-Through «сквозное чтение». При использовании данного шаблона приложение всегда считывает данные из кэша, а за синхронизацию между хранилищем данных и кэшем отвечает провайдер кэширования. Read-Through также как и Cache-aside загружает данные по требованию при первом запросе на чтение. Схема работы такого подхода изображена на рисунке 3.2.

Read-Through

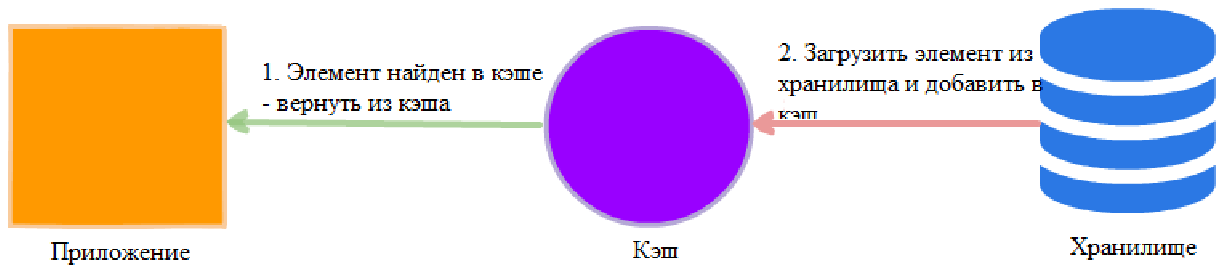

Рис. 3.2. Схема работы шаблона Read-Through

Несмотря на то, что данные шаблоны кэширования очень похожи, есть два ключевых отличия:

- В архитектуре Cache-aside приложение ответственно за извлечение данных из хранилища и добавление их в кэш. При использовании Read-Through эта логика обычно поддерживается специальными библиотеками или отдельным провайдером кэша;
- В отличии от Cache-aside модель данных при Read-Through не может отличаться от модели хранилища данных.

Данная архитектура в целом имеет те же преимущества и недостатки что и Cache-aside, однако имеет следующие преимущества над ним:

- Так как приложение взаимодействует напрямую с кэшем, логика приложения становится гораздо проще;
- Лучшая масштабируемость чтения: существует много ситуаций, когда истекает срок жизни элемента кэша и несколько параллельных пользовательских потоков попадают в хранилище данных. Если умножить это на миллионы кэшированных элементов и тысячи параллельных пользовательских запросов, нагрузка на хранилище данных станет заметно выше. Но сквозное чтение сохраняет элемент кеша в кеше, пока он извлекает его последнюю копию из базы данных. Затем он обновляет элемент кеша.

Конечным результатом является то, что приложение никогда не обращается к хранилищу данных для этих элементов кэша, и загрузка хранилища данных остается минимальной;

- Автоматическое обновление кеша по истечении срока действия: сквозное чтение позволяет кешу автоматически перезагружать объект из хранилища данных по истечении срока его действия. Это означает, что вашему приложению не нужно обращаться к хранилищу данных при пиковой нагрузке, потому что самые свежие данные всегда находятся в кеше;
- Автоматическое обновление кеша при изменении хранилища данных: сквозное чтение позволяет кешу автоматически перезагружать объект из хранилища данных при изменении соответствующих данных в хранилище данных. Это означает, что кеш всегда свежий, и вашему приложению не нужно обращаться к хранилищу данных при пиковой нагрузке, потому что в кеше всегда находятся самые свежие данные.

3. Write-through «сквозная запись». В этой стратегии записи данные сначала записываются в кеш, а затем в хранилище данных. Кеш находится вместе с хранилищем данных, и записи всегда проходят через кеш в основное хранилище. Схема работы такого подхода изображена на рисунке 3.3.

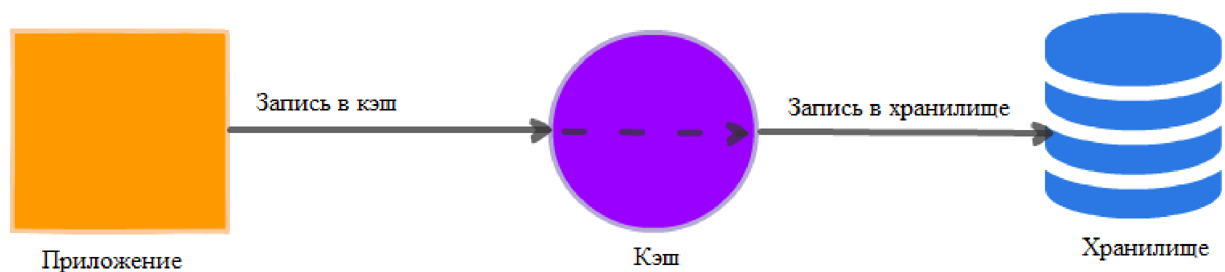

Рис. 3.3. Схема работы шаблона Write-Through

Сам по себе кэш со сквозной записью почти не используются так как на самом деле он создает дополнительную задержку записи, потому что данные сначала записываются в кэш, а затем в основное хранилище данных. Но в сочетании с кэшем сквозного чтения получаются все преимущества сквозного чтения, а также гарантия согласованности данных, освобождающая от использования методов недействительности кэша. Данная стратегия широко используется в различных программных продуктах.

4. Write-around. Данная стратегия похожа на Write-through, однако элемент в кэше только обновляется, а если его там нет, то добавляется только в основное хранилище. Схема работы такого подхода изображена на рисунке 3.4.

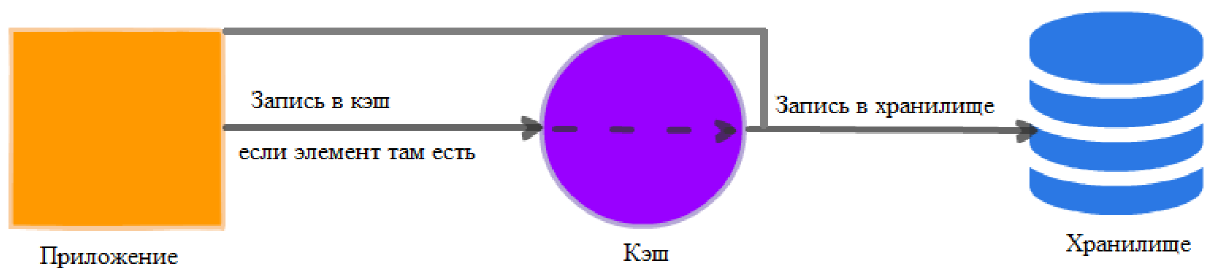

Рис. 3.4. Схема работы шаблона Write-Around

Write-around можно комбинировать со сквозным чтением, что обеспечивает хорошую производительность в ситуациях, когда данные записываются один раз, а читаются достаточно редко или никогда. Например, журналы в реальном времени или сообщения в чате. Точно так же этот шаблон можно комбинировать с cache-aside.

5. Write-Behind (Write-Back) «кэш с обратной записью». При такой архитектуре приложение записывает данные в кэш, которые немедленно подтверждает, а после некоторой задержки записывает данные обратно в базу данных. Схема работы такого подхода изображена на рисунке 3.5.

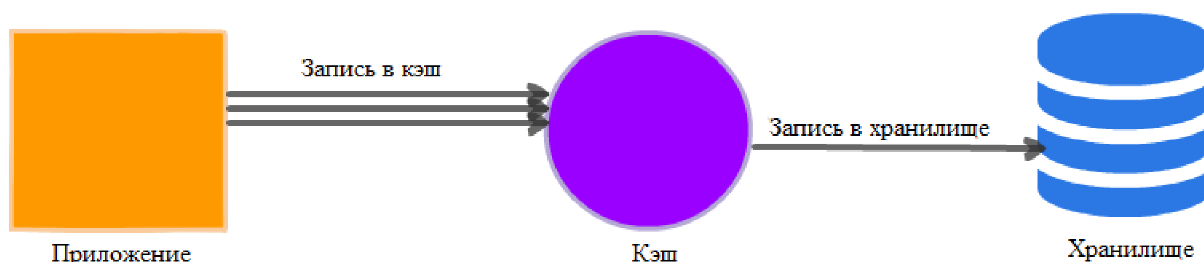

Рис. 3.5. Схема работы шаблона Write-Back

Данный шаблон повышает производительность записи и подходит для нагрузок с большим объемом записи. В сочетании со сквозным чтением он хорошо работает для смешанных рабочих нагрузок, когда самые последние обновленные и доступные данные всегда доступны в кеше.

Кэш с обратной записью устойчив к сбоям хранилища данных и может допускать некоторое время отключения хранилища данных. Если поддерживается пакетная обработка, это может уменьшить общее количество операций записи в базу данных, что снижает нагрузку и снижает затраты, если поставщик хранилища данных взимает плату за количество запросов, например DynamoDB.

Также возможно использование кэша Redis как для резервного кеширования, так и для обратной записи, чтобы лучше поглощать пики во время пиковой нагрузки. Главный недостаток состоит в том, что в случае сбоя кеша данные могут быть безвозвратно потеряны.

Большинство реляционных баз данных (например, InnoDB) имеют кеш с обратной записью, включенный по умолчанию во внутреннем устройстве. Запросы сначала записываются в память, а затем сбрасываются на диск.

6. Refresh-ahead «кэш упреждающего обновления». В данной стратегии кэш настроен таким образом, что он автоматически и асинхронно обновляет недавно использованный элемент кеша до истечения срока его действия.

В результате после того, как часто используемая запись попала в кеш, скорость чтения приложения из кэша не будет снижена из-за того, что запись перезагружается из основного хранилища данных из-за истечения срока действия. Асинхронное обновление запускается только при доступе к объекту, срок действия которого достаточно близок к истечению срока его действия. А если к объекту обращаются после истечения срока его действия, кэш выполнит синхронное чтение из основного хранилища, чтобы обновить его значение. Схема работы такого подхода изображена на рисунке 3.6.

Refresh-ahead

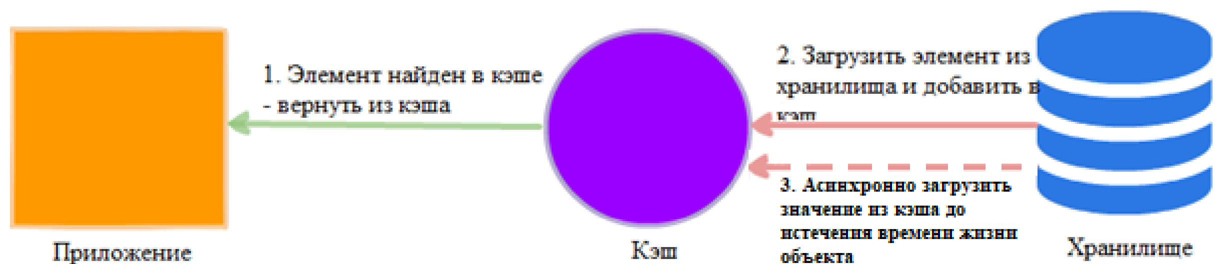

Рис. 3.6. Схема работы шаблона Refresh-ahead

Упреждающее обновление может привести к снижению задержки, если кэш может точно предсказать, какие элементы могут потребоваться в будущем. При полной точности этих прогнозов упреждающее обновление обеспечивает меньшую задержку и отсутствие дополнительных накладных расходов. Чем выше частота неточного прогноза, тем сильнее будет влияние на пропускную способность (поскольку в базу данных будет отправлено больше ненужных запросов) - потенциально даже отрицательное влияние на задержку, если база данных начнет отставать при обработке запросов.

Таким образом, архитектуру кэширующего модуля самоадаптирующегося контейнера стоит выбирать в зависимости от задач и условий, в которых он будет работать.

Для исследования зависимости эффективного размера кэша от среднеквадратичного отклонения (σ) и соотношения скоростей хранилищ для нормального распределения был выбран шаблон cache-aside. Для проверки ***леммы 6*** (формула 2.8.9) достаточно использовать схему без поддержки обновления данных, таким образом нет необходимости вводить время жизни объекта в кэше и реализовывать отдельный провайдер кэша, следовательно данная архитектура удовлетворяет потребностям приложения. Схема использования шаблона cache-aside для исследования зависимости эффективного размера кэша от среднеквадратичного отклонения и соотношения скоростей хранилищ изображена на рисунке 3.7. Кэш и основное хранилище являются ассоциативными структурами данных, которые хранят пары «ключ-значение» (<Tkey, Tvalue>). Ключи k_i на данной схеме принадлежат объектам, сгенерированным по закону нормального распределения для некоторой трассы, а соотношение хранилищ k определяется по соотношению медленной и быстрой памяти.

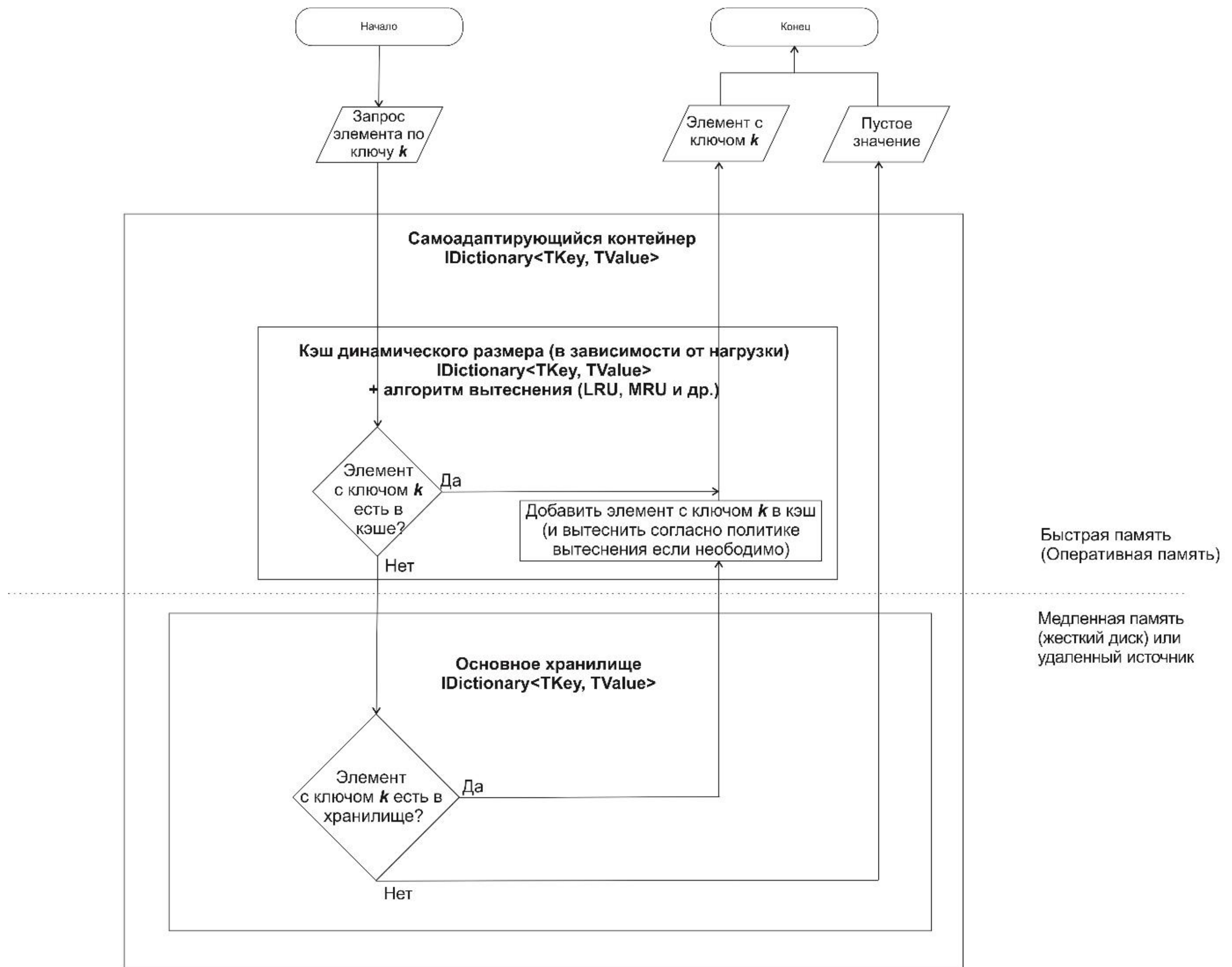

Рис. 3.7. Блок-схема самоадаптирующегося контейнера данных с использованием кэша для исследования зависимости эффективного размера кэша от среднеквадратичного отклонения и соотношения скоростей хранилищ

В разделе 4.1 описаны условия и результат исследования зависимости эффективного размера кэша от среднеквадратичного отклонения (σ) и соотношения скоростей хранилищ для нормального распределения, проведенный с использованием данного алгоритма [7].

3.2. Алгоритм работы модуля определения параметров сложной нагрузки на самоадаптирующиеся контейнеры данных

Для разработки модуля определения параметров сложной нагрузки на самоадаптирующиеся контейнеры данных необходимо решить задачу 2.9.1 из раздела 2.9. Для решения такой задачи в основном используется алгоритм EM

(Expectation Maximization) и его различные модификации [130] [131]. Для решения задачи алгоритмом EM вводится вспомогательный вектор скрытых переменных $\boldsymbol{G}$, который позволяет упростить задачу максимизации. Данный вектор состоит из элементов g_{nk}, которые определяют некую апостериорная вероятность того, что элемент выборки x_n принадлежит кластеру k. EM алгоритм состоит из последовательного выполнения 2-х шагов:

1. Шаг E (Expectation) – пересчет вектора $\boldsymbol{G}$ на основе текущего значения вектора θ по формуле:

$$g_{nk} = \frac{\pi_k N(x_n|\mu_k,\ \sigma_k^2)}{\sum_{j=1}^{K} \pi_j N(x_n|\mu_j,\ \sigma_j^2)}, k = 1..K \tag{3.1}$$

2. Шаг M (Maximization) – максимизация функции 2 и вычисление нового значения вектора θ на основе вектора $\boldsymbol{G}$ и текущего значения вектора θ (для $k = 1..K$) по формулам:

$$\pi_k^{new} = \frac{1}{K} \sum_{n=1}^{K} g_{nk} \tag{3.2}$$

$$\mu_k^{new} = \frac{1}{K\pi_k^{new}} \sum_{n=1}^{K} g_{nk} x_n$$

$$\sigma_k^{2\,new} = \frac{1}{K\pi_k^{new}} \sum_{n=1}^{K} g_{nk} \left(x_n - \mu_k^{new}\right)^2$$

Итерации происходят до сходимости (норма разности векторов скрытых переменных или изменение логарифмического правдоподобия на каждой итерации не будет превышать заданную константу) или достижения максимального числа итераций.

Одним из важных недостатков алгоритма EM является то, что данный алгоритм находит локальный экстремум функции правдоподобия, значение которого может оказаться гораздо ниже, чем глобальный максимум. Таким образом, в зависимости от выбора начального приближения алгоритм может сходиться к разным точкам. Существует несколько подходов к инициализации начальных параметров (матожидание, дисперсия и вес для каждого одномерного нормального распределения $N(\mu_k, \sigma_k^2)$ k=1...K) [132] [133]. Наиболее простой

способ задания начальных значений это случайный, когда вес задается как 1/K, дисперсия и матожидания выбираются случайным образом. Данный способ был протестирован в первую очередь, так как он не требует дополнительных затрат, однако даже при небольшом количестве компонент, в большинстве случаев результирующие кластеры сильно отличались от входных.

Кроме того, были рассмотрены алгоритмы rndem и emEm [134] [135]. Данные алгоритмы также случайным образом задают входные параметры, но выполняет несколько неполных итераций основного алгоритма, после чего выбирают начальные значения с наибольшим значением функции правдоподобия. Однако в случае большого объема данных, выполнение нескольких итераций может занять существенное количество времени. Другие популярные подходы основаны на агломеративной иерархической кластеризации (hierarchical agglomerative clustering (HAC)) [136], однако эти методы работают даже медленнее чем предыдущие.

