Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Санкт-Петербургский государственный университет»

Кафедра Системного Программирования

Бутрова Александра Сергеевна

Алгоритмы предварительной подгрузки для рабочей нагрузки с большой долей случайных запросов в СХД

Курсовая работа

Научный руководитель:
к. т. н., доцент Лазарева С. В.

Санкт-Петербург
2016

Оглавление

Введение **3**

1. Обзор существующих решений **4**

1.1. SARC 4

1.2. Markov prefetching 5

1.3. Global History Buffer 5

1.4. Алгоритмы анализа данных 6

1.4.1. C-Miner 9

2. Постановка задачи **11**

3. Предлагаемое решение **12**

3.1. Режим обучения 12

3.1.1. Связанные определения 12

3.1.2. Сбор информации 13

3.1.3. Составление правил 16

3.2. Режим работы 16

4. Результаты **18**

Заключение **20**

Список литературы **21**

Введение

В современном мире объём генерируемой информации с каждым днём растёт экспоненциально, количество пользователей, запрашивающих эту информацию, увеличивается – всё это усиливает нагрузку на системы хранения данных (СХД).

Производительность СХД характеризуют два основных показателя: время доступа и полоса пропускания. Эти показатели можно улучшить путем усовершенствования программной части.

Одной из идей оптимизации является подгрузка страниц до того момента, как они будут запрошены. Для различных типов запросов используются различные технологии: для последовательных (sequential) используется упреждающее чтение (read ahead), которое загружает одну или несколько следующих страниц; для случайных (random) – предварительная загрузка (prefetching).

Последняя технология редко используется для последовательных запросов, так как является более сложной и ресурсозатратной относительно упреждающего чтения, которое показывает такие же результаты.

В данной работе рассматриваются нагрузки (workload) с большой долей случайных запросов и их оптимизация с помощью предварительной подгрузки данных.

1. Обзор существующих решений

Блочная система хранения данных предоставляет небольшое количество сведений о запросе, а именно: адрес, размер и время прихода запрашиваемых данных. Ниже рассмотрены алгоритмы, которые, используя лишь эти параметры, могут предсказать что, куда и когда подгрузить.

1.1. SARC

Первым рассматриваемым алгоритмом будет SARC (Sequential Prefetching in Adaptive Replacement Cache) [2]. Он основывается на идее разделения последовательных (sequential) и случайных (random) запросов. Для этого используется два адаптивных списка, в зависимости от попаданий (cache hit) и промахов (cache miss) меняющих не только свой размер, но и скорость изменения размера списков. Эти изменения называются адаптацией и описываются несколькими параметрами.

Этапы работы алгоритма:

- Если данные попали в случайный список, то изменяются переменные адаптации в пользу увеличения размера случайного списка и считываемый блок подгружается в список.
- Если данные попали в последовательный список, то увеличивается адаптация для последовательного списка, подгружается считываемый блок с некоторым интервалом блоков вперед.
- Если случился промах
  - если предыдущий блок был подгружен ранее, то в зависимости от длины уже подгруженной последовательности погружаем данные в последовательный или случайный список.
  - иначе ничего не подгружаем.
- Адаптация происходит только когда блок данных вытеснен из одного из списков.

Можно заметить, что минусы такого подхода в том, что список для случайных запросов может быть небольшим, и данные в нем могут вытесняться довольно быстро, что влечет лишь загрязнение кэша (cache

pollution). Одним из решений может быть прогнозирование следующих случайных запросов на основе имеющихся.

1.2. Markov prefetching

Далее рассмотрим модель, предложенную А. Марковым (Markov Prediction Scheme for Cache Prefetching)[5], которая строит таблицу истории, на основании которой можно подгружать данные.

Алгоритм использует двухуровневую модель кэша L1 и L2, а также два дополнительных хранилища: таблица истории промахов (miss history table), в которую записывается следующий блок данных после промаха в L1, и буфер предвыборки (Prefetch Buffer), который находится в L1 и содержит блоки предварительной выборки.

Рассмотрим алгоритм:

- Если произошёл промах в L1-кэше
  - Проверяем L2-кэш и буфер предвыборки.
  - Изменяем таблицу истории промахов.
  - На основании истории промахов и входящих запросах подгружаем данные в буфер предвыборки.

