Московский государственный университет М.В.Ломоносова
Факультет вычислительной математики и кибернетики
Кафедра системного программирования

Дипломная работа

Эффективные методы замещения страниц в кэше XML СУБД

Автор:
гр. 528
Сиващенко Дмитрий Витальевич

Научные руководители:
ак. РАН проф. Иванников Виктор Петрович,
м. н. с. ИСП РАН Таранов Илья Сергеевич

Москва 2012

Аннотация

Эффективные методы замещения страниц
в кэше XML СУБД

Сиващенко Дмитрий Витальевич

В этой работе рассмотрена возможность применения в XML СУБД Sedna некоторых известных алгоритмов замещения страниц в кэше СУБД. Также были предложены и реализованы алгоритмы замещения, учитывающие особенности XML СУБД Sedna. В ходе работы проведено сравнение эффективности работы алгоритмов замещения на различных наборах тестов.

Abstract

Advanced strategies for buffer page replacement
in cache of Native XML DBMS

Dmitry Sivashchenko

In this paper we consider the possibility of using some well-known algorithms for page replacement in XML database Sedna cache. We also proposed and implemented replacement algorithms, which taking into account the features of XML database Sedna. We carried out this work to compare the effectiveness of the replacement algorithms on different sets of tests.

Содержание

1 Введение 4

1.1 Аномалия Белади 6

1.2 Оптимальный алгоритм (OPT) 8

2 Постановка задачи 9

3 Обзор существующих решений 10

3.1 LRU 10

3.2. CLOCK 11

3.3 LFU 12

3.4 GCLOCK 13

3.5 LRD 14

3.6 MQ 15

3.7 LIRS 17

4 Исследование и построение решения задачи 19

4.1 Особенности сериализации поддеревьев документов СУБД Sedna. 20

4.2 Типизация страниц в кэше СУБД Sedna 25

5 Описание практической части 27

6 Заключение 33

Список литературы 35

Приложение А. Список использовавшихся в сценариях тестирования запросов. 37

1 Введение

Операционные системы и СУБД используют внешние устройства (например, магнитные диски) для продолжительного хранения больших объемов данных. Такие устройства имеют низкую стоимость хранения одного бита информации и являются энергонезависимыми, что делает их незаменимыми в своей области применения. Однако операции с данными могут производиться лишь в том случае, когда они находятся в оперативной памяти компьютера (верно для большинства операционных систем).

Скорость доступа к оперативной памяти на несколько порядков превышает скорость доступа к внешним устройствам, в то же время оперативная память способна хранить лишь ограниченный объем данных (в свою очередь на несколько порядков меньше доступного объема данных на внешних устройствах). Таким образом, возникает проблема эффективного управления оперативной памятью, решением которой занимаются как операционные системы, так и СУБД.

Несмотря на то, что большинство операционных систем поддерживают специальный кэш в оперативной памяти для размещения там элементов файловой системы, такой кэш не подходит для использования его СУБД: основной причиной является отсутствие у операционной системы информации о внутренних принципах работы конкретной СУБД. Как следствие, при необходимости поместить в оперативную память новую порцию данных, операционная система не в состоянии определить, за счет каких из уже находящихся в памяти данных рациональнее всего было бы освободить место. Поэтому СУБД обычно решают проблему эффективного управления памятью (кэшем) самостоятельно. Компонент СУБД, ответственный за решение данной проблемы, называется менеджером буферов[1][2].

Далее под кэшем (буфером) будет подразумеваться память, отведенная СУБД, а не вся оперативная память в целом.

В виду особенностей аппаратной реализации чтения данных с внешних устройств (или их записи), данные БД, хранящиеся на внешнем устройстве, делятся на страницы (блоки) одинакового размера (обычно 512 и более байт). Для простоты отображения данных кэш состоит из

страниц того же размера, объем кэша (в страницах) является настраиваемым параметром СУБД. Работа с внешними устройствами (чтение и запись) происходят постранично, таким образом минимальный объем данных, который может быть считан с внешнего устройства (или записан на него) за одну операцию равен размеру одной страницы.

Логическим обращением называется произвольное обращение к некоторой странице, независимо от того, находится эта страница в кэше или нет. Для начала работы с произвольными данными необходимо, чтобы содержащие эти данные страницы были подгружены в кэш. При каждом обращении к произвольным данным (то есть логическом обращении к страницам, содержащим эти данные) менеджер буферов производит следующую последовательность действий:

1. Производится поиск требуемой страницы в кэше. В случае если страницы там не обнаружено, требуется загрузка этой страницы в кэш с внешнего устройства хранения данных. Каждое обращение к странице с данными на внешнем устройстве называется физическим обращением. Чтобы поместить найденную страницу в кэш, необходимо освободить место, выгрузив одну из ранее загруженных страниц (в случае, если кэш уже полон). Кандидаты на замещение определяются с помощью специального алгоритма, являющегося частью функционала менеджера буферов. Если замещаемая страница была изменена в процессе работы с ней, то производится ее запись на внешнее устройство хранения.