Одним из самых эффективных согласно исследованию [132], является алгоритм kmeans++ [137]. Поэтому именно он был реализован в модуле определения параметров сложной нагрузки. Кроме того, в условиях big data могут быть использованы масштабируемые версии данного алгоритма [138] или даже версия с использованием технологии MapReduce [139] [140]. kmeans++ представляет собой алгоритм выбора начальных значений (центров кластеров, называемых также центроидами) для проведения кластеризации. Для поиска таких значений необходимо выполнить несколько шагов:

1. Случайным образом из выборки χ выбирается первый центроид c_1;
2. Для всех оставшихся данных выборки χ находится расстояние $D(x)$ до ближайшего центроида;
3. Выбирается новый центроид c_i таким образом, что элемент $x \in \chi$ может быть выбран в качестве центроида с вероятностью $\frac{D(x)^2}{\sum_{x \in \chi} D(x)^2}$;
4. Шаги 2 и 3 необходимо повторять пока не будут выбраны все центроиды.

Кроме того, необходимо учитывать, что данные постоянно поступают в контейнер, зачастую в большом количестве и на большой скорости. Следовательно,

необходимо применять специальные алгоритмы кластеризации на потоке [141] [142]. К таким алгоритмам относится EM алгоритм с использованием буфера. Модуль самоадаптирующегося контейнера данных был реализован с применением данного алгоритма: сначала данные в режиме онлайн накапливаются в буфере, а затем в режиме оффлайн через некоторые заданные промежутки времени/заданное количество вновь прибывших элементов или по запросу контейнера (в случае если кластеризацию нужно провести немедленно) выполняется EM алгоритм кластеризации на буфере. Размер буфера является настраиваемым параметром и определяет, насколько быстро устаревают данные [8].

3.3. Алгоритм работы адаптивного кэширующего контейнера данных с использованием интервального статистического ряда

На основе математической модели 2.10.7 из раздела 2.10 был разработан алгоритм кэширующего контейнера данных с использованием интервального статистического ряда:

1. <u>Инициализация.</u> В процессе инициализации основное хранилище разбивается на n_b блоков по формуле 2.10.2, как показано на рисунке 3.8.

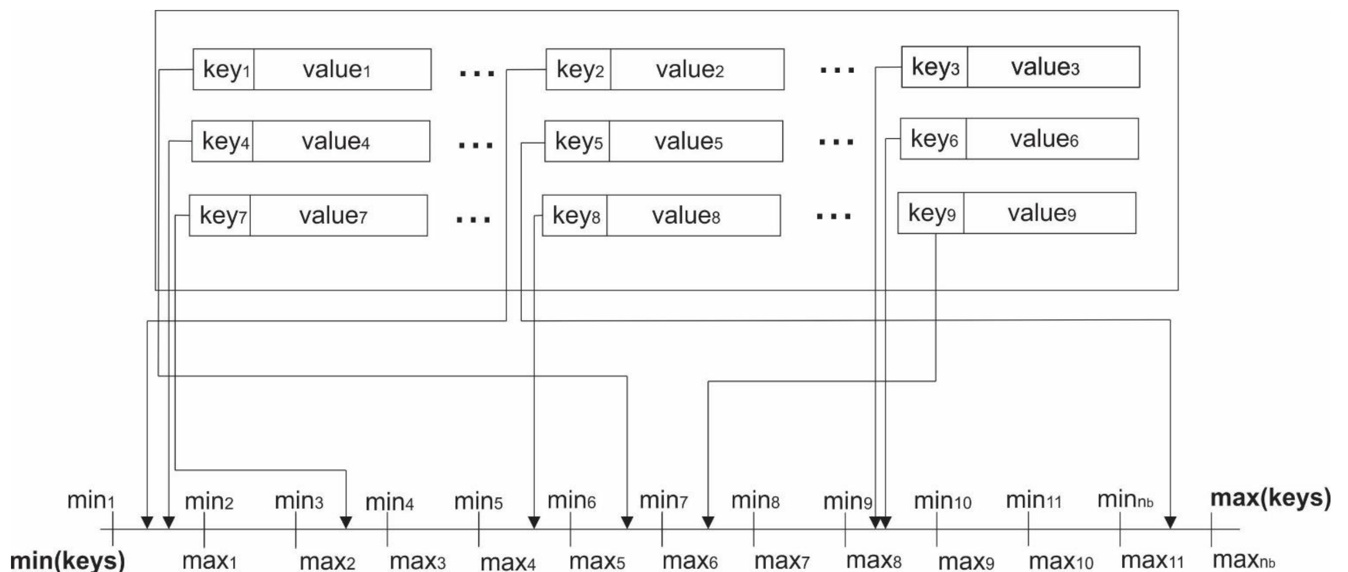

Рис. 3.8. Схема разбиения основного хранилища на блоки

Блок-схема работы данной процедуры изображена на рисунке 3.9. В результате выполнения получается набор блоков, состоящий из элементов, каждый из которых содержит:

- Минимальное значение блока;
- Максимальное значение блока;
- Ключи основного хранилища, которые располагаются в интервале [минимальное значение блока; максимальное значение блока].

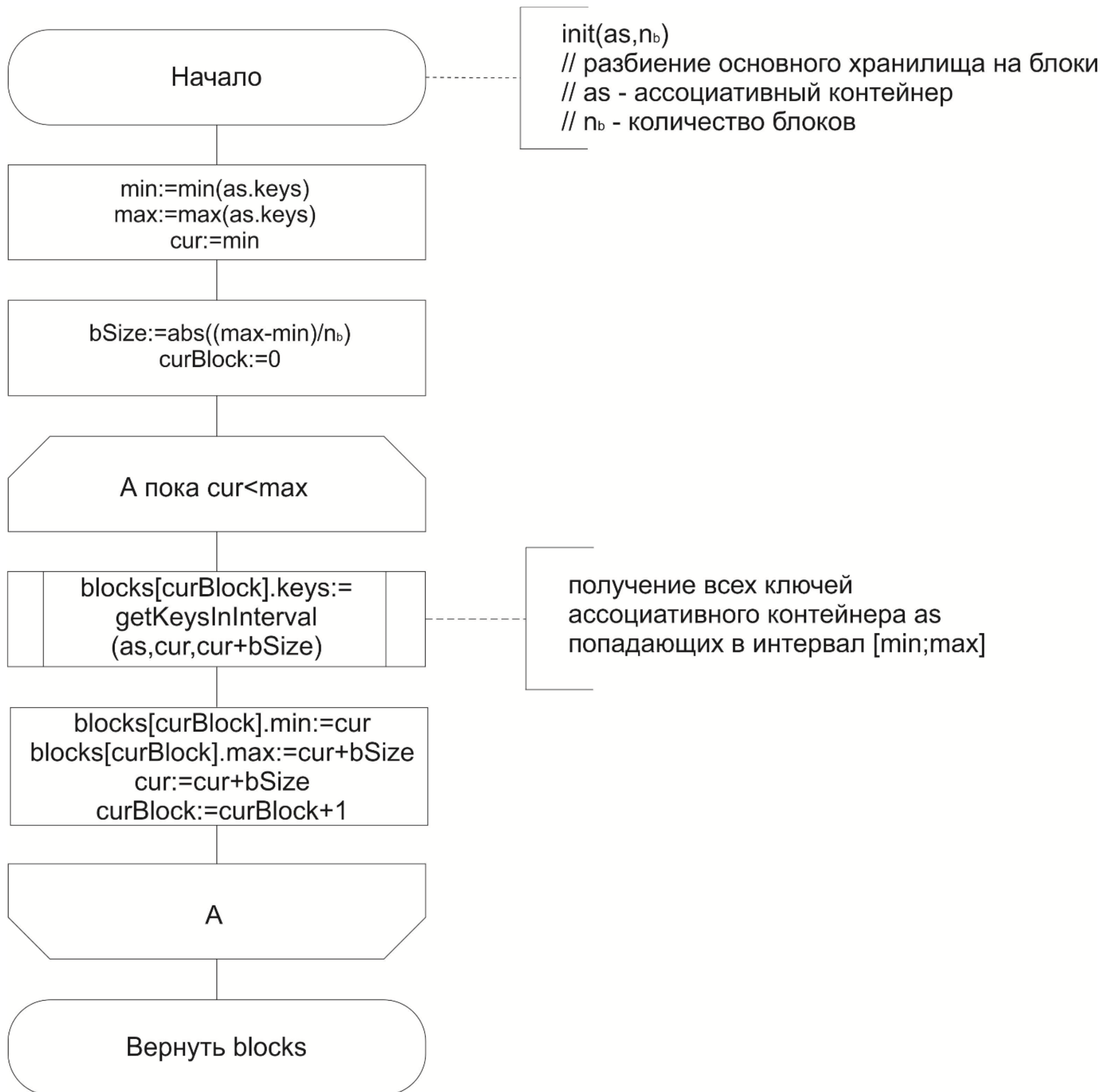

Рис. 3.9. Блок-схема инициализации кэширующего контейнера

2. Построение нулевого интервального статистического ряда и очистка буфера. На основе полученного на шаге 1 набора блоков, строится нулевой интервальный статистический ряд, т.е. для каждого интервала выставляется значение 0. Также очищается буфер, содержащий l ключей запросов.

3. <u>Обработка поступления запроса в кэширующий контейнер.</u> Если в кэше есть такой ключ, то в ответ возвращается объект из кэша, если нет – из основного хранилища. При этом, если кол-во ключей в буфере не достигло значения *l*, то ключ запроса добавляется в буфер. Иначе выполняются шаги 4-5.
4. <u>Построение ранжированного интервального статистического ряда для буфера</u> по следующему правилу: для каждого интервала интервального статистического ряда подсчитывается сколько ключей из буфера попадает в интервал (рис. 3.10), далее интервальный статистический ряд упорядочивается по количеству ключей от большего к меньшему.

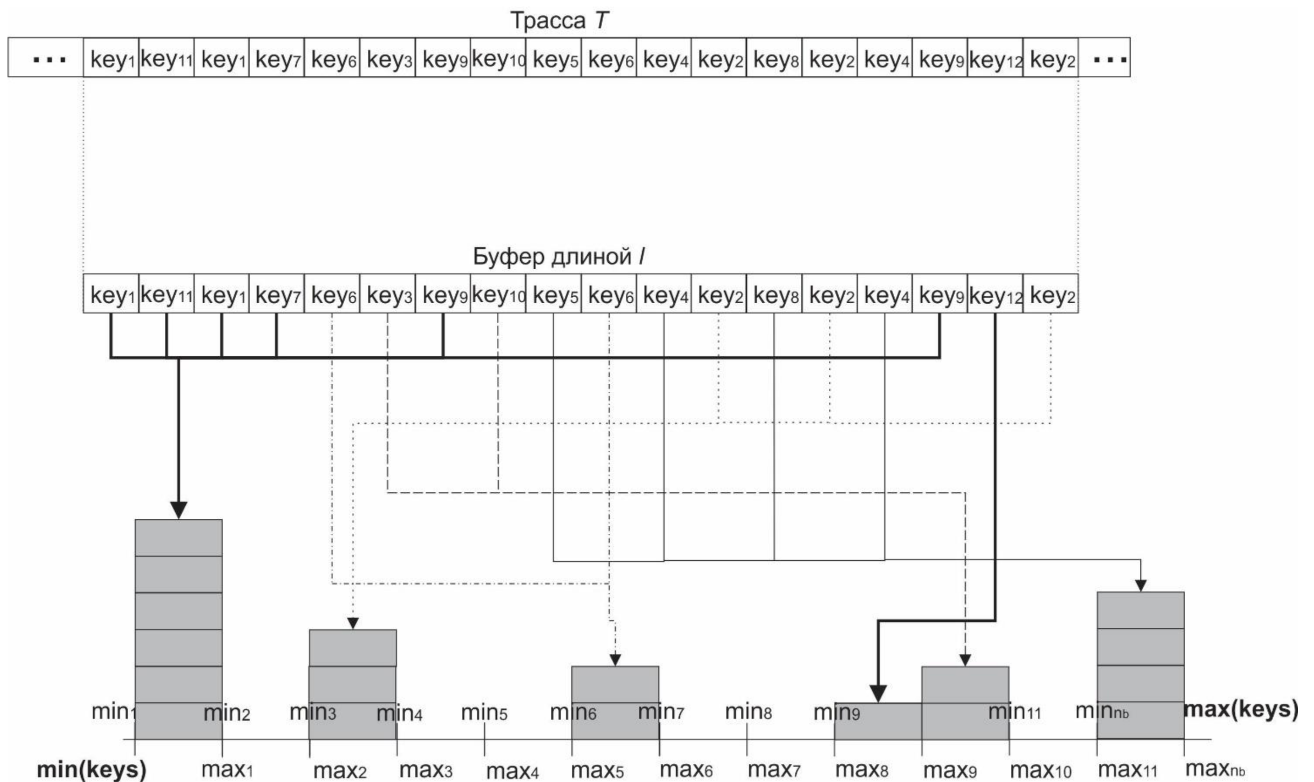

Рис. 3.10. Формирование интервального статистического ряда по буферу длиной l, заполненному ключами участка трассы T

Для того чтобы подсчитать сколько ключей буфера попадает в каждый интервал интервального статистического ряда можно применить очевидный алгоритм. Для каждого ключа буфера перебирать все интервалы и проверять попадает ли он в данный интервал. Однако это

требует O(n) времени, где n – количество интервалов. Для того чтобы выполнить поиск интервала эффективно было применено дерево интервалов (дерево отрезков, segment tree или interval tree) [35].

Дерево интервалов – это древовидная структура данных для хранения интервалов. В частности, она позволяет эффективно находить все интервалы, которые пересекаются с любым заданным интервалом или точкой. Это часто необходимо для оконных запросов, например, чтобы найти все дороги на карте внутри прямоугольного окна просмотра или найти все видимые элементы внутри трехмерной сцены. Временная сложность данного дерева показана в таблице 3.1.

Таблица 3.1 – Временная сложность интервального дерева

Алгоритм	**Среднее**	**Худший случай**
Расход памяти	O(n)	O(n)
Поиск	O(log n)	O(n)
Вставка	O(log n)	O(n)
Удаление	O(log n)	O(n)

Для построения ранжированного интервального статистического ряда для буфера вставка и удаление не требуются, однако требуется один раз построить интервальное дерево что потребует n вставок в дерево, т.е. O(nlog n) времени.

Интервальное дерево реализуется с помощью расширения сбалансированного бинарного дерева (такого как красно-черное, АВЛ и т.д.).

Каждый узел интервального дерева содержит следующую информацию:

- интервал [min^i; max^i];
- максимальное значение max^i в поддереве с корнем в этом узле $max^{subtree}$.

Минимальное значение интервала min^i используется в качестве ключа для поддержания порядка в BST. Операции вставки и удаления аналогичны операциям вставки и удаления в самобалансирующемся BST. Устройство интервального дерева изображено на рисунке 3.11.

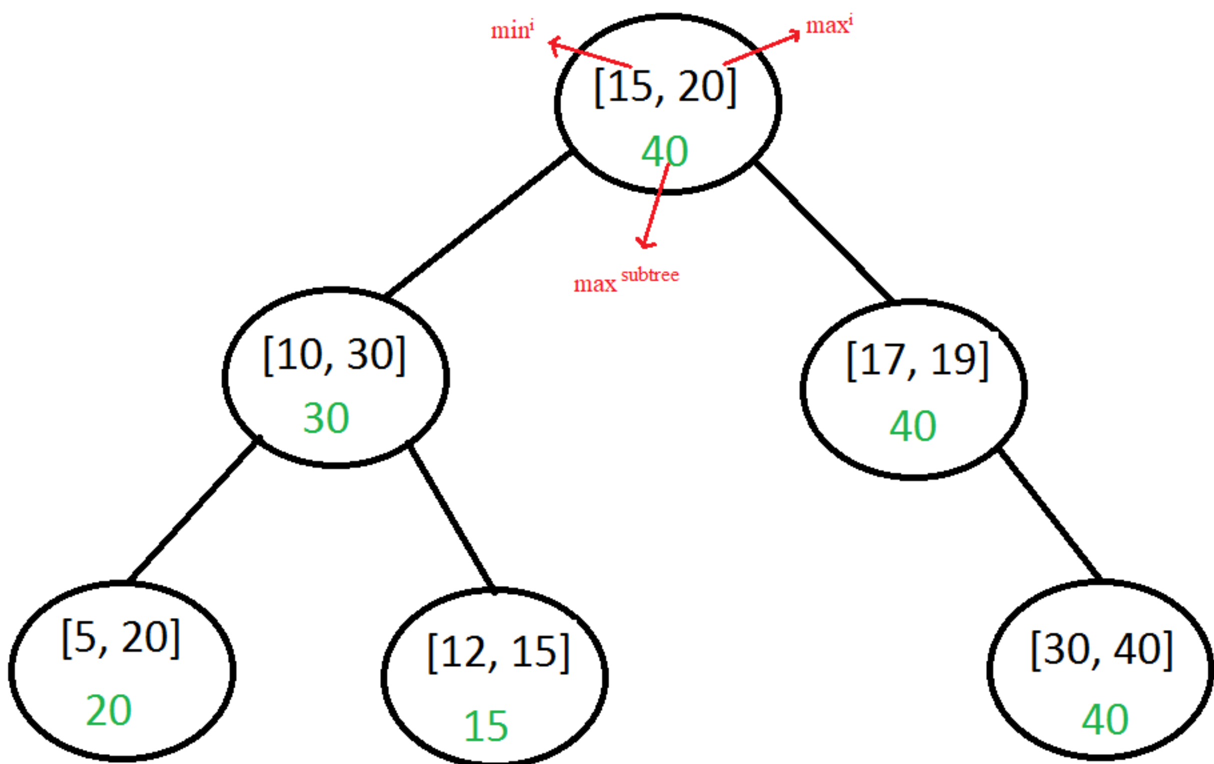

Рис. 3.11. Интервальное дерево

Поиск интервала, содержащего указанную точку x, происходит по следующему алгоритму (см. рис. 3.12):

1. Если интервал текущего узла содержит точку x, то нужный интервал найден.

2. Если левый потомок непустой и $max^{subtree}>x$, переход к пункту 1 с левым потомком в качестве текущего узла. Иначе переход к пункту 1 с правым потомком в качестве текущего узла.