"Чистая" модель Маркова имеет ряд недостатков, например, на практике последовательности запросов не повторяют одних и тех же шаблонов, и вероятности, полученные в первом исполнении, могут давать неточные результаты для последующих исполнений. Поэтому модель Маркова используется совместно с другими алгоритмами.

1.3. Global History Buffer

Следующее решение – Буфер Глобальной Истории(Global History Buffer[3]) – также является концепцией, применяемой непосредственно с другими алгоритмами. Идея возникла в связи с потребностью уменьшить количество используемой памяти и увеличить актуальность информации. Осуществляется это с помощью дополнительной структуры, которая называется буфер глобальной истории. Буфер поддерживает связный список, который содержит адреса с одинаковыми свойствами

(обычно этим свойством является совпадение адресов). К структуре доступ осуществляется из хэш-таблицы, где ключом является адрес, а значением – ссылка на элемент с таким адресом в буфере.

1.4. Алгоритмы анализа данных

В связи с необходимостью обрабатывать большие объемы данных, было принято решение обратиться к алгоритмам интеллектуального анализа данных (Data Mining). Это решение предполагает поиск ассоциативных правил, которые позволяют находить закономерности среди объектов.

Первым алгоритмом был Априори (Apriori) [6]. Для его понимания необходимо ввести терминологию.

I $= i_1, i_2, ..., i_n$ – набор элементов.

Транзакция T – набор элементов, такой что T $\in$ I.

Последовательность – лист $(s_1, s_2, ..., s_n)$, состоящий из набора элементов s_i.

База данных D – множество последовательностей.

Набор $(a_1, a_2, ...a_n)$ содержится в наборе $(b_1, b_2, ..., b_m)$, если существует номера $i_1 < i_2 < ... < i_n$ такие, что $a_1 \in b_{i_1}, a_2 \in b_{i_2}, ..., a_n \in b_{i_n}$.

Клиентская последовательность – множество транзакций, упорядоченных по времени.

Длина последовательности – количество элементов в ней.

Поддержка элемента (support) – количество клиентских последовательностей, которые содержат последовательность с данным элементом.

Частая последовательность – последовательность с поддержкой не меньшей заданного порога (minimum support).

Цель алгоритма: найти все частные последовательности во всей базе данных с заданной минимальной поддержкой.

Данный алгоритм был придуман для анализа покупок ещё в 1994 году, но до сих пор появляются его различные модификации и для других областей, таких как кэш с предварительной подгрузкой данных.

ID клиента	ID клиента	ID клиента
1	01.04.2016	1, 3, 15
2	27.03.2016	1, 15
2	07.04.2016	4, 15
3	01.04.2016	1, 3, 7
4	29.03.2016	3, 15
4	01.04.2016	1, 15, 16

а)

ID клиента	Клиентская последовательность
1	<(1, 3, 15)>
2	<(1, 15), (4, 15)>
3	<(1, 3, 7)>
4	<(3, 15), (1, 15, 16)>

б)

Элемент	Поддержка
1	4
3	3
4	1
7	1
15	3
16	1

в)

ID клиента	Клиентская последовательность
1	<(1, 3, 15)>
2	<(1, 15), (15)>
3	<(1, 3)>
4	<(3, 15), (1, 15)>

г)

Рис. 1: Некоторые фазы работы алгоритма: а) исходная база данных; б) результат фазы сортировки; в) результат фазы набора элементов; г) результаты фазы трансформации.

Работа алгоритма делится на 4 этапа:

1. Фаза сортировки: сортируем всю базу данных по клиентским последовательностям и времени.

2. Фаза набора элементов: найти все единичные частые последовательности и их поддержку.

3. Фаза трансформации: в каждой транзакции оставить только частные последовательности.

4. Фаза последовательности: использовать одну из модификаций алгоритма Априори для нахождения всех частых последовательностей.

Последняя фаза заслуживает особое внимание – именно она подвергается различным модификациям.