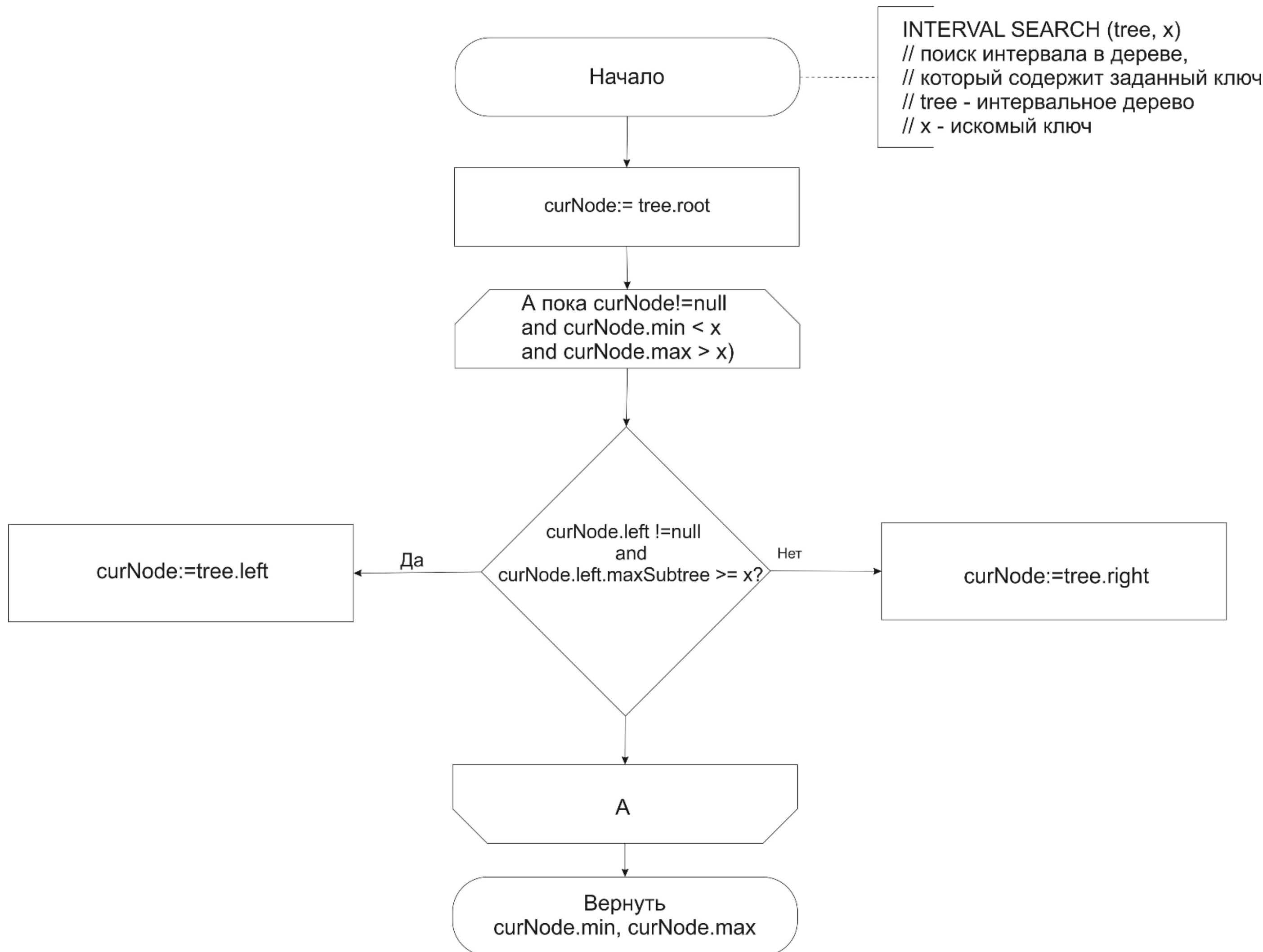

Рис. 3.12. Алгоритм поиска в интервальном дереве

Данный алгоритм подтверждается следующим образом. Пусть искомый интервал *int*. Возможны две ситуации:

- Ситуация 1. Когда происходит поиск в правом поддереве, должно выполняться одно из следующих утверждений:
  - Точка находится в интервале, находящемся в правом поддереве: в этом случае необходимо вернуть найденный интервал из правого поддерева;

  - Ни в одном поддереве нет интервала, содержащего искомую точку: переход к правому поддереву происходит только тогда, когда либо left=null, либо значение $left.max^{subtree} < int.min$. Таким образом, интервал не может присутствовать в левом поддереве.
- Ситуация 2. Когда происходит поиск в левом поддереве, должно выполняться одно из следующих утверждений:
  - Точка находится в интервале, находящемся в левом поддереве: в этом случае необходимо вернуть найденный интервал из левого поддерева;
  - Ни в одном поддереве нет интервала, содержащего искомую точку – это самый сложный случай, необходимо учитывать следующие факты:
    - Поиск происходит в левом поддереве, потому что $int.min <= left.max^{subtree}$;
    - $left.max^{subtree}$ является максимумом одного из интервалов, скажем [a, max] в левом поддереве;
    - Поскольку не существует интервала *int*, который бы содержал искомую точку в левом поддереве, *int*.min должен быть меньше, чем "a";
    - Все узлы в BST упорядочены по минимальному значению интервала, поэтому все интервалы в правом поддереве должны иметь минимальное значение больше, чем «a»;

    Из приведенных выше фактов можно сделать вывод, что все интервалы в правом поддереве имеют минимальное значение, превышающее *int*.min. Таким образом, не существует интервала *int*, содержащего искомую точку, в правом поддереве.

5. Заполнение кэша. В кэш помещаются все объекты, ключи которых попадают в интервалы, для которых произошло наибольшее число запросов за последнее количество запросов *l* по следующему правилу:
   - пока кэш не заполнен по интервальному статистическому ряду определяются интервалы с наибольшим числом попаданий;

- по набору блоков, полученном в процессе инициализации, определяются все ключи основного хранилища, которые необходимо загрузить в кэш;
- в кэш загружаются все объекты основного хранилища, у которых ключи совпадают с найденными на предыдущем этапе.

Схема заполнения кэша изображена на рисунке 3.13. Далее происходит переход к шагу 2.

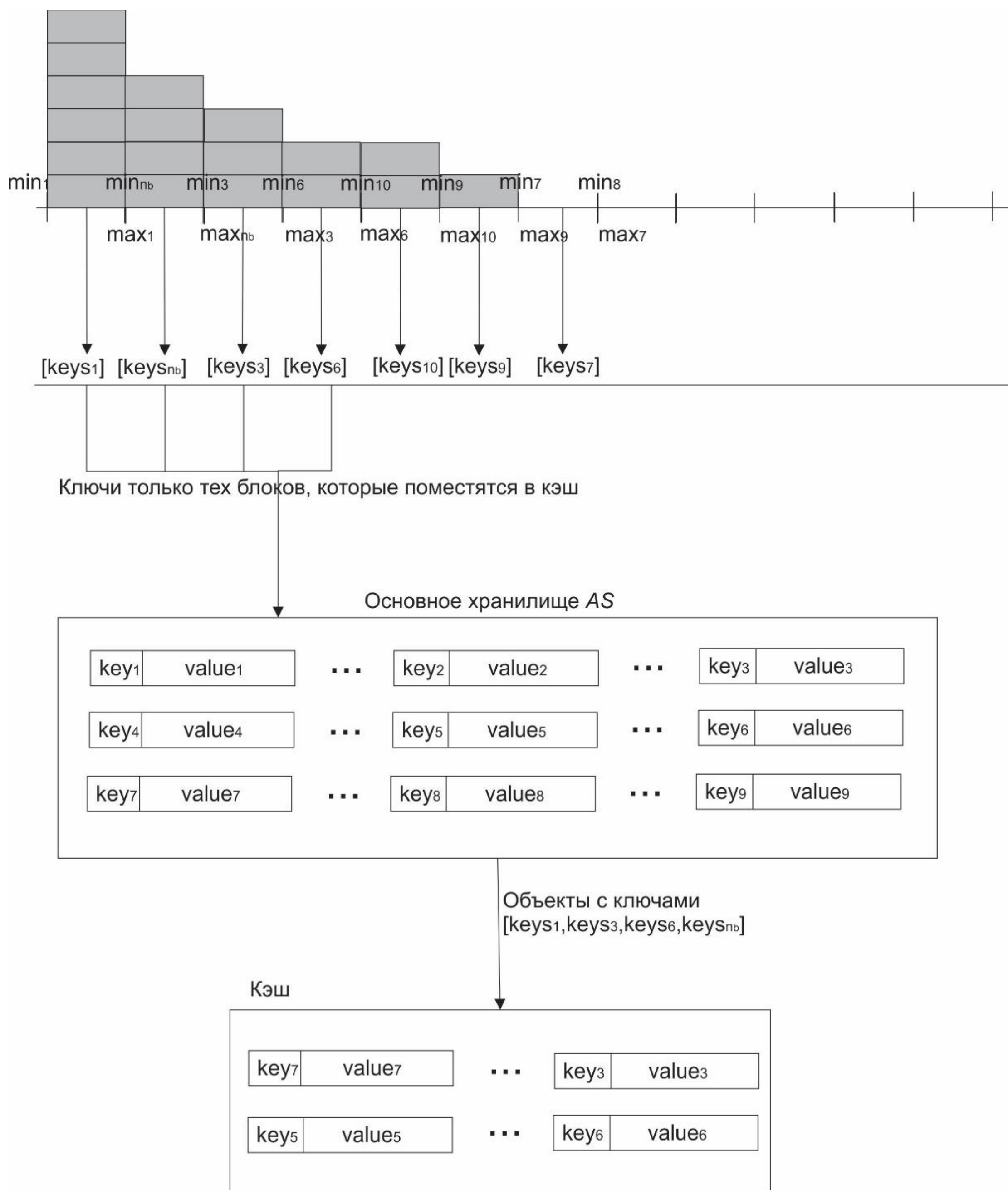

Рис. 3.13. Схема заполнения кэша по интервальному статистическому ряду

Шаг 3 выполняется каждый раз при обращении к кэширующему контейнеру. Общий алгоритм работы кэширующего контейнера с использованием интервального статистического ряда изображен на рисунке 3.14.

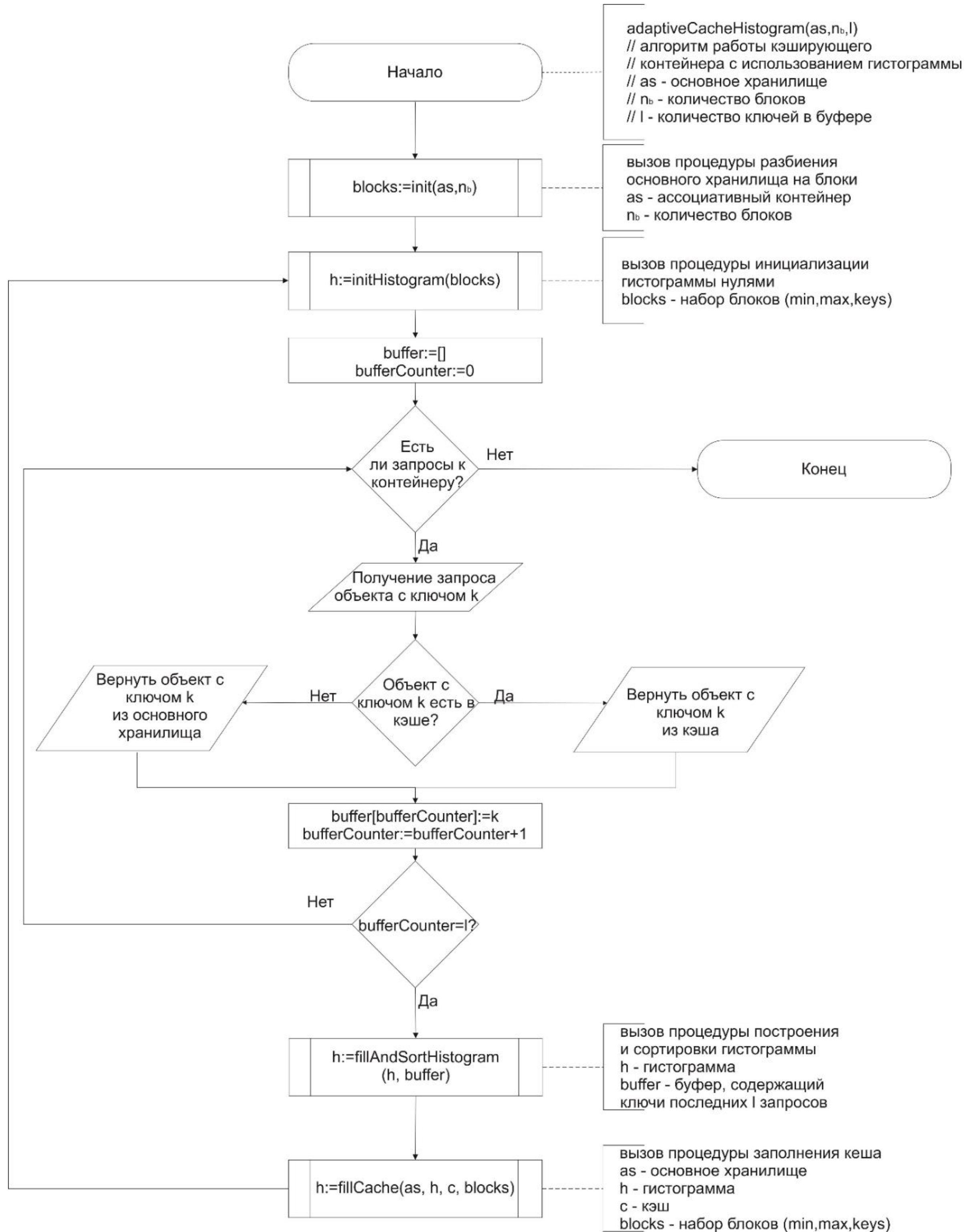

Рис. 3.14. Блок-схема кэширующего контейнера с использованием интервального статистического ряда

3.4. Выводы

1. Разработан алгоритм кэширующего модуля самоадаптирующегося контейнера, на котором также возможно исследовать зависимость оптимального размера кэша от соотношения скоростей хранилищ и среднеквадратичного отклонения ключей в нагрузке.
2. На основе алгоритма EM с инициализацией kmeans++ разработан алгоритм модуля определения параметров сложной нагрузки [8], состоящей из смеси нормальных распределений.
3. Разработан алгоритм работы адаптивного кэширующего контейнера данных с использованием интервального статистического ряда.

Глава 4. Вычислительные эксперименты

В данной главе приводятся результаты вычислительных экспериментов по исследованию эффективности разработанных в диссертации модулей самоадаптирующегося контейнера данных, а также проведено исследование зависимости эффективности кэша от среднеквадратичного отклонения и соотношения скоростей хранилищ для нормального распределения [7].

4.1. Исследование эффективности применения кэша в зависимости от среднеквадратичного отклонения и соотношения скоростей хранилищ для нормального распределения

4.1.1. Описание условий тестирования

Для решения поставленной задачи было проведено тестирование. Для этого было реализовано консольное приложение языка C#, которое использует:

1. Библиотеку Troschuetz.Random для генерации ключей по закону нормального распределения.
2. Класс Stopwatch [143] языка C# для подсчета времени выполнения (в тактах процессора) функции поиска в контейнере.
3. Программу Excel 2013 [144] для визуализации полученных численных результатов.

Тестирование с помощью данного приложения проводилось на следующей программно-аппаратной платформе:

1. Processor: Intel Core i7 6500U @ 2.50 GHz (2 cores).
2. RAM: 16 GB.
3. OS: Windows 10.
4. MS Visual Studio 2015.

В процессе исследования был проведен ряд тестов, параметры которых заданы следующим образом:

1. Элементом кэша и основного хранилища является пара «ключ-значение» (<string, int>), где ключом является строка языка C# (*string*) длиной в 10 символов, значением выступает случайное число типа *int* языка C#;
2. Основное хранилище содержит 10^5 элементов, которые генерируется случайным образом.
3. Запрашивается 10^7 элементов по ключам, которые генерируются следующим образом:
   a. для заданного среднеквадратичного отклонения σ согласно закону нормального распределения $N(0, \sigma^2)$ генерируются целые числа;
   b. полученные числа трансформируются в строки длиной в 10 символов путем добавления нулей слева для недостающих символов. Например, для целого числа 1256 строка будет «0000001256», а для числа 965169 – «0000965169».

   Таким образом параметр σ показывает, насколько большой разброс значений в нагрузке, т.е. чем больше σ, тем меньше вероятность повторного запроса ключа, чем меньше σ, тем больше вероятность повторного запроса.
4. В ходе эксперимента основное хранилище эмулировалось с помощью ассоциативной структуры данных языка C# Dictionary<string, int>, расположенной в оперативной памяти. Таким образом, соотношение скорости кэша и основного хранилища (далее обозначается $\boldsymbol{k}$) для эксперимента равно единице ($\boldsymbol{k} = 1$).
5. Для каждого теста отдельно вычисляется время, затраченное на поиск в кэше ($t_{кэша}$) и поиск в основном хранилище ($t_{осн.хранилища}$).

Один тест представляет собой комбинацию из нескольких параметров:

1. Размер кэша M. Рассматривались значения $M \in \{10, 9100, 18190, 27280, 36370, 45460, 54550, 63640, 72730, 81820, 90910, 100000, 500000, 1000000, 2000000\}$ элементов в кэше.

2. Среднеквадратичное отклонение набора запрашиваемых ключей $\sigma \in \{10, 50, 100, 500, 1000, 5000, 10000, 15000, 20000\}$ при матожидании μ = 0 для нормального распределения N(μ, σ^2).
3. Используемый алгоритм вытеснения $A \in \{LRU, MRU\}$, где LRU (Least Recently Used) [103] и MRU (Most Recently Used) [113]. Данное исследование направлено на нахождение общих зависимостей, независимо от конкретных алгоритмов вытеснения. Поэтому для тестирования были выбраны алгоритмы, имеющие принципиально противоположные стратегии вытеснения (LRU вытесняет неиспользованный дольше всех элемент, а MRU – последний использованный), а результаты тестов были усреднены.
4. Ассоциативная структура данных для кэша, реализованная в языке C# (Dictionary<string, int>, SortedDictionary<string, int> и SortedList<string, int>) [145].

Было проведено по 20 тестов каждой из комбинаций параметров и вычислено среднее арифметическое время поиска в кэше и основном хранилище по всем комбинациям и таким образом получено среднее время $t_{\text{среднее кэша}}$ и $t^{k=1}_{\text{среднее осн.хранилища}}$ для всех пар (σ, M) при $\boldsymbol{k} = 1$ (т.к. во время эксперимента основное хранилище расположено в оперативной памяти, а, следовательно, соотношение скоростей равно единице) [7].

4.1.2. Вычислительный эксперимент и анализ

Используя полученные в результате эксперимента данные по среднему времени поиска в кэше и основном хранилище, расположенном в оперативной памяти, можно вычислить общее время работы самоадаптирующегося контейнера для ситуаций, когда основное хранилище содержит достаточно много элементов и располагается на жестком диске или удаленном источнике. В этом случае соотношение скорости кэша и основного хранилища будет больше единицы ($\boldsymbol{k}$>1), т.к. скорость самых современных SSD дисков в разы меньше скорости оперативной

памяти, а для HDD дисков и удаленных источников достигает сотен тысяч раз и более (в зависимости от конфигурации компьютера и скорости доступа к удаленному источнику). Таким образом среднее время поиска в основном хранилище может быть вычислено как $k * t_{\text{среднее осн.хранилища}}^{k=1}$ (т.к. время поиска на диске или удаленном источнике в $\boldsymbol{k}$ раз больше чем время поиска в оперативной памяти), а среднее общее время работы контейнера для всех пар (σ, M) может быть получено по формуле:

$$t_{\text{среднее общее}} = t_{\text{среднее кэша}} + k * t_{\text{среднее осн.хранилища}}^{k=1}, \quad (4.1)$$

где $\boldsymbol{k}$ – переменная, обозначающая необходимое соотношение скорости кэша и основного хранилища,

$t_{\text{среднее кэша}}$ – среднее время поиска (в тактах процессора) в кэше, расположенном в оперативной памяти,

$t_{\text{среднее осн.хранилища}}^{k=1}$ – среднее время поиска (в тактах процессора) в основном хранилище, расположенном в оперативной памяти (при $\boldsymbol{k}$=1).

Применяя данную формулу и используя средства Excel, было построено большое количество графиков зависимости времени работы контейнера от среднеквадратичного отклонения ключей в нагрузке и размера кеша для различных значений $\boldsymbol{k}$. В результате анализа этих графиков было выявлено, что при указанных выше условиях и входных данных можно выделить три различные по характеру графиков группы в зависимости от $\boldsymbol{k}$:

1. $\boldsymbol{k} < 8$ (кэш ненамного быстрее основного хранилища). В этом случае наиболее оптимальным будет не использовать кэш вообще, т.к. время выполнения без него меньше, чем при любых размерах кэша. Это связано с тем, что при небольших значениях k, время, затраченное на поиск в кэше больше, чем выгода от его использования. На рисунках 4.1 и 4.2 наглядно изображено что для любых значений среднеквадратичного отклонения минимальные значения достигаются при нулевом размере кэша (стрелками указано, что

минимум на графиках достигается при нулевом размере кэша). В таблице 4.1 представлены результаты тестов для k=5. Таким образом, данная группа экспериментов при таких значениях ***k*** подтверждает ***лемму 1*** из раздела 2.8 и $K_0 = \{\forall k \in R: 1 < k < 14\}$.

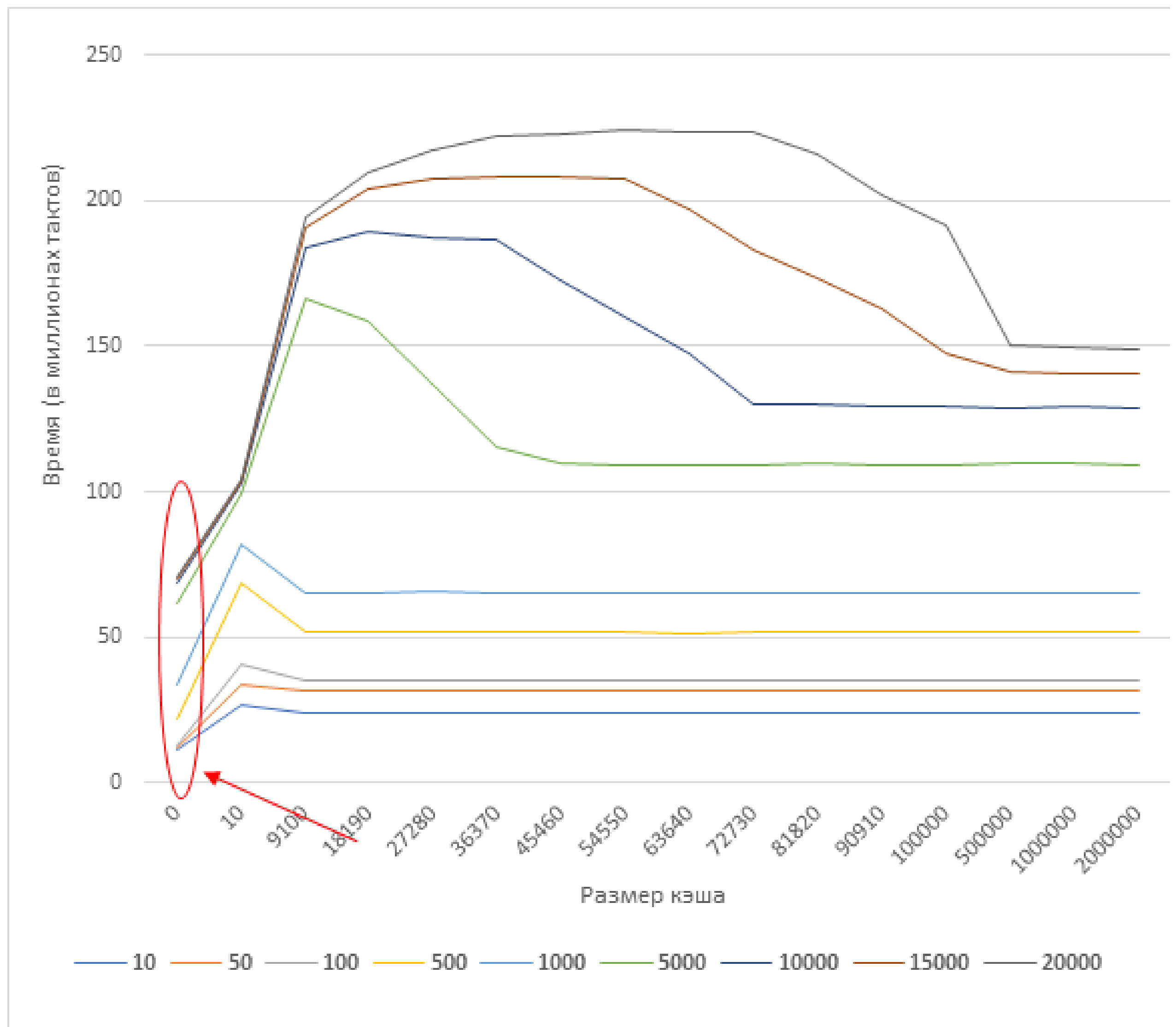

Рис. 4.1. Зависимость времени выполнения поиска в контейнере от размера кэша для различных σ при k=5. Различными цветами изображены различные значения σ

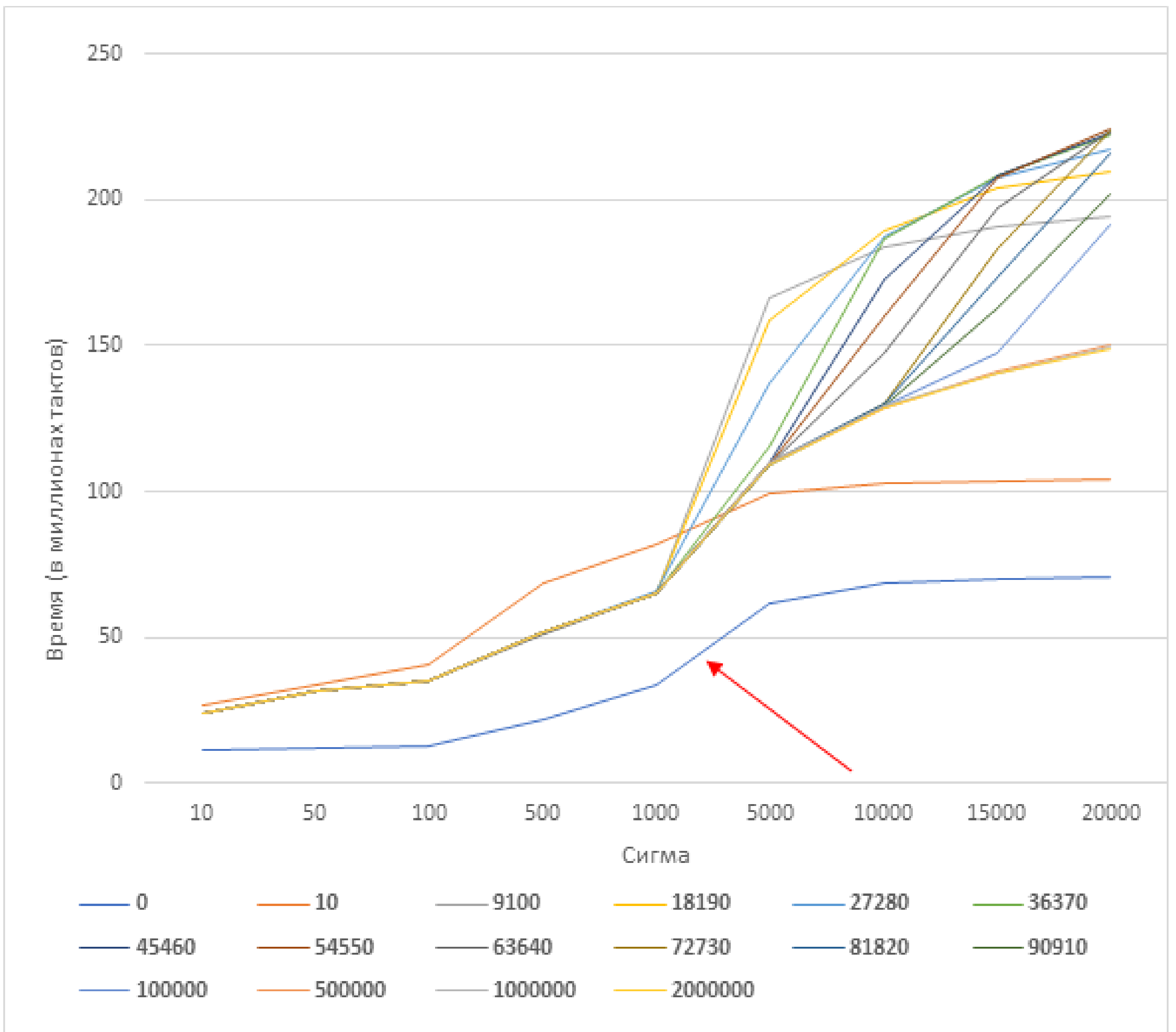

Рис. 4.2. Зависимость времени выполнения поиска в контейнере от σ для различных размеров кэша при $k=5$. Различными цветами изображены различные размеры кэша

Таблица 4.1 – Время поиска в зависимости от σ и размера кеша для $k=5$

	σ								
Размер кеша	**10**	**50**	**100**	**500**	**1000**	**5000**	**10000**	**15000**	**20000**
0	11008775	11717288	12987867	21848120	33667977	61566760	68428464	69680182	70807284
10	26940705	33890779	40835969	68498983	81871135	99303146	102834053	103707956	103845549
9100	23845447	31277860	35182731	51799736	65341135	166050254	184057853	191102304	194483186
18190	23840746	31314295	35246913	51764485	65095889	158558086	189196597	203905118	209896699
27280	23923557	31357142	35175279	51649246	65459080	136677576	187540742	207563583	217477651
36370	23852812	31345004	35122799	51525828	65150845	115482041	186774839	208546705	221866535
45460	23830789	31370141	35141213	51487477	65000654	109702657	172906803	208553253	222735013
54550	23879114	31352730	35219454	51850471	65122474	109123617	159923849	207654561	224457537

63640	23846348	31325336	35178629	51432401	65344654	109195183	147747251	197333128	223470878
72730	23856182	31365298	35156191	51602573	64936630	109068561	129887982	183386692	223257982
81820	23931219	31407959	35134752	51612698	64890774	109982322	129943863	173610780	215949778
90910	23824473	31292224	35211458	51477154	64830806	109140683	129291463	162604933	201759919
100000	23843379	31376769	35201755	51553538	64965822	108824473	129628560	147536611	191798905
500000	23872934	31377701	35154747	51678465	65384576	109447917	128939390	141179342	150550711
1000000	23840667	31339943	35078915	51574599	65066264	109459951	129133388	140370802	149799327
2000000	23836529	31327811	35159711	51846836	64963718	108802369	128970369	140220465	149080165

2. $\boldsymbol{k}$>14 (кэш намного быстрее основного хранилища). В этом случае оптимальным размером кэша является примерно 8σ, то есть практически все элементы участвующие в выборке помещаются в кэш и затраты на поиск в таком кэше окупаются. Больше этого значения увеличение кэша не дает ощутимого прироста в производительности. На рисунках 4.3 и 4.4 можно заметить, что время поиска достигает минимума для каждой σ при достижении размера кэша примерно 8σ, после чего меняется незначительно. В таблице 4.2 представлены результаты эксперимента для k=20. Следовательно ***лемма 3*** из раздела 2.8 верна и $K_b = \{\forall k \in R: k > 14\}$.

   При увеличении k увеличивается разность между временем поиска без использования кэша и временем поиска с использованием оптимального размера кэша, т.е. эффективность использования увеличивается с ростом k.

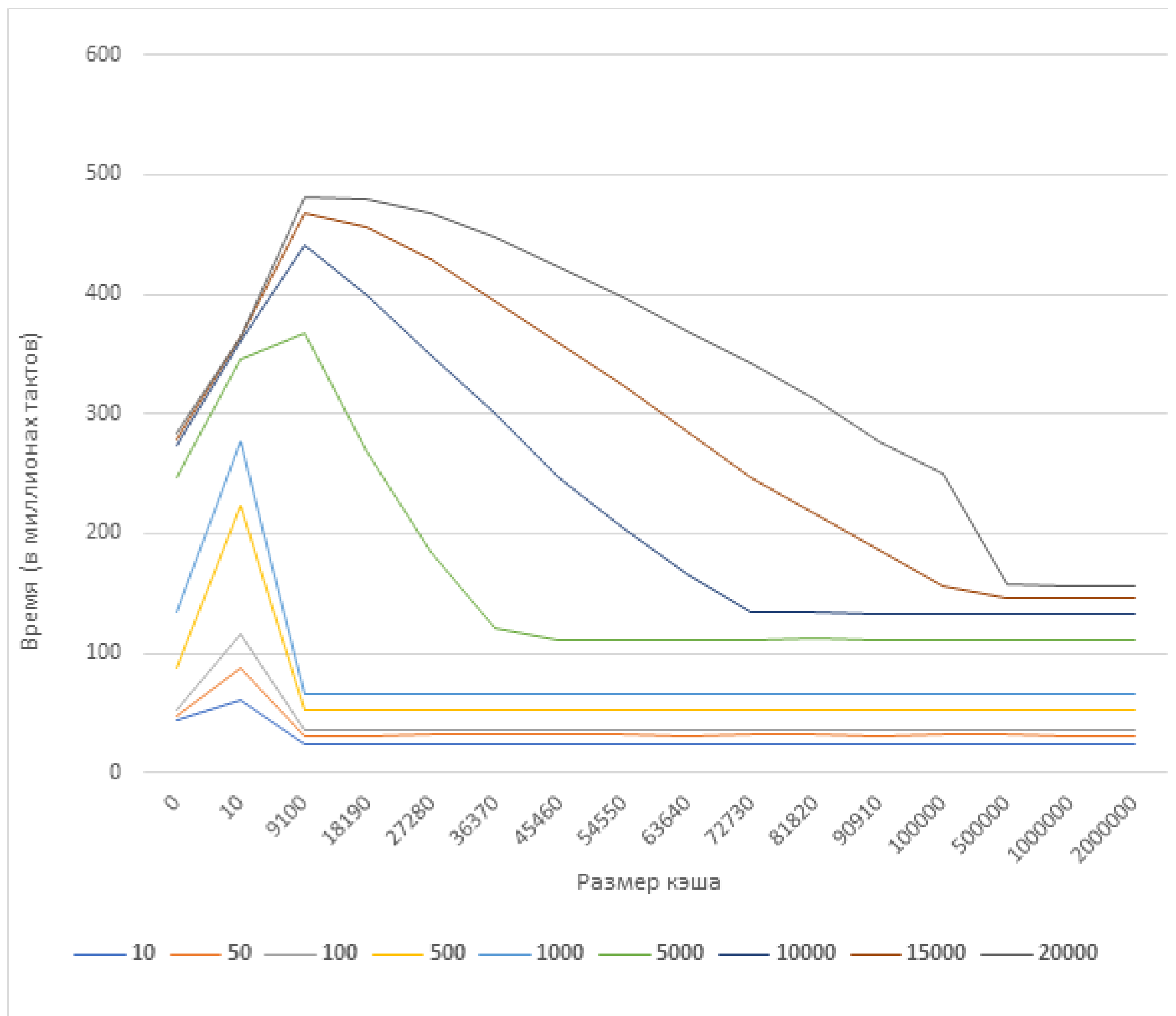

Рис. 4.3. Зависимость времени выполнения поиска в контейнере от размера кэша для различных σ при *k=20*. Различными цветами изображены различные значения σ

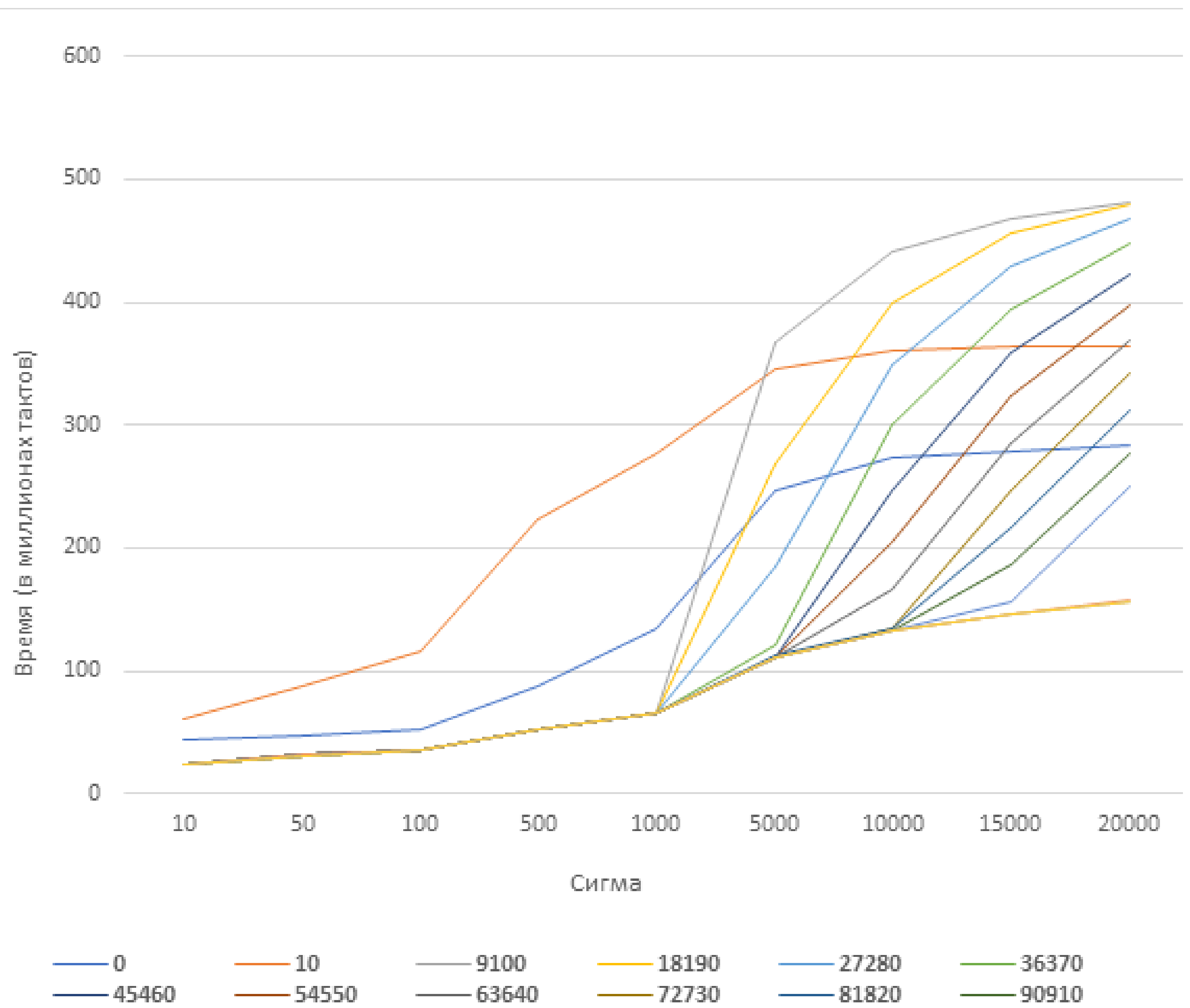

Рис. 4.4. Зависимость времени выполнения поиска в контейнере от σ для различных размеров кэша при *k=20*. Различными цветами изображены различные размеры кэша

Таблица 4.2 – Время поиска в зависимости от σ и размера кэша для k=20

	σ								
Размер кеша	**10**	**50**	**100**	**500**	**1000**	**5000**	**10000**	**15000**	**20000**
0	44035099	46869151	51951467	87392480	134671909	246267041	273713854	278720729	283229134
10	61378441	88280668	115161873	224040782	276603584	345967157	359823183	363293504	364105260
9100	23853405	31312333	35246751	52067457	65825853	367023041	441143427	468451518	481804017
18190	23848658	31348402	35310551	52030195	65582426	268362245	399521701	455330614	479776381
27280	23931456	31391515	35238668	51913748	65941924	184213669	349734798	428521244	468664433
36370	23860900	31379294	35185284	51791098	65633320	120602442	300075423	394017569	448067855
45460	23838958	31404270	35204942	51752677	65486242	111741472	246978712	359526265	422497948
54550	23887135	31386604	35283781	52119815	65607091	111159454	204013413	324071662	397874795

63640	23854226	31359105	35242288	51700179	65836834	111223692	166663594	284596358	369389281
72730	23864142	31399760	35219158	51868719	65421600	111099971	133618742	246133997	342868562
81820	23939240	31441928	35197994	51879990	65371615	112033657	133690887	215972201	312121471
90910	23832615	31325819	35274286	51743793	65314687	111181060	133029939	186473018	277001355
100000	23851499	31410873	35265944	51819655	65451104	110867690	133388158	156539510	249907663
500000	23880920	31411112	35216670	51945440	65872213	111491062	132668858	146516036	157480155

Однако не всегда есть возможность создать кэш необходимого размера. В этом случае оптимальным решением может оказаться не максимально доступный размер кэша, а отказ от его использования. Это связано с тем, что затраты на поиск в кэше могут быть больше, чем прирост производительности от его использования. На рисунке 4.5 изображена зависимость времени поиска от размера кэша для различных σ, где наглядно виден рост времени поиска при использовании недостаточного кэша, и плавное уменьшение при увеличении доступного размера кэша, что подтверждает ***лемму 2*** из раздела 2.8. На основании проведенных экспериментов можно выделить 2 случая оптимального размера кэша:

1. Если доступный размер кэша больше определенного значения (но меньше 8σ) (далее M_0), то оптимальный размер кэша равен максимально доступному значению (что подтверждает ***лемму 4*** из раздела 2.8);
2. При доступном размере кэша меньше M_0 – оптимальный размер кэша равен 0. Данные эксперименты подтверждают ***следствие 1*** из раздела 2.8.

При этом M_0 аппроксимировано может быть вычислено для определенного k и σ (или близких значений из таблиц) по формуле: $x_{опт} = \frac{(y_0 - y_1 + \frac{(y2-y_1)}{(x_2-x_1)} * x_1)}{\frac{(y2-y_1)}{(x_2-x_1)}}$,

где y_0 – время в тактах при нулевом размере кэша, y_1, y_2 ($y_2 > y_0 > y_1$) – время в тактах на концах отрезка (x_1; x_2), где x_1 и x_2 – размеры кэша, следующие друг за другом в таблице.