В статье [1] приведены некоторые параметры классификации алгоритмов интеллектуального анализа данных последовательностей (sequence Data Mining). Первоначально авторы делят все алгоритмы на два типа: основанные на Априори (Apriori Based) и основанные на

росте шаблонов (Pattern Growth Based). Рассмотрим свойства, присущие каждому из типов:

- Априори [4]:
    1. Обход в ширину (Breadth-first search): на k-ой итерации находит все k-элементные последовательности.
    2. Генерация и тестирование (Generate-and-test): сначала алгоритм создает все возможные последовательности, затем путем проверки каждого кандидата на выполнение условий удаляет неподходящие, тем самым потребляем большое количество памяти на ранних стадиях работы алгоритма.
    3. Многократный обход базы данных (Multiple scans of the database): эта особенность предполагает сканирование исходной базы данных, чтобы установить является ли список кандидатов частым или нет. Это самое ресурсоёмкое свойство, характерное для большинства алгоритмов на основе Априори.
- Основанные на росте шаблонов:
    1. Разбиение пространства поиска (Search space partitioning): алгоритм разбивает всю историю запросов на окна (windows), затем анализирует каждое из них. У такого подхода есть два существенных плюса: возможность легко параллельно обрабатывать окна и оптимизация алгоритма путем равномерного распределения частых последовательностей.
    2. Проектирование дерева (Tree projection): большинство алгоритмов такого типа предполагают построение дерева, по которому будет идти поиск частых последовательностей.
    3. Обход в глубину (Depth-first traversal): это свойство дает большой выигрыш в производительности, а также помогает отсекать неподходящих кандидатов на ранних стадиях и находить замкнутые последовательности. Основной причиной этого является тот факт, что обход в глубину использует гораздо меньше памяти, более направленное пространство по-

иска, и, таким образом, меньше генерирует последовательности кандидатов, чем поиск в ширину.

4. Отсечение кандидатов (Candidate sequence pruning): в алгоритмах пытаются использовать структуру данных, которая позволяет отсечь кандидатов в начале процесса поиска.

Существует ещё одна важная классификация:

- Многомерные шаблоны (Multidimensional Sequential Pattern Mining [9]): характеризуется наличием нескольких элементов с одним временным показателем.
- Шаблоны на основе ограничений (Discovering Constraint Based Sequential Pattern [13]): характеризуются дополнительными параметрами, которые нужно учесть при поиске последовательностей.
- Шаблоны с временными интервалами (Discovering Time-interval Sequential Pattern): характеризуется наличием необходимости знания интервала времени, проходящего между элементами одной последовательности.
- Замкнутые шаблоны (Closed Sequential Pattern Mining): характеризуется высокой производительностью по сравнению с другими типами при условии наличия длинных последовательностей в малом количестве или наличия большого числа последовательностей с одинаковой поддержкой.

1.4.1. C-Miner

Последний алгоритмом, который рассмотрим, является C-Miner [8].

C-Miner основывывается на алгоритме CloSpan[7], который относится к типу замкнутых шаблонов и к алгоритмам, основанным на росте шаблона.

Этот алгоритм адаптирован для систем хранения данных и хорошо справляется с большим количеством правил с почти одинаковой поддержкой.

Рассмотрим этапы его работы:

1. Среди входящих запросов ищем короткие повторяющиеся ограниченные подпоследовательности запросов.

2. Из полученных подпоследовательностей извлекаются суффиксы, которые затем упорядочиваются в соответствии с частотой встречаемости.

3. На основе найденных суффиксов и цепочек суффиксов генерируются правила, описывающие наиболее вероятные блоки, которые будут запрошены по адресам из коррелирующей области.

4. Полученные правила сортируются по их надёжности.

Алгоритм хорошо подходит для систем хранения данных, данные в которых обновляются редко, а запрашиваются часто и небольшими порциями. В этом случае можно составить малоизменяемые правила, по которым будет производиться упреждающее чтение, и потери будут только при первых запросах, когда эти правила создаются.

2. Постановка задачи

Целью данной работы является улучшение алгоритма упреждающего чтения, используемого в ПО RAIDIX, для уменьшения рабочей нагрузки с большой долей случайных запросов. Для достижения результатов выделены следующие пункты:

- Исследовать существующие алгоритмы предварительной подгрузки.
- Разработать алгоритм выявления связей между случайными запросами.
- Реализовать модель для анализа результатов.
- Провести тестирование реализованного модуля.