Рис. 4.5. Зависимость времени выполнения поиска в контейнере от размера кэша для различных σ при *k=20*. Различными цветами изображены различные значения σ

3. 14> ***k***> 8. При данном соотношении скоростей оптимальным может оказаться как решение из пункта 1, так и из пункта 2 (в зависимости от k и σ, таблица 4.3). На графике рисунка 4.6 стрелками указано, что оптимальным размером кэша (минимальным значением времени поиска) может быть как нулевой размер, так и максимально доступный размер кэша [7].

   В ходе эксперимента для данных соотношений скоростей хранилищ было выявлено, что зависимость оптимального размера кэша удовлетворяет условиям ***леммы 4***. При этом $K_d = \{\forall k \in R: 8 < k < 14\}$, а значение σ_0 экспериментально было найдено $\sigma_0 \approx 750$.

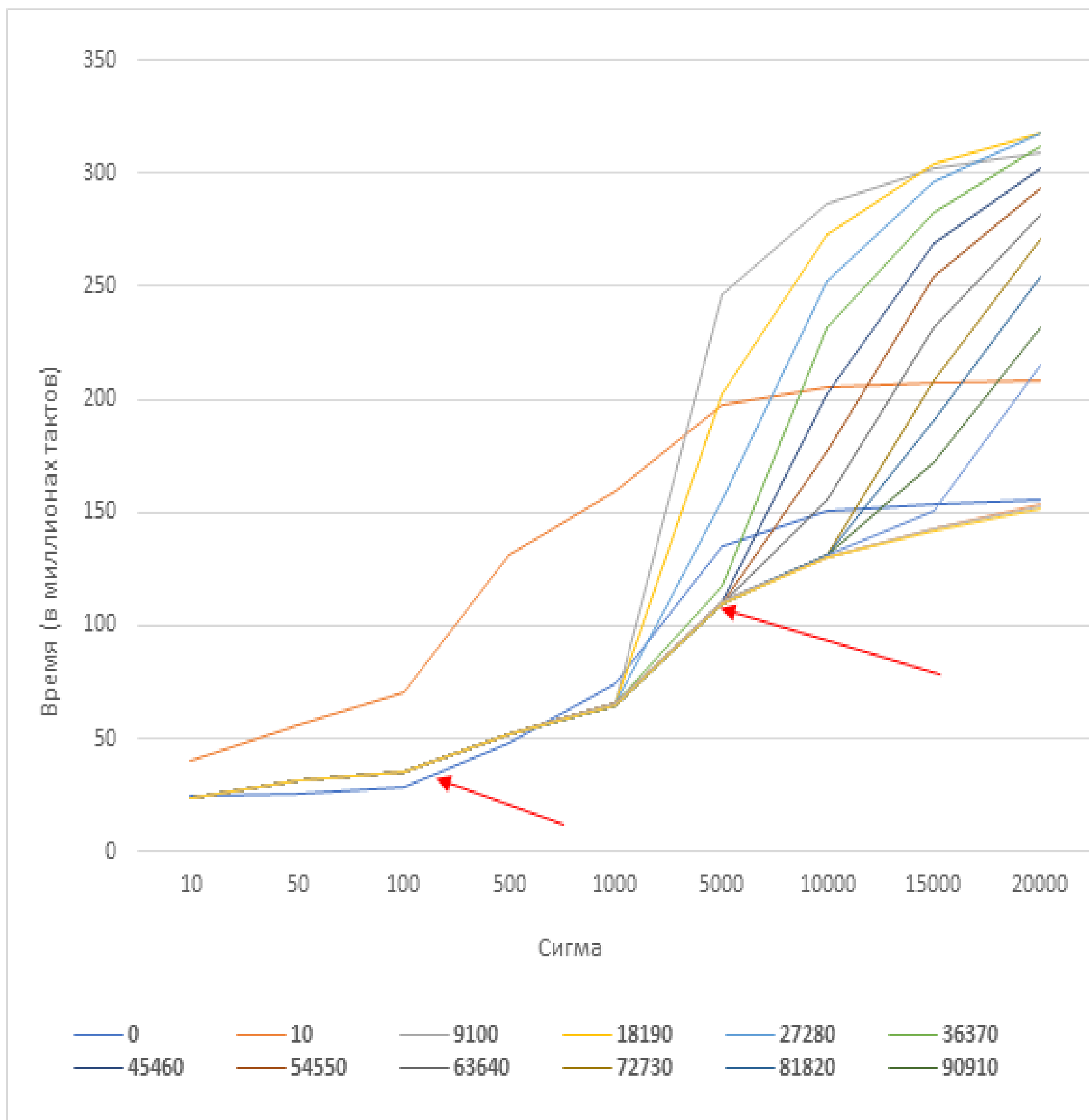

Рис. 4.6. Зависимость времени выполнения поиска в контейнере от размера кэша для различных σ при $k=11$. Различными цветами изображены различные размеры кэша

Таблица 4.3 – Время поиска в зависимости от σ и размера кэша для *k=11*

	σ								
Размер кеша	**10**	**50**	**100**	**500**	**1000**	**5000**	**10000**	**15000**	**20000**
0	24219304	25778033	28573307	48065864	74069550	135446873	150542620	153296401	155776024
10	40715800	55646735	70566331	130715703	159764115	197968750	205629705	207542175	207949433
9100	23848631	31291649	35208339	51906824	65535022	246439369	286892083	302041989	309411519
18190	23843911	31327938	35272368	51870769	65290504	202479750	273326639	304475316	317848572
27280	23926716	31370891	35200634	51755047	65652218	155692013	252418364	295946648	317952364
36370	23856047	31358720	35147793	51631936	65343835	117530201	232095073	282735051	312347063
45460	23834057	31383792	35166705	51593557	65194889	110518183	202535567	268942458	302640187
54550	23882322	31366279	35245185	51958209	65316321	109937952	177559674	254221401	293824440
63640	23849499	31338844	35204092	51539512	65541526	110006587	155313788	232238420	281838239
72730	23859366	31379083	35181378	51709031	65130618	109881125	131380286	208485614	271102214
81820	23934427	31421546	35160049	51719615	65083111	110802856	131442673	190555348	254418455
90910	23827730	31305662	35236590	51583809	65024358	109956834	130786853	172152167	231856493
100000	23846627	31390410	35227431	51659985	65159935	109641760	131132399	151137771	215042408
500000	23876129	31391065	35179516	51785255	65579630	110265175	130431177	143314019	153322489
1000000	23843821	31353546	35104710	51682270	65259442	110281483	130629389	142494953	152557330
2000000	23839726	31341438	35184410	51954870	65157271	109614881	130472266	142340488	151810443

4.2. Исследование эффективности модуля определения параметров сложной нагрузки на самоадаптирующиеся контейнеры данных

4.2.1. Описание условий тестирования

Алгоритм модуля определения параметров сложной нагрузки на самоадаптирующиеся контейнеры данных из раздела 3.2 был реализован на языке JavaScript с использованием библиотеки gaussian-mixture-model [146], которая реализует модификацию алгоритма EM на потоке с использованием буфера. Кроме того, данная библиотека позволяет учитывать уменьшение актуальности данных с течением времени.

Для тестирования подпрограммы был реализован алгоритм генерации смеси нормальных гауссовских распределений и средства визуализации кластеризации в одномерном пространстве. Для генерации одного распределения используется центральная предельная теорема [147]. Для вычисления времени работы модуля использовалось Web Performance API языка JavaScript.

Тестирование модуля проводилось на следующей программно-аппаратной платформе:

1. Processor: Intel Core i7 6500U @ 2.50 GHz (2 cores).
2. RAM: 16 GB.
3. OS: Windows 10.
4. Веб-браузеры Google Chrome 68.0.3440.84 и Opera 54.0.2952.64 [8].

4.2.2. Эксперименты

В первую очередь было проведено тестирование на одном кластере. Данная ситуация часто возникает в приложении, когда долгое время данные используются только для одной задачи или одним источником запросов. Например, это различные аналитические системы с задачами, требующими много данных и времени выполнения, работающие в режиме реального времени. Размер буфера

был выбран $1.5*10^6$ элементов. Была сгенерирована выборка в количестве 10^6 ключей по закону нормального распределения N (-25, 5.5) и в потоковом режиме ключи поступали в модуль. На рисунке 4.7 изображены ключи и границы кластера, построенного модулем. На изображении кластер обозначен как окружность с радиусом 4σ, т.к. в интервал $(\underline{x} - 4\sigma; \underline{x} + 4\sigma)$ попадает 99.993669986724854% всех ключей, принадлежащих кластеру [148], что является достаточным при таком количестве ключей в выборке.

Было проведено 10^6 таких тестов. В каждом были вычислены среднеквадратичное отклонение ключей и матожидание для полученной выборки. Параметры распределения, полученные в результате обработки данных ключей модулем кластеризации, совпали с вычисленными, что говорит о том, что алгоритм вычисления одного кластера работает верно.

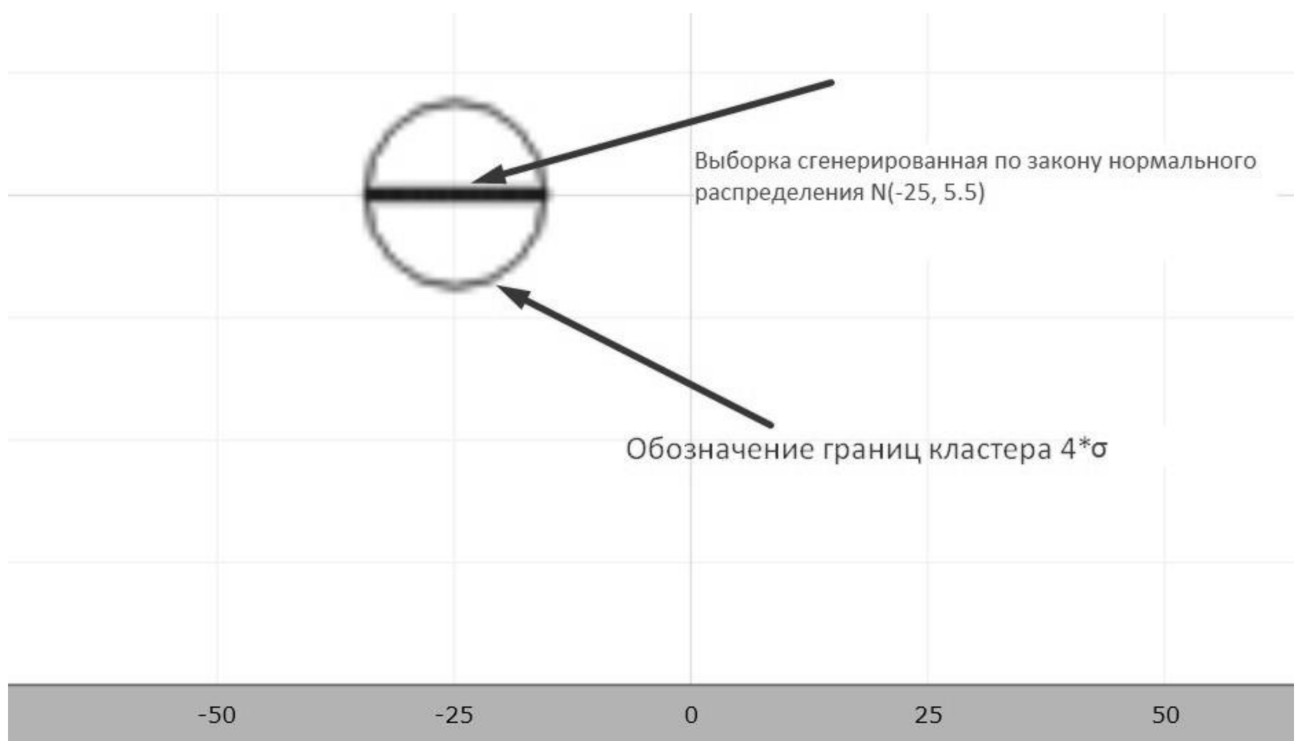

Рис. 4.7. Первичная выборка и кластер

Далее был протестирован случай, когда параметры распределения данных изменяются динамически. Для этого для заданного кластера в модуль было добавлено $5*10^5$ ключей в диапазоне [-51; -49] таким образом, что закон

распределения кластера поменялся на N (-33, 142). На рисунке 4.8 изображены ключи и границы нового кластера, построенного модулем.

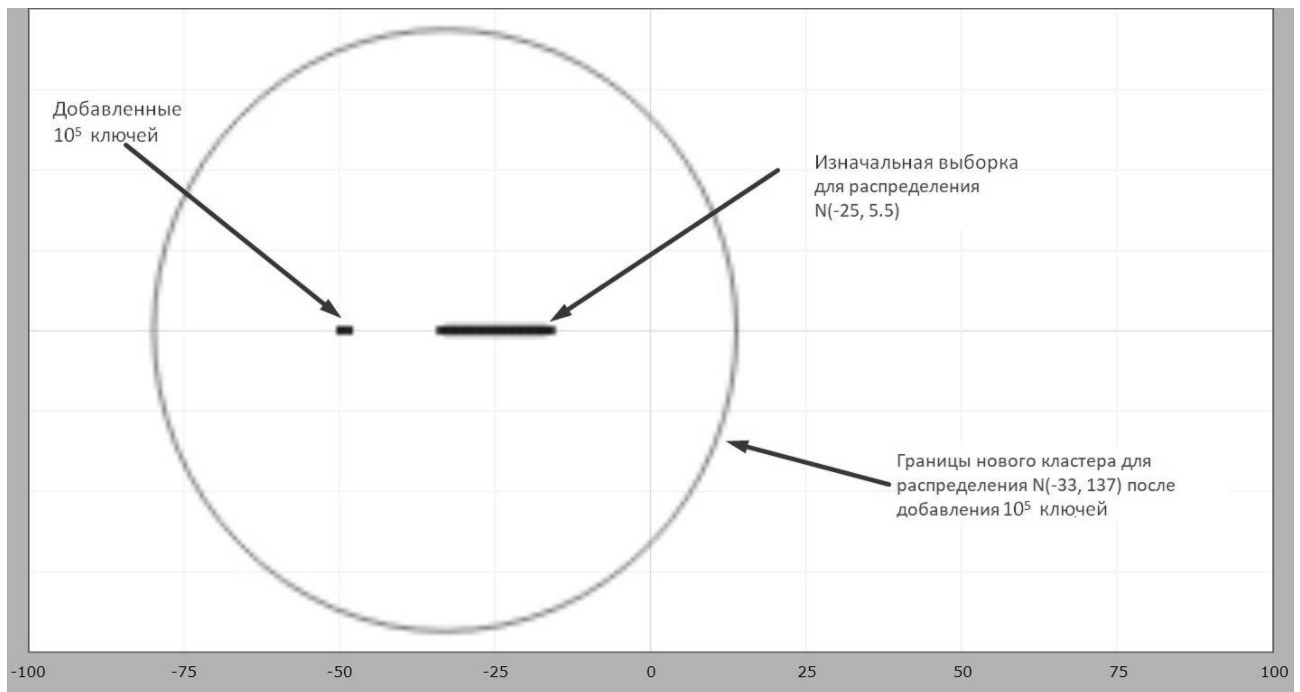

Рис. 4.8. Начальная выборка, добавленные $5*10^5$ ключи и обновленный кластер

После этого была исследована ситуация с устареванием данных. Такая ситуация так же часто встречается в реальных приложениях при смене источников запросов или смене задач, требующих большего количества данных и времени выполнения. Для этого было добавлено еще 10^6 ключей из диапазона [-51; -49], что привело к тому, что модуль перестал учитывать данные из первоначального распределения N (-25, 5.5) и построил кластер N (-50, 0.5). На рисунке 4.9 наглядно изображен результат работы модуля после добавления нового набора данных.

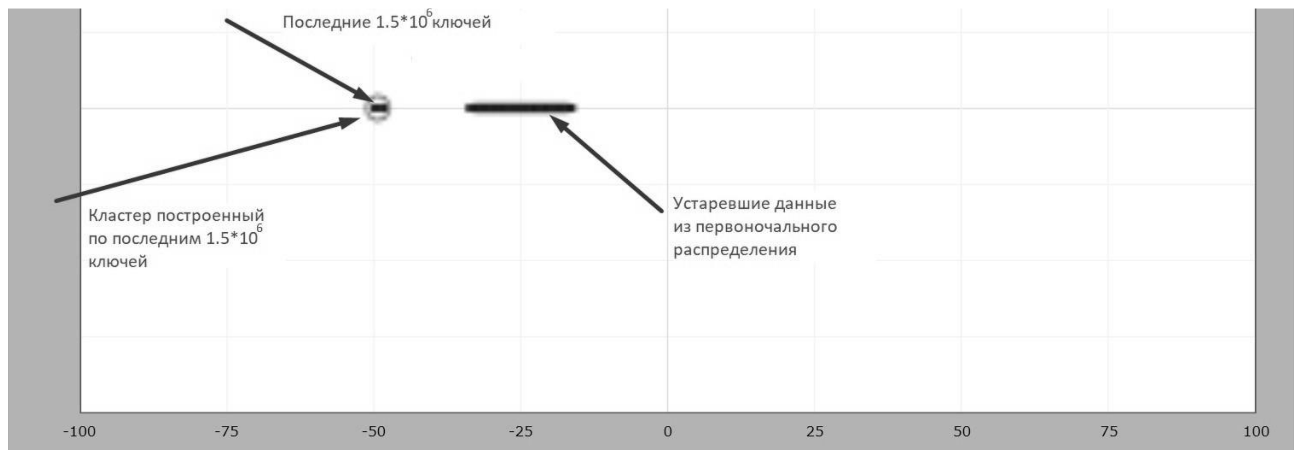

Рис. 4.9. Кластер после добавления $1.5*10^6$ ключей и изначальная (неактуальная) выборка

Далее была исследована кластеризация нескольких кластеров. Данная ситуация наиболее часто встречается в современных приложениях с большим числом задач и источников запросов, например, веб-приложениях, высоконагруженных системах. Для этого было сгенерировано 5 выборок по законам нормального распределения N (-80, 2.35), N (-40, 2.35), N (0, 2.35), N (40, 2.35), N (80, 2.35) в количестве 10^7 ключей каждая и в потоковом режиме ключи поступали в модуль. Размер буфера был выбран 10^8 элементов. В первую очередь была протестирована реализация модуля со случайной инициализацией. Было проведено 10^6 таких тестов. Однако при использовании такого подхода в большей части тестов кластеризация была проведена некорректно (успешно проведено было только 16% тестов). Примеры результатов такой кластеризации изображены на рисунке 4.10 (только на последнем графике изображена правильная кластеризация).