3. Предлагаемое решение

Реализована модель, которая имеет два режима работы: режим обучения модуля предварительной загрузки и режим работы этого модуля.

Для построения модели был выбран алгоритм C-Miner, который используется для режима обучения. По завершению работы этого режима получим ассоциативные правила, которые имеют вид: $a_1, a_2, ..., a_n \rightarrow b$. Эта запись означает, что после получения запроса на блоки $a_1, a_2, ..., a_n$ появится запрос на блок b с заданной вероятностью.

Такие правила позволяют легко отслеживать запросы, которые удовлетворяют этим правилам, и предварительно подгружать нужные блоки. Этим будет заниматься отдельный модуль prefetcher.

3.1. Режим обучения

Данный режим делится на три этапа: сбор информации, обработка информации и составление правил.

3.1.1. Связанные определения

Входными данными являются запросы, о которых известна следующая информация:

lba – адрес запрашиваемых данных;

size – размер запрашиваемых данных;

action – действие, которое необходимо выполнить: читать (read) или записать (write);

time – время прихода запроса.

Определения для понимания работы алгоритма:

Последовательность – набор адресов lba, состоящих из одного или более элементов.

Нашей задачей является поиск наиболее частых последовательностей, но также нам не интересны частые последовательности, которые встретились много раз на небольшом интервале времени, то есть на небольшом интервале истории. Это обосновывается тем, что в будущем

эта последовательность встретится с маленькой вероятностью. Чтобы отсечь такие случаи разобьем всю историю на окна (window), в которых по отдельности будем искать последовательности. Размер окна (window size) является параметром программы.

Поддержка последовательности (support) – количество окон, в которых хотя бы раз встречается данная последовательность.

Частая последовательность – последовательность, которая имеет заданную минимальную поддержку или больше. Минимальная поддержка также является параметром программы.

Надпоследовательность последовательности – последовательность, которая содержит в себе данную. То есть последовательность $(b_1, b_2, ..., b_n)$ является надпоследовательностью последовательности $(a_1, a_2, ..., a_m)$, если $n > m$ и существует такая последовательность индексов $(k_1, k_2, ..., k_m)$, что $a_1 = b_{k_1}, a_2 = b_{k_2}, ..., a_m = b_{k_m}$.

3.1.2. Сбор информации

На этом этапе работает обычная модель кэша с занесением информации в историю запросов, следуя правилам разделения на окна (см. рис. 2a).

Сохранение информации происходит, если входящий запрос на чтение данных.

Алгоритм обработки информации получает на вход историю запросов и начинает всю работу, проходя этапы:

1. Поиск единичных частых последовательностей.

   Обозначим множество частых последовательностей за L.

   (a) Сканируем историю, сохраняя встретившиеся единичные последовательности в L, и подсчитываем поддержку каждого элемента.

   (b) Удаляем из L последовательности, поддержка которых меньше минимальной (параметр minimum support).

Последовательность	Поддержка
1	3
2	2
4	3
5	1
6	1
7	4

Последовательность	Поддержка
1	3
4	3
7	4

Рис. 2: а) этап сбора информации: запрос на чтение по адресу 7; б) заполненная история, которая получается после сбора информации; в) найденное множество одноэлементных последовательностей с поддержкой элементов; г) множество частых единичных последовательностей L для поддержки support = 3.

2. Генерация кандидатов.

   На k-ом проходе генерирует k-элементные последовательности-кандидаты. Обозначим Ck.

   (a) Генерируем всех кандидатов путем склеивания имеющихся частых последовательностей, полученных на предыдущих шагах.

   (b) Если после склеивания получилась последовательность-кандидат, которая содержит хотя бы одну нечастую подпоследовательность, то удаляем её.

   Эти два шага пользуются вспомогательными утверждениями, чтобы улучшить работу алгоритма.

   Утверждение для шага (а). Частая последовательность состоит из частых подпоследовательностей.

Утверждение для шага (b). Если последовательность S содержит хотя бы одну нечастую подпоследовательность, тогда последовательность S не может быть частой.

Доказательство утверждений тривиальны: если последовательность s не встречается в истории n раз, то её надпоследовательность S не может содержать последовательность s хотя бы n раз.