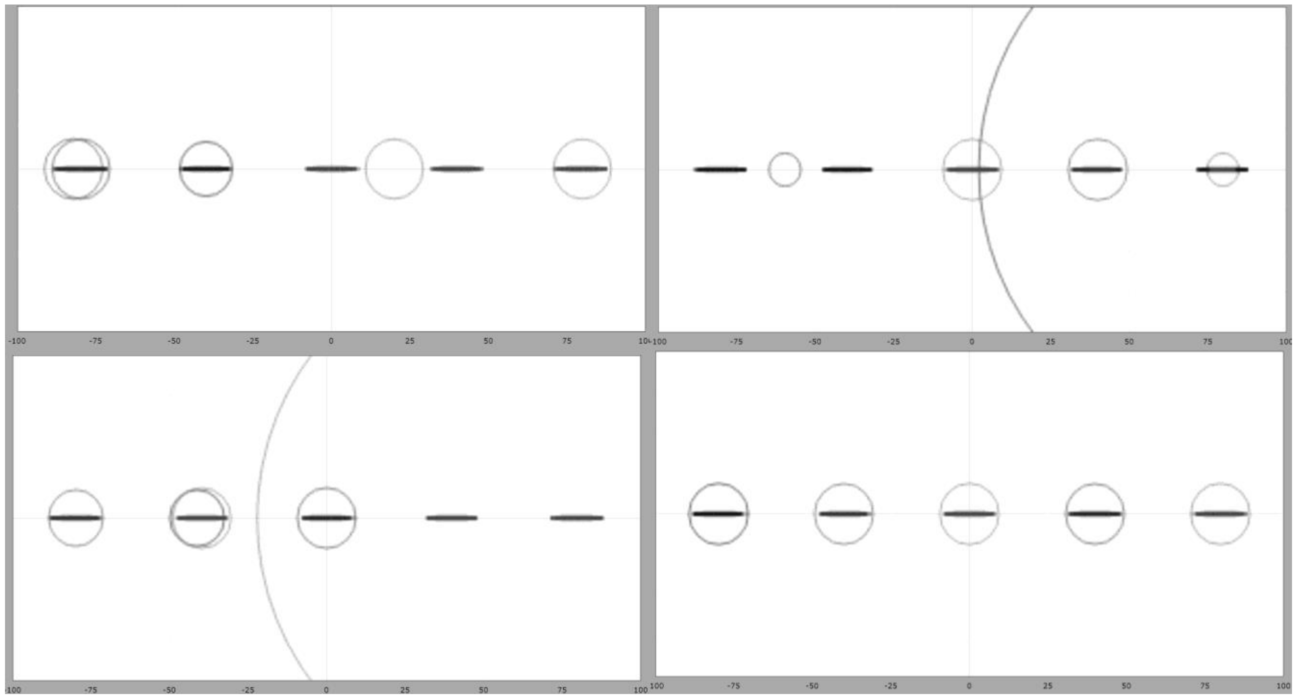

Рис. 4.10. Результаты кластеризации при случайной инициализации модуля

Далее на тех же данных модуль был протестирован с использованием kmeans++ инициализации. В результате для заданной смеси распределений кластеризация во всех 10^6 тестах была произведена достаточно корректно (доля ошибочных кластеризаций составила 4%). Кроме того, было протестировано пересечение кластеров (для смеси N (-50,22.5), N (-20, 22.5), N (0,5.55), N (20,12.3), рис. 4.11), проверены ситуации со смещением кластеров и устареванием данных для нескольких кластеров [8].

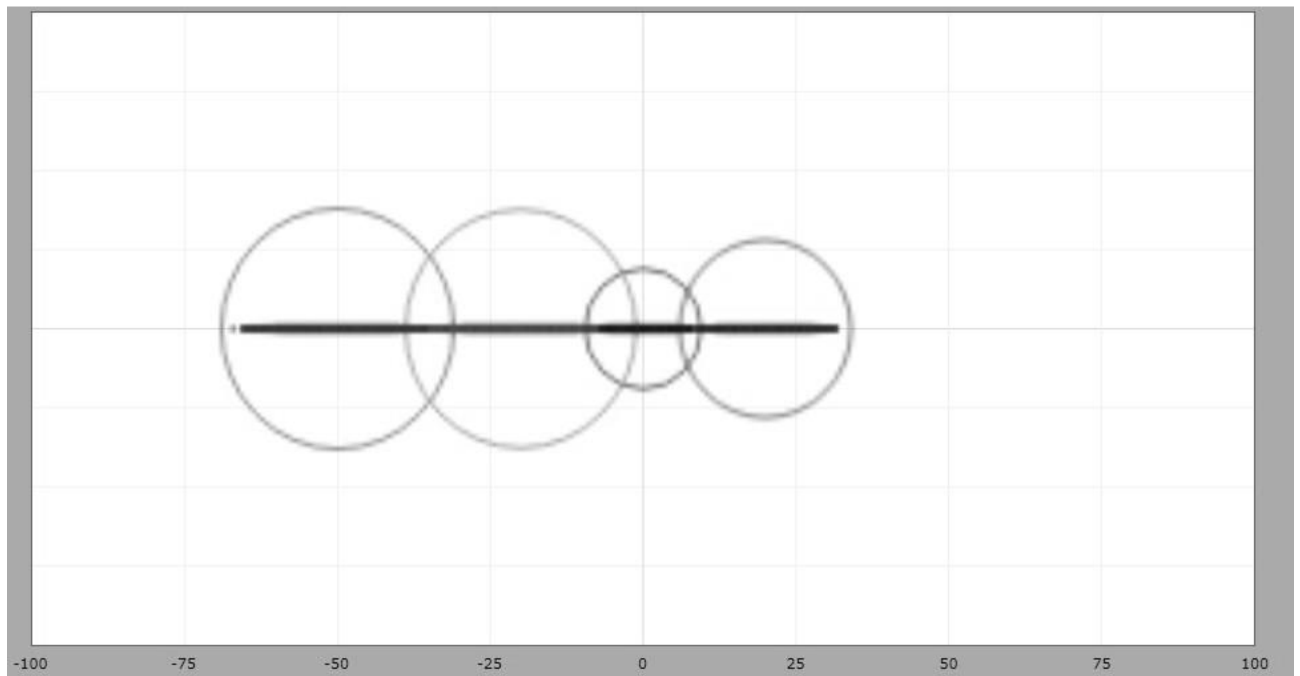

Рис. 4.11. Результаты кластеризации при пересечении кластеров

4.2.3. Анализ времени выполнения

Время выполнения EM кластеризации зависит линейно от количества итераций i, количества кластеров k и количества ключей n в анализируемом буфере (сложность O(i*k*n)). Данная закономерность была подтверждена тестами, результат которых представлен в таблицах 4.4-4.6. Исходя из этих данных можно прогнозировать скорость работы в зависимости от количества данных и кластеров.

Таблица 4.4 – Зависимость времени кластеризации от количества кластеров

Количество итераций и ключей в буфере	Количество кластеров	Среднее время кластеризации (в миллисекундах)
i=2 n=6*10^4	k=1	527
	k=2	872
	k=3	1253
	k=4	1567
	k=5	1891

Таблица 4.5 – Зависимость времени кластеризации от количества ключей в буфере

Количество итераций и кластеров	Количество ключей в буфере	Среднее время кластеризации (в миллисекундах)
i=2 k=4	n=6*10^4	1567
	n=12*10^4	3043
	n=24*10^4	6244
	n=48*10^4	13395
	n=96*10^4	27129

Таблица 4.6 –Зависимость времени кластеризации от количества итераций

Количество кластеров и ключей в буфере	Количество итераций	Среднее время кластеризации (в миллисекундах)
$n=6*10^4$ k=4	i=2	1567
	i=20	12566
	i=200	145004

Необходимо отметить, что кластеризация даже порядка 10^6 ключей для 4 кластеров занимает около 30 секунд (аналогичные результаты для алгоритма EM с буферизацией были получены в исследовании [149]). Несмотря на довольно долгое выполнение алгоритма EM, эффективность модуля определения параметров сложной нагрузки даже в условиях большой нагрузки будет высокой. Это связано в первую очередь с тем, что выполнение кластеризации не должно происходить слишком часто. Потому что изменение кэша в самоадаптирующемся контейнере на основе полученных от модуля параметров не должно происходить слишком часто, так как перестроение кэша для большого объема данных требует много ресурсов. Кроме того, производительность модуля может быть улучшена несколькими способами:

1. Увеличением размера буфера.
2. Созданием временного буфера для поступающих во время кластеризации ключей [8].

4.3. Исследование эффективности адаптивного кэширующего контейнера данных с использованием интервального статистического ряда

4.3.1. Описание условий тестирования

Алгоритм адаптивного кэширующего контейнера данных с использованием интервального статистического ряда, описанный в разделе 3.3 был реализован на языке C#.

Для тестирования данного контейнера была использована модифицированная версия консольной программы из раздела 4.1.1.

В процессе исследования был проведен ряд тестов, параметры которых заданы следующим образом:

1. Элементом кэша и основного хранилища является пара «ключ-значение» (<long, long>), где ключом является число типа *long* языка C#, значением выступает случайное число типа *long* языка C#;
2. Основное хранилище содержит 10^5 элементов, которые генерируется случайным образом.
3. В ходе эксперимента основное хранилище эмулировалось с помощью ассоциативной структуры данных языка C# Dictionary<long, long>, расположенной в оперативной памяти.
4. Кэш реализован с помощью ассоциативной структуры данных языка C# Dictionary<long, long>.
5. Для каждого теста подсчитывается процент попаданий в кэш (hit rate), т.к. это наиболее распространённый метод оценки алгоритмов кэширования [7].

Один тест представляет собой комбинацию из нескольких параметров:

1. Размер кэша M. Рассматривались значения $M \in \{100, 500, 1000, 2000, 5000, 10000, 15000\}$ элементов в кэше [7].
2. Используемый алгоритм вытеснения $A \in \{LRU, MRU, ARC, LIRS, AH\}$, где LRU (Least Recently Used) [103], MRU (Most Recently Used) [113], ARC

(Adaptive replacement cache) [109], LIRS [114], AH (реализованный автором алгоритм кэширования с использованием интервального статистического ряда).

3. Трасса, представляющая собой набор запросов на получение элементов из контейнера с ключами типа *long* языка C#. Были протестированы следующие трассы, по которым зачастую оценивается эффективность алгоритмов кэширования:
   a. Трасса из 10^7 запросов, ключи для которых сгенерированы по закону нормального распределения $N(0, 10000^2)$.
   b. Трасса из 10^7 запросов, ключи для которых сгенерированы по закону нормального распределения $N(0, 50000^2)$.
   c. OLTP – трасса из данных полученных от Online Transaction Processing системы [108] [109] [110]. Данная трасса состоит из запросов, отправленных в течении часа к базе данных Codasyl. Содержит 914145 запросов с 186880 уникальными ключами.
   d. Cpp [114] [119] – трасса из логов компилятора препроцессора GNU C (cpp). Получена в результате компиляции исходного кода на языке C в размере порядка 11Мб.
   e. Glimpse [114] [119] – трасса из логов утилиты поиска текстовой информации. Получена в результате работы программы с текстовыми файлами общим объемом порядка 50Мб.
   f. Web12 (trace-mt-20121220) [150] – трасса, сформированная из запросов, происходивших в декабре 2012 года к веб-страницам, содержащим описание некоторых продуктов в интернет-магазине. Каждому уникальному веб-адресу сопоставлено некоторое число. Содержит 95607 запросов с 13765 уникальными ключами.
   g. Web07 (trace-mt-20130703) [150] – трасса, сформированная из запросов, происходивших в июле 2013 года к веб-страницам,

содержащим описание некоторых продуктов в интернет-магазине. Каждому уникальному веб-адресу сопоставлено некоторое число.

h. OrmAccessBusy (trace-mt-db-20160419-busy) [150] – трасса, сформированная из запросов Java-приложения к ORM фреймворку. Трасса была записана во время дня, когда приложение работало активно. Содержит 5000000 запросов с 76349 уникальными адресами.
i. OrmAccessNight (trace-mt-db-20160419-night) [150] – трасса, сформированная из запросов Java-приложения к ORM фреймворку. Трасса была записана ночью, когда приложение не работало активно.
j. Multi2 [114] [119] – трасса, сформированная из логов работы трех приложений – компилятора (cpp трасса), утилиты интерактивной проверки исходного кода языка C с исходным кодом объема 9Мб и логов запросов соединения четырех таблиц в реляционной базе данных Калифорнийского Университета.
k. Sprite [114] [119] – трасса, полученная из данных работы сетевой файловой системы Sprite. Содержит запросы от клиентских рабочих станций к файл-серверу за двухдневный период.
l. P1-P7, P12 [151] – трассы, собранные с рабочих машин в течении нескольких месяцев под управлением ОС Windows NT с использованием программы Vtrace, которая записывает историю операций ввода-вывода при работе с жестким диском.
m. DS1 [151] – трасса, полученная из логов работы сервера базы данных, работавшего на коммерческом сайте, на котором выполнялось приложение EPR, запущенное на коммерческой базе данных. Содержит 43704979 запросов с 10516352 уникальными адресами.

4.3.2. Вычислительный эксперимент

В рамках численного эксперимента по его результатам (см. таблицы 4.7-4.24) получены следующие научно-прикладные результаты:

1. Для многих трасс ($N(0, 10000^2)$, $N(0, 50000^2)$) для всех размеров кэша алгоритм кэширующего контейнера с использованием интервального статистического ряда заметно превосходит другие алгоритмы кэширования. Это связано с тем, что в данных трассах часто используемые элементы находятся достаточно близко друг к другу, а загрузка элементов в кэш осуществляется блоками, что позволяет иметь в кэше необходимые элементы еще до первого запроса к ним и не требует поэлементной загрузки.
2. Для некоторых трасс (CPP, Glimps, Web12, Web07, Multi2, Sprite, DS1, P1, P2, P3, P5) при некоторых размерах кэша алгоритм кэширующего контейнера с использованием интервального статистического ряда уступает другим алгоритмам кэширования, а при других размерах кэша превосходит их. В первую очередь это связано с тем, что алгоритм автора загружает блоками, которые содержат наиболее используемые элементы, однако для данных трасс блок также может содержать элементы, которые используются редко или к которым вообще не было и не будет обращений. Следовательно, при определенных размерах кэша и блока, в кэше оказывается слишком большое количество ненужных элементов, однако при других размерах кэша такой подход начинает себя оправдывать. Данный недостаток алгоритма может быть устранен правильным размером блока в кэширующем контейнере с использованием интервального статистического ряда.
3. Для некоторых трасс (OLTP, OrmAccessBusy, OrmAccessNight, P4, P12) при всех исследованных размерах кэша алгоритм кэширующего контейнера с использованием интервального статистического ряда уступает другим алгоритмам кэширования. Это связано с тем, что часто запрашиваемые ключи расположены в большом количестве разных блоках и поэлементные стратегии кэширования оказываются эффективнее. Для некоторых трасс уменьшение размера блока может улучшить процент попадания в кэш, однако может существенно

увеличить время поиска, т.к. значительно увеличится количество блоков.

Таблица 4.7 – Процент попаданий в кэш в зависимости от алгоритма и размера кэша для трассы N(0,10000^2)

Алгоритм	Размер кэша						
	100	500	1000	2000	5000	10000	15000
AH	0.38%	1.98%	3.89%	7.75%	19.47%	38.18%	54.57%
ARC	0.30%	1.55%	3.08%	6.27%	15.87%	30.30%	42.76%
LRU	0.29%	1.42%	2.84%	5.64%	13.94%	27.40%	40.22%
MRU	0.19%	0.95%	1.89%	3.76%	9.37%	18.79%	28.16%
LIRS	0.33%	1.65%	3.26%	6.47%	15.71%	30.53%	44.54%

Таблица 4.8 – Процент попаданий в кэш в зависимости от алгоритма и размера кэша для трассы N(0,50000^2)

Алгоритм	Размер кэша						
	100	500	1000	2000	5000	10000	15000
AH	0.09%	0.43%	0.86%	1.73%	4.00%	7.96%	11.92%
ARC	0.07%	0.32%	0.62%	1.27%	3.23%	6.27%	9.31%
LRU	0.05%	0.27%	0.56%	1.11%	2.79%	5.58%	8.33%
MRU	0.05%	0.23%	0.46%	0.93%	2.29%	4.52%	6.75%
LIRS	0.06%	0.33%	0.65%	1.30%	3.21%	6.37%	9.43%

Таблица 4.9 – Процент попаданий в кэш в зависимости от алгоритма и размера кэша для трассы OLTP

Алгоритм	Размер кэша						
	100	500	1000	2000	5000	10000	15000
AH	0.52%	1.06%	1.45%	2.57%	4.90%	9.00%	12.63%
ARC	10.29%	30.61%	39.06%	46.17%	55.12%	61.92%	65.58%
LRU	8.28%	23.45%	32.83%	42.47%	53.65%	60.70%	64.63%
MRU	0.05%	0.24%	0.46%	0.90%	2.17%	4.56%	6.85%
LIRS	9.36%	26.87%	34.82%	42.52%	52.80%	60.35%	63.99%

Таблица 4.10 – Процент попаданий в кэш в зависимости от алгоритма и размера кэша для трассы CPP

Алгоритм	Размер кэша						
	100	500	1000	2000	5000	10000	15000

AH	23.85%	81.80%	93.40%	97.23%	97.23%	97.23%	97.23%
ARC	76.96%	85.83%	86.42%	86.48%	86.48%	86.48%	86.48%
LRU	69.71%	84.78%	86.40%	86.48%	86.48%	86.48%	86.48%
MRU	21.72%	70.08%	82.03%	86.48%	86.48%	86.48%	86.48%
LIRS	77.63%	85.91%	86.43%	86.48%	86.48%	86.48%	86.48%

Таблица 4.11 – Процент попаданий в кэш в зависимости от алгоритма и размера кэша для трассы Glimps

Алгоритм	Размер кэша						
	100	500	1000	2000	5000	10000	15000
AH	2.89%	9.11%	32.55%	73.13%	95.68%	95.68%	95.68%
ARC	1.38%	1.38%	21.30%	57.41%	57.96%	57.96%	57.96%
LRU	0.91%	0.95%	11.21%	57.41%	57.96%	57.96%	57.96%
MRU	5.77%	32.37%	49.79%	57.96%	57.96%	57.96%	57.96%
LIRS	6.07%	33.60%	50.56%	57.96%	57.96%	57.96%	57.96%

Таблица 4.12 – Процент попаданий в кэш в зависимости от алгоритма и размера кэша для трассы Web12

Алгоритм	Размер кэша						
	100	500	1000	2000	5000	10000	15000
AH	2.89%	9.11%	32.55%	73.13%	95.68%	95.68%	95.68%
ARC	1.38%	1.38%	21.30%	57.41%	57.96%	57.96%	57.96%
LRU	0.91%	0.95%	11.21%	57.41%	57.96%	57.96%	57.96%
MRU	5.77%	32.37%	49.79%	57.96%	57.96%	57.96%	57.96%
LIRS	6.07%	33.60%	50.56%	57.96%	57.96%	57.96%	57.96%

Таблица 4.13 – Процент попаданий в кэш в зависимости от алгоритма и размера кэша для трассы Web07

Алгоритм	Размер кэша						
	100	500	1000	2000	5000	10000	15000
AH	2.39%	8.52%	13.17%	21.16%	38.67%	61.56%	80.67%
ARC	36.81%	48.30%	53.08%	57.86%	64.32%	69.65%	72.13%
LRU	33.40%	45.58%	50.41%	55.50%	62.67%	69.00%	72.00%
MRU	7.33%	8.82%	10.68%	13.98%	22.20%	39.99%	58.25%
LIRS	32.13%	46.83%	52.31%	57.22%	63.49%	68.80%	71.87%

Таблица 4.14 – Процент попаданий в кэш в зависимости от алгоритма и размера кэша для трассы OrmAccessBusy

Алгоритм	Размер кэша						
	100	500	1000	2000	5000	10000	15000
AH	3.10%	5.94%	16.86%	23.18%	31.40%	41.12%	45.96%
ARC	51.51%	69.87%	78.53%	81.54%	85.68%	89.68%	92.07%
LRU	51.48%	69.28%	78.78%	81.39%	85.46%	89.49%	91.97%
MRU	4.91%	5.64%	6.39%	12.94%	19.60%	32.02%	39.48%
LIRS	37.50%	66.42%	73.69%	78.40%	85.18%	89.56%	92.06%

Таблица 4.15 – Процент попаданий в кэш в зависимости от алгоритма и размера кэша для трассы OrmAccessNight

Алгоритм	Размер кэша						
	100	500	1000	2000	5000	10000	15000
AH	0.68%	2.04%	5.28%	18.35%	24.61%	28.17%	32.66%
ARC	45.56%	63.97%	67.68%	71.08%	89.04%	94.13%	95.28%
LRU	45.18%	60.95%	67.32%	69.77%	87.85%	94.06%	95.28%
MRU	3.14%	3.62%	4.10%	5.01%	12.27%	38.44%	47.65%
LIRS	29.38%	55.06%	61.36%	70.85%	89.33%	93.99%	95.22%

Таблица 4.16 – Процент попаданий в кэш в зависимости от алгоритма и размера кэша для трассы Multi2

Алгоритм	Размер кэша						
	100	500	1000	2000	5000	10000	15000
AH	2.07%	11.47%	22.18%	43.02%	90.01%	98.72%	98.72%
ARC	25.89%	39.49%	50.75%	64.13%	78.38%	78.40%	78.40%
LRU	6.73%	35.98%	47.80%	49.00%	78.38%	78.40%	78.40%
MRU	1.84%	9.62%	19.11%	38.45%	73.03%	78.40%	78.40%
LIRS	29.82%	51.08%	58.15%	71.07%	78.40%	78.40%	78.40%