На рисунке 2а и 2г показаны результаты этапа генерации кандидатов для 2 и 3 итерации.

3. Отбор кандидатов.

   (a) Сканируем историю подсчитывая поддержку для каждого кандидата из Ck.

   (b) Если поддержка кандидата больше минимальной, то добавляем его в L.

   Результаты отбора кандидатов на второй итерации показаны на рисунке 2б и 2в.

4. Проверка на окончание работы алгоритма.

   (a) Если на предыдущем шаге не добавлена ни одна последовательность, тогда завершаем работу алгоритма.

   (b) Иначе возвращаемся к шагу 2.

   На нашем примере окончание сбора информации происходит на третьей итерации, так как, как показывает рисунок 2г, ни один кандидат не будет добавлен в множество частых последовательностей L, потому что каждый из них имеет нечастую подпоследовательность. Таким образом, результаты, представленные на рисунке 2в, будут результатами этапа сбора информации.

Кандидаты С2	Нечастая последовательность в кандидате
1, 4	-
1, 7	-
4, 1	-
4, 7	-
7, 1	-
7, 4	-

а)

Кандидаты С2	Поддержка
1, 4	2
1, 7	3
4, 1	0
4, 7	1
7, 1	0
7, 4	1

б)

Последовательность	Поддержка
1	3
4	3
7	4
1, 7	3

в)

Кандидаты С3	Нечастая последовательность в кандидате
1, 4, 7	1, 4
1, 7, 4	1, 4
4, 1, 7	4, 1
4, 7, 1	4, 7
7, 1, 4	7, 1
7, 4, 1	7, 4

г)

Рис. 3: а) генерация кандидатов С2; б) отбор кандидатов С2; в) множество частых последовательностей L; г) Генерация кандидатов С3.

3.1.3. Составление правил

С предыдущего этапа имеем множество L, в котором хранятся последовательности разной длины. Это и будет множеством правил: последовательность $(lba_1, lba_2, ..., lba_n)$ есть правило $lba_1, ..., lba_{n-1} \rightarrow lba_n$.

3.2. Режим работы

Для режима работы создается новый модуль prefetcher (упреждающее чтение). Он является независимой частью кэш памяти, то есть имеет свои независимые структуры.

Множество L становится множеством правил, которые prefetcher умеет интерпретировать.

Prefetcher имеет дополнительный буфер локальной истории, в котором хранятся последние адреса запросов. Он имеет структуры FIFO и содержит количество адресов, равное размеру окна.

Работа модуля:

- Если пришел запрос на чтение:
  - Добавляем адрес в буфер (рисунок 4а).

– Проверяем буфер на наличие правил. Если правило найдено, то производим предварительную загрузку (рисунок 4б).
– Если адрес отсутствует в prefetcher, то ничего не делаем.
– Если адрес есть в памяти этого модуля, то читаем из него (рисунок 4в).

- Если пришел запрос на запись:
  – Если адрес не содержится в prefetcher, то ничего обновлять не нужно – никакие действия не производятся.
  – Если адрес содержится в модуле, то обновляем данные по этому адресу.

Проверка на наличие адреса происходит в модуле упреждающего чтения и в префетчере.

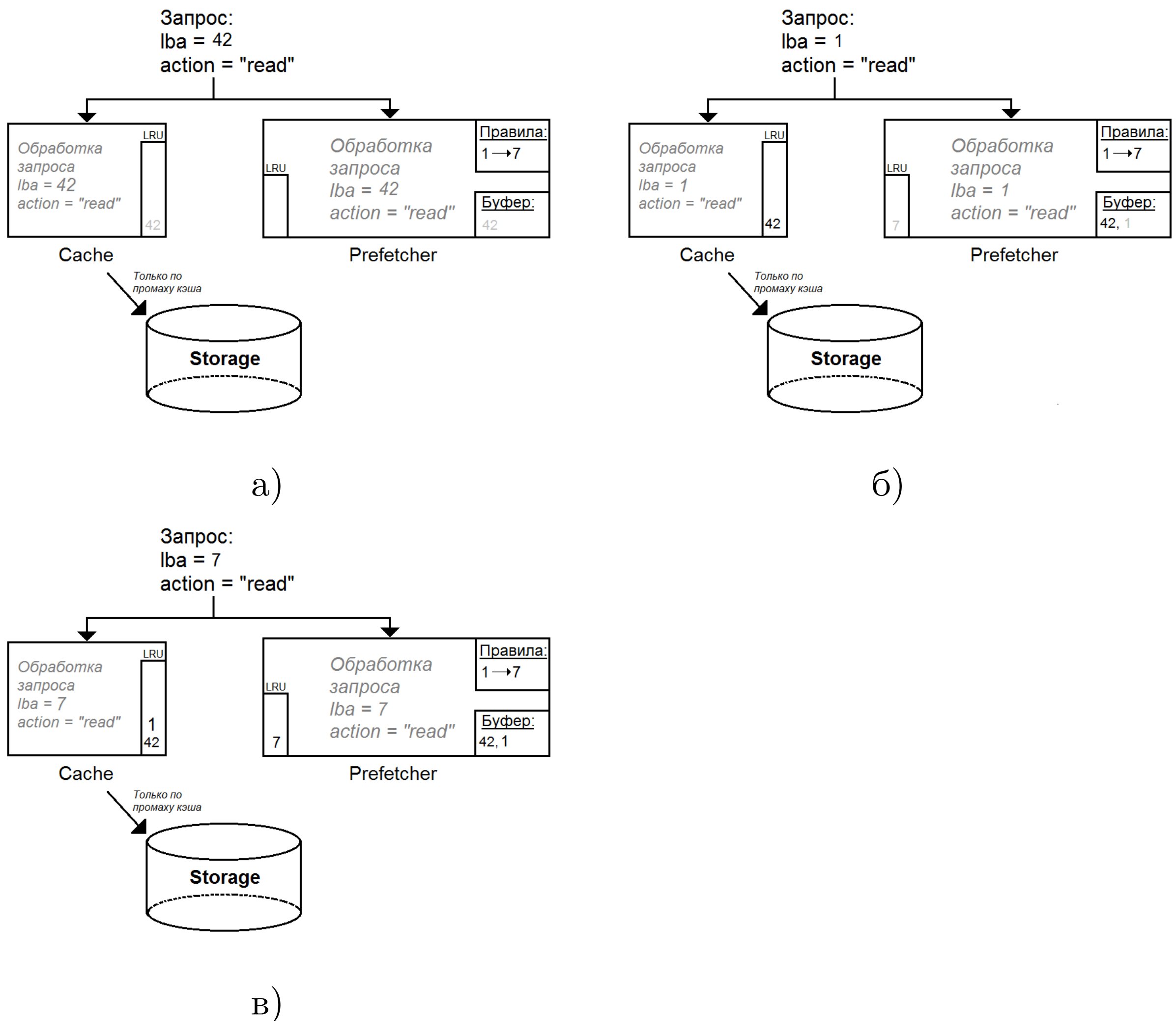
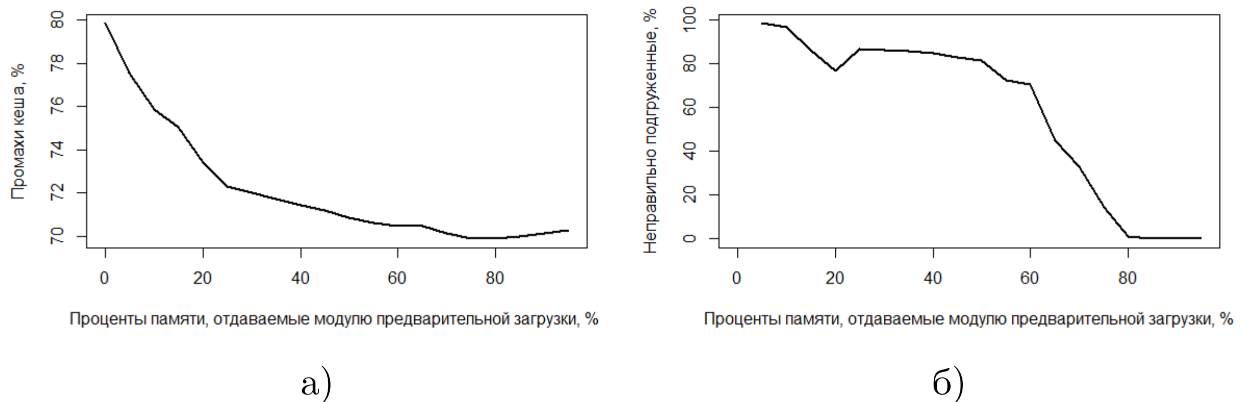


Рис. 4: Некоторые примеры работы модели.