Таблица 4.17 – Процент попаданий в кэш в зависимости от алгоритма и размера кэша для трассы Sprite

Алгоритм	Размер кэша						
	100	500	1000	2000	5000	10000	15000
AH	3.99%	16.61%	29.16%	49.25%	85.39%	99.64%	99.64%
ARC	25.59%	77.38%	89.84%	93.33%	94.33%	94.72%	94.72%
LRU	21.58%	78.30%	90.64%	93.48%	94.65%	94.72%	94.72%
MRU	5.37%	12.06%	21.85%	36.92%	91.83%	94.72%	94.72%
LIRS	25.28%	75.96%	87.61%	91.20%	93.86%	94.72%	94.72%

Таблица 4.18 – Процент попаданий в кэш в зависимости от алгоритма и размера кэша для трассы DS1

Алгоритм	Размер кэша						
	100	500	1000	2000	5000	10000	15000
AH	0.05%	0.23%	0.39%	0.63%	1.24%	2.17%	2.62%
ARC	0.06%	0.34%	0.46%	0.55%	0.76%	0.95%	1.09%
LRU	0.05%	0.13%	0.40%	0.50%	0.72%	0.95%	1.12%
MRU	0.00%	0.00%	0.00%	0.00%	0.05%	0.20%	0.37%
LIRS	0.03%	0.15%	0.21%	0.26%	0.36%	0.50%	0.62%

Таблица 4.19 – Процент попаданий в кэш в зависимости от алгоритма и размера кэша для трассы P1

Алгоритм	Размер кэша						
	100	500	1000	2000	5000	10000	15000
AH	0.12%	0.43%	1.02%	2.38%	4.85%	9.04%	11.65%
ARC	0.15%	0.59%	0.90%	1.87%	6.07%	11.19%	15.60%
LRU	0.11%	0.28%	0.37%	0.48%	0.92%	2.91%	5.79%
MRU	0.03%	0.08%	0.13%	0.26%	0.77%	1.76%	3.39%
LIRS	0.11%	0.52%	1.03%	2.25%	6.14%	11.21%	15.49%

Таблица 4.20 – Процент попаданий в кэш в зависимости от алгоритма и размера кэша для трассы P2

Алгоритм	Размер кэша						
	100	500	1000	2000	5000	10000	15000
AH	0.32%	1.52%	2.45%	3.71%	6.68%	11.39%	12.39%
ARC	0.71%	2.01%	2.56%	3.20%	5.07%	10.75%	15.45%
LRU	0.67%	2.02%	2.57%	2.97%	3.65%	5.65%	8.82%
MRU	0.02%	0.15%	0.22%	0.30%	1.10%	2.55%	4.12%
LIRS	0.12%	0.33%	0.65%	1.38%	4.47%	9.99%	14.12%

Таблица 4.21 – Процент попаданий в кэш в зависимости от алгоритма и размера кэша для трассы P3

Алгоритм	Размер кэша						
	100	500	1000	2000	5000	10000	15000
AH	0.24%	1.06%	1.93%	3.33%	5.21%	9.80%	10.16%
ARC	0.32%	0.87%	1.12%	1.52%	2.85%	4.51%	6.37%
LRU	0.30%	0.81%	1.05%	1.15%	1.39%	1.73%	1.98%
MRU	0.02%	0.10%	0.23%	0.58%	1.55%	3.17%	4.79%
LIRS	0.07%	0.27%	0.59%	1.13%	2.80%	6.02%	9.17%

Таблица 4.22 – Процент попаданий в кэш в зависимости от алгоритма и размера кэша для трассы P4

Алгоритм	Размер кэша						
	100	500	1000	2000	5000	10000	15000
AH	0.13%	0.53%	0.75%	1.25%	2.00%	4.09%	4.71%
ARC	0.33%	2.25%	2.68%	2.97%	3.61%	4.53%	5.47%
LRU	0.31%	1.89%	2.67%	2.96%	3.40%	3.74%	4.01%
MRU	0.02%	0.08%	0.16%	0.30%	0.55%	1.20%	1.85%
LIRS	0.03%	0.08%	0.16%	0.44%	1.09%	2.40%	3.90%

Таблица 4.23 – Процент попаданий в кэш в зависимости от алгоритма и размера кэша для трассы P5

Алгоритм	Размер кэша						
	100	500	1000	2000	5000	10000	15000
AH	0.20%	0.93%	1.60%	2.70%	5.86%	7.64%	9.84%
ARC	0.51%	3.68%	4.18%	4.53%	4.97%	5.37%	5.57%
LRU	0.51%	3.68%	4.18%	4.53%	4.97%	5.37%	5.57%
MRU	0.14%	0.17%	0.23%	0.49%	1.30%	2.42%	3.59%
LIRS	0.93%	2.06%	2.15%	2.38%	3.15%	4.76%	6.72%

Таблица 4.24 – Процент попаданий в кэш в зависимости от алгоритма и размера кэша для трассы P12

Алгоритм	Размер кэша						
	100	500	1000	2000	5000	10000	15000
AH	0.26%	1.00%	1.67%	2.66%	3.92%	5.79%	6.33%
ARC	0.73%	3.51%	4.14%	4.85%	6.07%	7.86%	9.73%
LRU	0.67%	3.43%	4.06%	4.80%	5.79%	6.49%	6.90%
MRU	0.02%	0.10%	0.14%	0.19%	0.45%	2.25%	4.08%
LIRS	0.10%	0.24%	0.40%	0.81%	2.23%	4.70%	7.08%

4.4. Выводы

1. В результате экспериментов были доказаны утверждения из раздела 2.8 и найдены множества K_d, K_b, K_0, размер кэша M_0 и значение среднеквадратичного отклонения σ_0, для которых выполняются утверждения, что позволило получить зависимость размера кэша от среднеквадратичного отклонения ключей в трассе и соотношения скоростей кэша и основного хранилища. Полученные закономерности могут быть использованы для динамического изменения размера кэша в зависимости от параметров нагрузки при реализации самоадаптирующегося контейнера данных [7].
2. Разработан модуль определения параметров сложной нагрузки был протестирован для одного и нескольких кластеров, смещения и устаревания данных. Точность кластеризации модуля при использовании k-means++ составила 100% для одного кластера и 96% для нескольких. Также приведены результаты экспериментов реализации модуля со случайной инициализацией (точность кластеризации нескольких кластеров составила 16%). Кроме того, приведены результаты экспериментов по определению зависимости времени работы модуля от количества кластеров, количества итераций и количества ключей в буфере, которые позволяют прогнозировать время работы модуля. Исходя из результатов тестирования, можно сказать, что данный модуль хорошо справляется с задачей определения параметров сложной нагрузки и может быть эффективно использован в самоадаптирующихся контейнерах данных [8].
3. Реализован адаптивный кэширующий контейнер данных с использованием интервального статистического ряда и проведено сравнение с другими известными и эффективными алгоритмами кэширования – LRU, MRU, ARC, LIRS. Тестирование проводилось на различных искусственных трассах запросов и трассах, состоящих из реальных данных. В результате экспериментов было выявлено, что адаптивный кэширующий контейнер

данных с использованием интервального статистического ряда во многих ситуациях превосходит указанные выше алгоритмы по проценту попаданий в кэш, а, следовательно, является эффективным.

Заключение

1. Разработан метод адаптации ассоциативного контейнера данных.
2. Определены области применения одномерных и многомерных ассоциативных контейнеров данных и алгоритмов кэширования с целью использования в самоадаптирующемся контейнере данных.
3. При исследовании зависимости размера кэша от среднеквадратичного отклонения нормального распределения и соотношения скоростей хранилищ была получена формула оптимального по времени размера кэша.
4. Разработан и реализован алгоритм определения параметров сложной нагрузки на самоадаптирующиеся контейнеры данных, который позволил достигнуть 100% точности для одного кластера и 96% для нескольких кластеров [8].
5. Разработан и реализован алгоритм адаптивного кэширующего контейнера данных с использованием интервального статистического ряда. Проведено сравнение с другими известными и эффективными алгоритмами кэширования (LRU, MRU, ARC, LIRS). В результате экспериментов выявлено, что данный алгоритм во многих ситуациях превосходит данные алгоритмы по проценту попаданий в кэш.

Список литературы

1. Зобов В.В. Инструмент для моделирования нагрузки на контейнеры данных / В.В. Зобов, К.Е. Селезнев // Материалы четырнадцатой научно-методической конференции «Информатика: проблемы, методология,технологии». – Воронеж, 2014. – Т. 3. – С. 154–161.
2. Елисеев Д.В. Аппаратно-программные средства карманных компьютеров / Д.В. Елисеев. – Санкт-Петербург: БХВ-Петербург, 2003. – 368 с.
3. Потапов Д.Р. Обзор условий адаптации самоадаптирующихся ассоциативных контейнеров данных / Д.Р. Потапов, М.А. Артемов, Е.С. Барановский // Вестник ВГУ. Серия Системный анализ и информационные технологии. – Воронеж, 2017. – Т. 1. – С. 112-119.
4. Учебник по высоким нагрузкам / О. Бунин, М. Лапшин, К. Осипов, К. Машуков. – Москва: Издательство Олега Бунина, 2018. – 53 с.
5. Balakrishnan C. Fundamentals of NoSQL / C. Balakrishnan. – Riga: LAP LAMBERT Academic Publishing, 2014. – 204 pp.
6. White T. Hadoop: The Definitive Guide / T. White. – 4th ed. – New York: O'Reilly Media, 2015. – 756 pp.
7. Потапов Д. Р. Исследование эффективности применения кэша для использования в самоадаптирующихся контейнерах данных / Д. Р. Потапов // Информационные технологии. – 2019. – Т. 25, № 4. – С. 216-222.
8. Потапов Д.Р. Реализация модуля определения параметров сложной нагрузки на самоадаптирующиеся контейнеры данных / Д. Р. Потапов // ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ И ВЫЧИСЛИТЕЛЬНЫЕ СИСТЕМЫ. – 2019. – № 1. – С. 87-95.
9. Добрица В. П. АНАЛИЗ ПАРАМЕТРОВ СОРТИРОВКИ ИНФОРМАЦИИ / В. П. Добрица, А. А. Липунов, Е. С. Савенкова // ИЗВЕСТИЯ ЮГО-ЗАПАДНОГО ГОСУДАРСТВЕННОГО УНИВЕРСИТЕТА. СЕРИЯ:

УПРАВЛЕНИЕ, ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА, ИНФОРМАТИКА. МЕДИЦИНСКОЕ ПРИБОРОСТРОЕНИЕ. – 2012. – Т. 1, № 2. – С. 101-105.

10. Обзор методов построения контейнеров данных «ключ-значение» для использования в самоадаптирующихся контейнерах данных / Д. Р. Потапов, М. А. Артемов, Е. С. Барановский, К. Е. Селезнев // Кибернетика и программирование. – 2017. – № 5. – С. 14-45.
11. Шевелев С. С. СОРТИРОВКА ИНФОРМАЦИИ МЕТОДОМ ДЕШИФРАЦИИ ДАННЫХ / С. С. Шевелев, В. П. Добрица // ТЕЛЕКОММУНИКАЦИИ. – 2010. – № 1. – С. 40-45.
12. SSTable & LSM-Tree. – URL: http://www.mezhov.com/2013/09/sstable-lsm-tree.html (дата обращения: 15.10.2019).
13. Carpenter J. Cassandra: The Definitive Guide / J. Carpenter, E. Hewitt. – 2nd ed. – New York: O'Reilly Media, 2016. – 360 pp.
14. Обработка больших объемов данных на основе MapReduce / М. А. Артемов, А. Ш. Исламов, Е. С. Барановский, М. В. Киргинцев // Информатика: проблемы, методология, технологии. Материалы XV международной научно-методической конференции. – Воронеж, 2015. – С. 78-80.
15. Ахо А.В. Структуры данных и алгоритмы / А. В. Ахо, Д. Э. Хопкрофт, Д. Д. Ульман. – Москва: Вильямс, 2003. – 382 с.
16. Структуры данных: бинарные деревья. Часть 2: обзор сбалансированных деревьев. – URL: https://habr.com/ru/post/66926/ (дата обращения: 15.10.2019).
17. Топп У. Структуры данных в С++ / У. Топп, У. Форд. – Москва: Бином, 2000. – 815 с.
18. Седжвик Р. Фундаментальные алгоритмы на Java. Ч. 1 – 4: Анализ. Структуры данных. Сортировка. Поиск / Р. Седжвик. – Киев: ДиаСофт, 2002. – 688 с.
19. Кнут Д. Искусство программирования / Д. Кнут – 3е-е изд. – Москва: Вильямс, 2006. – 720 с.

20. Aragon C.R. Randomized Search Trees / C.R. Aragon, R. Seidel // Proc. 30th Symp. Foundations of Computer Science. – Washington, 1989. – pp. 540-545.
21. Sleator D.D. Self-Adjusting Binary Search Trees / D.D. Sleator, R.E. Tarjan // Journal of the ACM. – 1985. – No. 32. – pp. 652-686.
22. Andersson A. Improving partial rebuilding by using simple balance criteria / A. Andersson // Proc. Workshop on Algorithms and Data Structures. – Berlin, 1989. – pp. 393-402.
23. Левитин А.В. Алгоритмы. Введение в разработку и анализ / А.В. Левитин. – Москва: Вильямс, 2006. – 576 с.
24. Зубов В.С. Структуры и методы обработки данных. Практикум в среде Delphi / В.С. Зубов, И.В. Шевченко. – Москва: Филинъ, 2004. – 304 с.
25. Berliner H. The B* Tree Search Algorithm. A Best-First Proof Procedure / H. Berliner // Artificial Intelligence. – 1979. – No. 12. – pp. 23-40.
26. Bagwell P. Ideal Hash Trees / P. Bagwell. – Ecole polytechnique fédérale de Lausanne, Technical Report 2001-001. – Lausanne, 2001.
27. Heinz S. Burst Tries: A Fast, Efficient Data Structure for String Keys / S. Heinz, J. Zobel, H. Williams // ACM Transactions on Information Systems. – 2002. – Vol. 20. – No. 2. – pp. 192-223.
28. Arge L. The buffer tree: a new technique for optimal I/O-algorithms / L. Arge // Proc. Workshop on Algorithms and Data Structures. – Berlin, 1995. – pp. 334-345.
29. An Introduction to Bε-trees and Write-Optimization / M. Bender, M. Farach-Colton, W. Jannen [and etc.] // ;login: magazine. – 2015. – Vol. 40. – No. 5. – pp. 22-28.
30. On external memory graph traversal / A.L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, J.R. Westbrook // Proceedings of the Eleventh Annual ACMSIAM Symposium on Discrete Algorithms (SODA ’00). Society for Industrial and Applied Mathematics. – Philadelphia, 2000. – pp. 859-860.

31. The log-structured merge-tree (LSM-tree) / P.E. O’Neil, E. Cheng, D. Gawlick, E.J. O’Neil // Acta Informatica. – 1996. – Vol. 33. – No. 4. – pp. 351-385.
32. Sears R. bLSM: a general purpose log structured merge tree / R. Sears, R. Ramakrishnan // Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. – New York, 2012. – pp. 217–228.
33. Diff-Index: Differentiated Index in Distributed Log-Structured Data Stores / W. Tan, S. Tata, Y. Tang, L. Fong // Proceedings of the 17th International Conference on Extending Database Technology, EDBT. – Konstanz, 2014. – pp. 700–711.
34. Cache-Oblivious streaming B-trees / Bender M.A., Farach-Colton M., Fineman J. [and etc.] // Proceedings of the 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures. – San Diego, 2007. – pp. 81-92.
35. Кормен Т. Алгоритмы: построение и анализ / Т. Кормен. – 2е изд. – Москва: Вильямс, 2005. – 1296 с.
36. Алексеев В. Е. Графы и алгоритмы. Структуры данных. Модели вычислений / В. Е. Алексеев, В. А. Таланов. – Москва: Бином, 2006. – 320 с.
37. Fredman M. L. Fibonacci heaps and their uses in improved network optimization algorithms / M. L. Fredman, R. E. Tarjan // Journal of the Association for Computing Machinery. – 1987. – Vol. 34. – No. 3. – pp. 596-615.
38. Kaplan H. Thin heaps, thick heaps / H. Kaplan, R. E. Tarjan // ACM Transactions on Algorithms (TALG). – 2008. – Vol. 4. – pp. 1-14.
39. Mehta D. P. Handbook of Data Structures and Applications / D. P. Mehta, S. Sahni. – 2nd ed. – London: Chapman and Hall/CRC, 2018. – 1120 pp.
40. Takaoka T. Theory of 2-3 Heaps / T. Takaoka // Discrete Applied Math. – 2003. – Vol. 126. – pp. 115-128.
41. Brodal G. S. Optimal purely functional priority queues / G. S. Brodal, C. Okasaki // Journal of Functional Programming. – 1996. – Vol. 6. – pp. 839-857.

42. Pagh R. Cuckoo Hashing / R. Pagh, F. R. Flemming // Journal of Algorithms. – 2004. – Vol. 51. – pp. 122-144.
43. Fredman M. L. Storing a Sparse Table with O(1) Worst Case Access Time / M. L. Fredman, J. Komlos, E. Szemeredi // Journal of the ACM. – 1984. – Vol. 31. – pp. 538-544.
44. Herlihy M. Hopscotch Hashing / M. Herlihy, N. Shavit, M. Tzafrir // Proceedings of the 22nd international symposium on Distributed Computing. – Arcachon, 2008. – pp. 350–364.
45. Фильтр Блума. – URL: https://ru.wikipedia.org/wiki/Фильтр_Блума (дата обращения: 15.10.2019).
46. Hash array mapped trie. – URL: https://habrahabr.ru/post/266861/ (дата обращения: 15.10.2019).
47. Гулаков В. К. Многомерные структуры данных / В. К. Гулаков, А. О. Трубаков. – Брянск: БГТУ, 2010. – 387 с.
48. Добрица В. П. ПРИМЕРЫ РЕШЕНИЯ ГЕОЛОГИЧЕСКИХ ЗАДАЧ В ГЕОЛОГИЧЕСКОЙ ИНФОРМАЦИОННОЙ СИСТЕМЕ "ГЕОМИКС" / В. П. Добрица, Т. В. Иванова // ВЕСТНИК МОСКОВСКОГО ГОРОДСКОГО ПЕДАГОГИЧЕСКОГО УНИВЕРСИТЕТА. СЕРИЯ: ИНФОРМАТИКА И ИНФОРМАТИЗАЦИЯ ОБРАЗОВАНИЯ. – 2018. – № 4. – С. 34-39.
49. Добрица В. П. СОВЕРШЕНСТВОВАНИЕ МЕТОДА ОБРАБОТКИ ГЕОЛОГИЧЕСКИХ ДАННЫХ С ПОМОЩЬЮ ПРИМЕНЕНИЯ ПРОГРАММЫ SURFER НА ПРИМЕРЕ МОДЕЛИРОВАНИЯ ГЕОХИМИЧЕСКОЙ КАРТЫ / В. П. Добрица, Е. И. Горюшкин, Т. В. Иванова // ИЗВЕСТИЯ ЮГО-ЗАПАДНОГО ГОСУДАРСТВЕННОГО УНИВЕРСИТЕТА. – 2019. – Т. 23. – № 5. – С. 175-184.
50. Добрица В. П. СУЩЕСТВОВАНИЕ КЛАССИФИЦИРУЮЩЕЙ НЕЙРОННОЙ СЕТИ ДЛЯ ПРОИЗВОЛЬНОГО РАЗБИЕНИЯ ВЕРШИН N-МЕРНОГО КУБА НА ДВА МНОЖЕСТВА / В. П. Добрица, Н. С. Уалиев,

Д. Н. Нургабыл // НЕЙРОКОМПЬЮТЕРЫ: РАЗРАБОТКА, ПРИМЕНЕНИЕ. – 2014. – № 6. – С. 12-49.

51. Gaede V. Multidimensional Access Methods / V. Gaede, O. Gunther // ACM Computing Surveys. – 1998. – Vol. 30. – No. 2. – pp. 170-231.
52. Анализ и некоторая классификация методов доступа к данным / В. И. Аверченков, В. К. Гулаков, А. О. Трубаков [и др.] // Вестник компьютерных и информационных технологий. – 2014. – Т. 12. – С. 48-55.
53. Vaishnavi V. K. Multidimensional height-balanced trees / V. K. Vaishnavi // IEEE Transactions on Computers. – 1984. – Vol. 33. – No. 4. – pp. 334-343.
54. Samet H. Applications of Spatial Data Structures / H. Samet. – Boston: Addison-Wesley Longman Publishing Co., 1990. – 507 pp.
55. Потапов Д. Р. Обзор методов построения многомерных контейнеров данных "ключ-значение" для использования в самоадаптирующихся контейнерах данных / Д. Р. Потапов // Прикладная информатика. – 2018. – № 2(74). – С. 69-82.
56. Kronacker M. High-concurrency locking in R-trees / M. Kronacker, D. Banks // 21th International Conference on Very Large Data Bases. – San Francisco, 1995. – pp. 134–145.
57. Sellis T. The R+Tree: A dynamic index for multi-dimensional objects / T. Sellis, C. Faloutsos, N. Roussopoulos // Proceedings of the 13th International Conference on Very Large Data Bases. – San Francisco, 1987. – pp. 507–518.
58. The R*-tree: an efficient and robust access method for points and rectangles / N. Beckmann, H. Kriegel, R. Schneider, B. Seeger // Proceedings of the 1990 ACM SIGMOD international conference on Management of data. – New York, 1990. – pp. 322-331.
59. Berchtold S. The X-tree: An Index Structure for High-Dimensional Data / S. Berchtold, D. A. Keim, H. P. Kriegel // Proceeding VLDB '96 Proceedings of the

22th International Conference on Very Large Data Bases. – San Francisco, 1996. – pp. 28–39.

60. Ciaccia P. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces / P. Ciaccia, M. Patella, P. Zezula // Proceedings of the 23rd International Conference on Very Large Data Bases. – San Francisco, 1997. – pp. 426–435.
61. White D. A. Similarity indexing with the ss-tree / D. A. White, R. Jain // Twelfth International Conference on Data Engineering. – Washington, 1996. – pp. 516–523.
62. Katayama N. The sr-tree: an index structure for high-dimensional nearest neighbor queries / N. Katayama, S. Satoh // ACM SIGMOD international conference on Management of data. – New York, 1997. – pp. 369–380.
63. Jain R. Similarity Indexing: Algorithms and Performance / R. Jain, D. A. White // Proc. SPIE Storage and Retrieval for Image and Video Databases IV. – 1996. – Vol. 2670. – pp. 62–75.
64. Fuchs H. On Visible Surface Generation by A Priori Tree Structures / H. Fuchs, Z. Kedem, B. F. Naylor // Proceedings of the 7th annual conference on Computer graphics and interactive techniques. – New York, 1980. – pp. 124–133.
65. Bentley J. L. Multidimensional binary search trees used for associative searching / J. L. Bentley // Communications of the ACM. – 1975. – Vol. 18. – pp. 509–517.
66. Bentley J. L. Data structures for range searching / J. L. Bentley, J. H. Friedman // ACM Computing Surveys. – 1979. – Vol. 11. – pp. 397–409.
67. Robinson J. T. The K-D-B-tree: A search structure for large multidimensional dynamic indexes / J. T. Robinson // ACM SIGMOD International Conference on Management of Data. – 1981. – pp. 10–18.
68. Lomet D. B. The hB-tree: A robust multiattribute search structure / D. B. Lomet, B. Salzberg // Proceedings of the Fifth International Conference on Data Engineering. – Washington, 1989. – pp. 296–304.

69. Henrich A. The LSD tree: Spatial access to multidimensional point and non-point objects / A. Henrich, H. W. Six, P. Widmayer // Fifteenth International Conference on Very Large Data Bases. – San Francisco, 1989. – pp. 45–53.
70. Ooi B. Spatial KD-tree: An Indexing Mechanism for Spatial / B. Ooi, K. J. McDonell, R. Sacks-Davis // Proceedings of the IEEE COMPSAC Conference. – 1987. – pp. 433–438.
71. Finkel R. A. Quad Trees: A Data Structure for Retrieval on Composite Keys / R. A. Finkel, J. L. Bentley // Acta Informatica. – 1974. – Vol. 4. – No. 1. – pp. 1–9.
72. Zeng M. Octree-based fusion for realtime 3D reconstruction / M. Zeng, F. Zhao, J. Zheng // Graph Model. – 2013. – Vol. 75. – No. 3. – pp. 126–136.
73. Yianilos P.N. Data structures and algorithms for nearest neighbor search in general metric spaces / P. N. Yianilos // Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms. – Philadelphia, 1993. – pp. 311–321.
74. Ohsawa Y. BD-Tree: A New N-dimensional Data Structure with Efficient Dynamic Characteristics / Y. Ohsawa, M. Sakauchi // Proc. 9th World Computer Congress, IFIP83. – Paris, 1983. – pp. 539–544.
75. Nievergelt J. The Grid File: An Adaptable, Symmetric Multikey File Structure / J. Nievergelt, H. Hinterberger, K. C. Sevcik // ACM Transactions on Database Systems. – 1984. – Vol. 9. – pp. 38–71.
76. Hutflesz A. Twin grid files: Space optimizing access schemes / A. Hutflesz, H. W. Six, P. Widmayer // ACM SIGMOD International Conference on Management of Data. – New York, 1988. – pp. 183–190.
77. Whang K.Y. The Multilevel Grid File - A Dynamic Hierarchical Multidimensional File Structure / K. Y. Whang, R. Krishnamurthy // Proceedings of the Second International Symposium on Database Systems for Advanced Applications. – Tokyo, 1991. – pp. 449–459.

78. Kriegel H. P. Multidimensional order preserving linear hashing with partial expansions / H. P. Kriegel, B. Seeger // Proceedings of the International Conference on Database Theory, LNCS 243. – London, 1986. – pp. 203–220.
79. Kriegel H. P. PLOP-hashing: A grid file without directory / H. P. Kriegel, B. Seeger // Proceedings of the Fourth IEEE International Conference on Data Engineering. – Washington, 1988. – pp. 369–376.
80. Seeger B. The buddy-tree: An efficient and robust access method for spatial database systems / B. Seeger, H. P. Kriegel // 16th International Conference on Very Large Data Bases. – San Francisco, 1990. – pp. 590–601.
81. Freeston M. The BANG file: A new kind of grid / M. Freeston // Proceedings of the ACM SIGMOD International Conference on Management of Data. – New York, 1987. – pp. 260–269.
82. Hutflesz A. The R- file: An efficient access structure for proximity / A. Hutflesz, H. W. Six, P. Widmayer // Proceedings of the Sixth IEEE International Conference on Data Engineering. – Washington, 1990. – pp. 372–379.
83. Orenstein J. A class of data structures for associative searching / J. Orenstein, T. H. Merrett // Proc. 3rd ACM SIGACT-SIGMOD Symp. on Principles of Database Systems. – New York, 1984. – pp. 181–190.
84. Faloutsos C. Multiattribute hashing using Gray-codes / C. Faloutsos // Proc. ACM SIGMOD Int. Conf. on Management of Data. – New York, 1986. – pp. 227–238.
85. Kamel I. Hilbert Rtree: An improved R-tree using fractals / I. Kamel, C. Faloutsos // Proceedings of the Twentieth International Conference on Very Large Data Bases. – San Francisco, 1994. – pp. 500–509.
86. Bayer R. The universal B-tree for multidimensional indexing / R. Bayer // Proceeding WWCA '97 Proceedings of the International Conference on Worldwide Computing and Its Applications. – Tsukuba, 1997. – pp. 198–209.

87. A Geographic Meshing and Coding Method Based on Adaptive Hilbert-Geohash / Guo N., Xiong W., Wu Y., Chen L. // IEEE Access. – 2019. – Vol. 7. – pp. 39815 - 39825.

88. Moenne-Loccoz N. High-Dimensional Access Methods for Efficient Similarity Queries / N. Moenne-Loccoz. – University of Geneva, Computer Vision and Multimedia Laboratory, Geneva, 2005. – Technical Report N:0505.

89. Kim K. Optimizing multidimensional index trees for main memory access / K. Kim, S. K. Cha, K. Kwon // Proceedings of the 2001 ACM SIGMOD international conference on Management of data. – New York, 2001. – pp. 139–150.

90. Kim K. C. MR-Tree: A Cache-Conscious Main Memory Spatial Index Structure for Mobile / K. C. Kim, S. W. Yun // 4th International Workshop, W2GIS. – Berlin, 2004. – pp. 167-180.

91. Gunther O. Oversize shelves: A storage management technique for large spatial data objects / O. Gunther, V. Gaede // International Journal of Geographical Information Science. – 1997. – Vol. 11. – No. 1. – pp. 5–32.

92. Oosterom P. Reactive data structures for geographic information systems / P. Oosterom. – New York: Oxford University Press, 1994. – 198 pp.

93. Nelson R. A population analysis for hierarchical data structures / R. Nelson, H. Samet // Proceedings of the SIGMOD Conference. – San Francisco, 1987. – pp. 270–277.

94. Таненбаум Э. Архитектура компьютера / Э. Таненбаум. – Санкт-Петербург: Питер, 2016. – 816 с.

95. Потапов Д. Р. Обзор алгоритмов кэширования для использования в самоадаптирующихся контейнерах данных / Д. Р. Потапов // Информатика: проблемы, методы, технологии: Матер. XX Международной научно-методической конференции. – Воронеж, 2020. – С. 1273-1282.

96. Корнеев В. В. Современные микропроцессоры / В. В. Корнеев, А. В. Киселев. – Москва: Нолидж, 1998. – 240 с.

97. Karedla R. Caching Strategies to Improve Disk System Performance / R. Karedla, J. S. Love, B. G. Wherry // Computer. – 1994. – Vol. 27. – No. 3. – pp. 38 - 46.
98. Sanderson D. Programming Google App Engine with Python / D. Sanderson. – Sebastopol: O’Reilly Media, 2015. – 538 pp.
99. Carlson J.L. Redis in Action / J. L. Carlson. – New York: Manning Publications, 2013. – 320 pp.
100. Macedo T. Redis Cookbook / T. Macedo, F. Oliveria. – Sebastopol: O’Reilly Media, 2011. – 78 pp.
101. Pandey S. K. Caching using Memcached in Open Source / S. K. Pandey. – Searching: Execution Time and. Riga: LAP LAMBERT Academic Publishing, 2015. – 80 pp.
102. Soliman A. Getting Started with Memcached / A. Soliman. – Birmingham: Packt Publishing, 2013. – 56 pp.
103. Mookerjee V. S. Analysis of a Least Recently used Cache Management Policy for Web Browsers / V. S. Mookerjee, Y. Tan // Operations Research. – 2002. – Vol. 50. – No. 2. – pp. 345-357.
104. Bilal M. Time Aware Least Recent Used (TLRU) cache management policy in ICN / M. Bilal, S. G. Kang // 16th International Conference on Advanced Communication Technology. – Pyeongchang, 2014. – pp. 528-532.
105. Gille D. Study of Different Cache Line Replacement Algorithms in Embedded Systems: Master’s thesis / D. Gille. – Stockholm, 2007. – p. 104.
106. So K. Cache Operations by MRU Change / K. So, R. N. Rechtschaffen // IEEE Transaction Computers. – 1988. – Vol. 37. – No. 6. – pp. 700-709.
107. O'Neil E. J. The LRU-K Page Replacement Algorithm / E. J. O'Neil, P. E. O'Neil, G. Weikum // SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data. – Washington, 1993. – pp. 297-306.

108. O'Neil E. J. An Optimality Proof of the LRU-K page replacement algorithm / E. J. O'Neil, P. E. O'Neil, G. Weikum // Journal of the ACM. – 1999. – Vol. 46. – No. 1. – pp. 92-112.

109. Megiddo N. ARC: A Self-Tuning, Low Overhead Replacement / N. Megiddo, D. S. Modha // FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies, USENIX Association. – San Francisco, 2003. – pp. 115-130.

110. Johnson T. 2Q: A Low Overhead High Performance Buffer / T. Johnson, D. Shasha // VLDB '94: Proceedings of the 20th International. – Santiago de Chile, 1994. – pp. 439-450.

111. Zhou Y. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches / Y. Zhou, J. Philbin, K. Li // Proceedings of the General Track: 2001 USENIX Annual Technical Conference. – Berkeley, 2001. – pp. 91-104.

112. Qureshi M. K. Adaptive insertion policies for high performance caching / M. K. Qureshi, A. Jaleel, S. C. Steely // Proceedings of the 34th annual international symposium on Computer architecture. – San Diego, 2007. – pp. 381-391.

113. Bhutra A. Reviewing various Cache Replacement Policies / A. Bhutra // ResearchGate. – 2015. – URL: https://www.researchgate.net/publication/301546620 (дата обращения: 25.06.2019).

114. Jiang S. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance / S. Jiang, X. Zhang // ACM SIGMETRIC. – Marina Del Rey, 2002. – pp. 31–42.

115. Einziger G. TinyLFU: A Highly Efficient Cache Admission Policy / G. Einziger, R. Friedman // Proceedings of the 2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. – Washington, 2014. – pp. 146-153.

116. Bilal M. A Cache Management Scheme for Efficient Content Eviction and Replication in Cache Networks / M. Bilal, S. Kang // IEEE Access. – 2017. – Vol. 5. – pp. 1692–1701.

117. Prabhashankar J. An Adaptive Dynamic Replacement Approach for a Multicast based Popularity Aware Prefix Cache Memory System / J. Prabhashankar, T. R. Gopalakrishnan Nair // InterJRI Computer Science and Networking. – 2009. – Vol. 1. – No. 1. – pp. 24-30.

118. Smith A.J. Sequentiality and prefetching in database systems / A. J. Smith // ACM Transactions on Database Systems (TODS). – 1978. – Vol. 3. – No. 3. – pp. 223-247.

119. Jiang S. CLOCK-Pro: An Effective Improvement of the CLOCK Replacement / S. Jiang, F. Chen, X. Zhang // 2005 USENIX Annual Technical Conference. – Anaheim, 2005. – pp. 323-336.

120. Bansal S. CAR: Clock with Adaptive Replacement / S. Bansal, D. S. Modha // Proceedings of the Third USENIX. – San Francisco, 2004. – pp. 187-200.

121. Kacimi M. Evaluation Study of a Distributed Caching Based on Query / M. Kacimi, K. Yetongnon // InfoScale '07 Proceedings of the 2nd international conference on Scalable information systems. – Suzhou, 2007. – pp. 1-7.

122. Jacob B. Memory Systems: Cache, DRAM, Disk / B. Jacob, D. Wang, N. Spencer. – San Francisco: Morgan Kaufmann Publishers Inc, 2007. – 900 pp.

123. Советов Б. Я. Архитектура информационных систем / Б. Я. Советов. – Москва: Издательский центр «Академия», 2012. – 288 с.

124. Гарсиа-Молина Г. Системы баз данных. Полный курс / Г. Гарсиа-Молина, Д. Ульман, Д. Уидом. – Москва: Вильямс, 2004. – 1088 с.

125. Дейт К. Д. Введение в системы баз данных / К. Д. Дейт. – Москва: Вильямс, 2006. – 1328 pp.

126. Жуков А. И. Модель адаптивного векторного управления стохастическим гибридным алгоритмом кэширования / А. И. Жуков // Вестник Донского государственного технического университета. – 2012. – Т. 12, № 5. – С. 19-29.

127. Bishop C. Pattern Recognition and Machine Learning / C. Bishop. – Heidelberg: Springer, 2006. – 738 pp.

128. McLachlan G. Finite Mixture Models / G. McLachlan, D. Peel. – New York: John Wiley & Sons, 2004. – 419 pp.

129. Cache-Aside pattern // Microsoft Docs. – URL: https://docs.microsoft.com/en-us/azure/architecture/patterns/cache-aside (дата обращения: 19.11.2019).

130. Королёв В.Ю. EM-алгоритм, его модификации и их применение к задаче разделения смесей вероятностных распределений. Теоретический обзор / В.Ю. Королёв. – Москва: ИПИ РАН, 2007. – 102 с.

131. McLachlan G. The EM algorithm and extensions. Wiley series in probability and statistics / G. McLachlan, T. Krishnan. – New York: John Wiley & Sons, 1997. – 400 pp.

132. Blömer J. Adaptive Seeding for Gaussian Mixture Models / J. Blömer, K. Bujna // PAKDD 2016 Proceedings, Part II, of the 20th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining. – Auckland, 2016. – pp. 296-308.

133. Baudry J. P. EM for Mixtures / J. P. Baudry, G. Celeux // Statistics and Computing. – 2015. – Vol. 25, No. 4. – pp. 713–726.

134. Melnykov V. Initializing the EM algorithm in Gaussian mixture models with an unknown number of components / V. Melnykov, I. Melnykov // Computational Statistics & Data Analysis. – 2012. – Vol. 56, No. 6. – pp. 1381-1395.

135. Biernacki C. Choosing starting values for the EM algorithm for getting the highest likelihood in multivariate Gaussian mixture models / C. Biernacki, G. Celeux, G. Govaert // Computational Statistics & Data Analysis. – 2003. – Vol. 41, No. 3. – pp. 561–575.

136. Meila M. An Experimental Comparison of Several Clustering and Initialization Methods / M. Meila, D. Heckerman // Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence. – San Francisco, 1998. – pp. 386–395.
137. Arthur D. K-means++: The Advantages of Careful Seeding / D. Arthur, S. Vassilvitskii // Proceedings of the 8th Annual ACMSIAM Symposium on Discrete Algorithms. – Philadelphia, 2007. – pp. 1027–1035.
138. Scalable k-means++ / Bahmani B., Moseley B., Vattani A. [and etc.] // Proceedings of the VLDB Endowment. – 2012. – Vol. 5. – pp. 622-633.
139. Zhao W. Parallel K-means clustering based on mapReduce / W. Zhao, H. Ma, Q. He // Proceedings of the 1st International Conference on Cloud Computing. – 2009. – Vol. 5931. – pp. 674–679.
140. Fast Scalable kmeans++ Algorithm with MapReduce / Y. Xu, W. Qu, Y. Li [and etc.] // Proceedings International Conference on Algorithms and Architectures for Parallel Processing. – Dalian, 2014. – Vol. 8631. – pp. 15-28.
141. A framework for clustering evolving data streams / C.C. Aggarwal, J. Han, J. Wang, P.S. Yu // Proceedings of the 29th international conference on Very large data bases. – Berlin, 2003. – pp. 81-92.
142. Liang P. Online EM for unsupervised models / P. Liang, D. Klein // Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics. – Boulder, 2009. – pp. 611-619.
143. Deitel P. C# 6 for Programmers / P. Deitel, H. Deitel. – 6th ed. – Upper Saddle River: Prentice Hall, 2016. – 768 pp.
144. Walkenbach J. Microsoft Excel 2016 bible: the comprehensive tutorial resource / J. Walkenbach. – New York: John Wiley & Sons, 2015. – 1152 pp.
145. Troelsen A. C# 6.0 and the. NET 4.6 Framework / A. Troelsen, P. Japikse. – New York: Apress, 2015. – 1832 pp.

146. Unsupervised machine learning with multivariate Gaussian mixture model which supports both offline data and real-time data stream. – URL: https://github.com/lukapopijac/gaussian-mixture-model (дата обращения: 19.11.2019).

147. Круглов В. М. Предельные теоремы для случайных сумм / В. М. Круглов, В. Ю. Королев. – Москва: Издательство Московского университета, 1990. – 269 с.

148. Гмурман В. Е. Теория вероятностей и математическая статистика: учебник для прикладного бакалавриата / В. Е. Гмурман. – Москва: Издательство Юрайт, 2014. – 479 с.

149. Bradley P. S. Scaling EM (Expectation-Maximization) Clustering to Large Databases / P. S. Bradley, U. M. Fayyad, C. A. Reina. – Microsoft Corporation, Redmond, 1998. – Technical Report MSR-TR-98-35.

150. cache2k - High Performance Java Caching, Benchmarks for cache2k. – URL: https://github.com/cache2k/cache2k-benchmark (дата обращения: 19.11.2019).

151. Hsu W. W. The automatic improvement of locality in storage systems / W. W. Hsu, H. C. Young, A. J. Smith // ACM Transactions on Computer Systems. – 2005. – Vol. 23, No. 4. – pp. 424-473.