4. Результаты

Тестирование модели было проведено на разых трейсах. Для этого использовались метрики:

- промахи кэша
- промахи модуля предварительной загрузки
- количество неправильно предварительно загруженных элементов.



а) б)

Рис. 5: Результаты для теста в 10^5 запросов и памятью кэша в 10^3 записей.

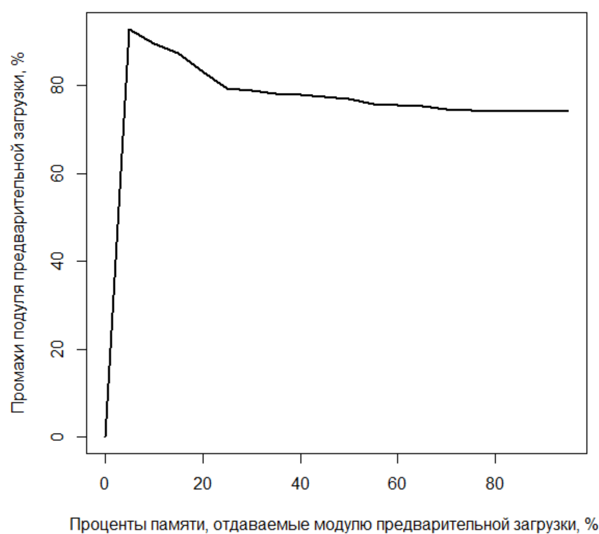


Рис. 6: Зависимость промахов в модуле предварительной загрузки для теста в 10^5 запросов и памятью кэша в 10^3 записей.

Из рисунка 5 видно, что можно достичь выигрыш в промахах кэша в 10%, при этом процент неправильно подгруженных блоков будет убывать.

На рисунке 6 показано, что процент промахов в модуле предварительной загрузки достаточно велик, но тем не менее убывает. Такой высокий процент обоснован тем, что минимальная поддержка для данного теста была выбрана низкой – 10% от количества окон. По этой причине было найдено неоптимальное количество правил для данного случая. Т.е. процент промахов в модуле предварительной загрузки зависит от соотношение количества найденых правил и размера этого модуля.

Таким образом, было установлено, что варьируя параметры минимальной поддержки, общего размера кэша и процента памяти отдаваемого модулю предварительной загрузки, можно найти оптимальные состояние, при котором процент, отдаваемый модулю предварительной загрузки, и процент промахов кэша будут минимальными.

Заключение

В ходе данной работы были достигнуты следующие результаты:

- Изучен принцип работы существующих алгоритмов предварительной загрузки.
- Разработан алгоритм для СХД с блочных доступом.
- Реализована модель алгоритма.
- Проведено функциональное и нагрузочное тестирование реализованного модуля.

Список литературы

[1] Chetna Chand Amit Thakkar Amit Ganatra. Sequential Pattern Mining: Survey and Current Research Challenges.

[2] Gill Binny S., Modha Dharmendra S. SARC: Sequential Prefetching in Adaptive Replacement Cache.

[3] J.Nesbit Kyle, E.Smith James. Data Cache Prefetching Using a Globla History Buffer.

[4] Mabroukeh Nizar R., C.I.Ezeife. Taxonomy of Sequential Pattern Mining Algorithms.

[5] Pranav Pathak Mehedi Sarwar Sohum Sohoni. Markov Prediction Scheme for Cache Prefetching.

[6] R. Agrawal R. Srikant. Fast Discovery of Association Rules.

[7] Xifeng Yan Jiawei Han Ramin Afshar. CloSpan: Mining Closed Sequential Pattern in Large Datasets.

[8] Zhenmin Li Zhifeng Chen Sudarshan M. Srinivasan, Zhou Yuanyuan. C-Miner: Mining Block Correlations in Storage Systems.