2. Найденная страница фиксируется на время работы с ней для предотвращения её замещения.

3. Фиксируется факт логического обращения к странице. Данное действие не является обязательным, однако для работы большинства алгоритмов замещения требуются те или иные сведения об обращениях к страницам, например общее число обращений к каждой странице, время последнего обращения и т.д.

4. Адрес страницы передается инициатору запроса среди компонентов СУБД.

После завершения работы со страницей, инициатор запроса к данным передает менеджеру буферов адрес этой страницы для снятия ограничения на возможность удаления этой страницы из кэша.

Пусть все страницы данных БД пронумерованы. Последовательность логических обращений к страницам, записанная в виде номеров адресуемых страниц, называется строкой логических обращений к страницам.

Строка логических обращений за выделенный промежуток времени определяется[1]:

- типами запросов, обработанных СУБД (update, select и т.п.).
- наличием параллельных транзакций и их количеством.
- структурами СУБД, использующимися для определения номера необходимой страницы (например, индексные B+ деревья и т.п.).

Одной и той же логической строке обращений могут соответствовать разные строки физических обращений к страницам в зависимости от размеров кэша и используемым менеджером буферов алгоритмом замещения страниц. В силу дороговизны (по времени) обращения к внешнему устройству хранения данных, одной из основных задач менеджера буферов является минимизация длины строки физических обращений к страницам данных для заданного объема кэша.

Задача выбора очередной страницы для замещения в рамках менеджера буферов решается введением специального алгоритма, называемого алгоритмом замещения (вытеснения). Предложим его формальное определение.

Пусть E – множество страниц в кэше СУБД, тогда алгоритм замещения φ определяется как $\varphi : (E, \delta) \rightarrow p$, $p \subset E$, где p – страницы, выбранные для замещения, δ - представляет дополнительные данные о страницах, которые использует алгоритм замещения.

1.1 Аномалия Белади

Одним из простейших решений для определения замещаемой страницы является алгоритм FIFO (First in, First Out), представляющий

множество страниц в кэше в виде однонаправленной очереди. *Аномалия Белади*[3] показывает несостоятельность интуитивного представления о том, что при увеличении объема кэша эффективность работы алгоритма возрастает.

Рассмотрим следующий пример.

Пусть есть кэш размером 3 страницы в первом случае и 4 – во втором (см. Рисунок 1 и 2 соответственно). Пусть страницы пронумерованы от 0 до 4, тогда первые строчки приведённых таблиц – последовательности логических обращений к страницам данных, идентичные для обоих рассматриваемых случаев. Кэш организован по принципу FIFO очереди. Как можно видеть, для кэша размером 3 число необходимых замещений (14) оказалось меньше, чем для кэша размером 4 (необходимо 15 замещений).

Таблица 1: Последовательность обращений к страницам и содержимое FIFO очереди для размера очереди равного 3 страницам.

Sequence of pages	**3**	**2**	**1**	**0**	**3**	**2**	**4**	**3**	**2**	**1**	**0**	**4**	**2**	**3**	**2**	**1**	**0**	**4**
FIFO	3	2	1	0	3	2	4	4	4	1	0	0	2	3	3	1	0	4
		3	2	1	0	3	2	2	2	4	1	1	0	2	2	3	1	0
			3	2	1	0	3	3	3	2	4	4	1	0	0	2	3	1
Hit/Miss	**M**	**M**	**M**	**M**	**M**	**M**	**M**	**H**	**H**	**M**	**M**	**H**	**M**	**M**	**H**	**M**	**M**	**M**

Таблица 2: Последовательность обращений к страницам и содержимое FIFO очереди для размера очереди равного 4 страницам.

Sequence of pages	**3**	**2**	**1**	**0**	**3**	**2**	**4**	**3**	**2**	**1**	**0**	**4**	**2**	**3**	**2**	**1**	**0**	**4**
FIFO	3	2	1	0	0	0	4	3	2	1	0	4	4	3	2	1	0	4
		3	2	1	1	1	0	4	3	2	1	0	0	4	3	2	1	0
			3	2	2	2	1	0	4	3	2	1	1	0	4	3	2	1
				3	3	3	2	1	0	4	3	2	2	1	0	4	3	2
Hit/Miss	**M**	**M**	**M**	**M**	**H**	**H**	**M**	**M**	**M**	**M**	**M**	**M**	**H**	**M**	**M**	**M**	**M**	**M**

Таким образом, в случае использования FIFO нельзя решить проблему оптимального управления кэшем простым увеличением его размера – актуальна задача поиска эффективного алгоритма замещения, лишенного описанного недостатка.

1.2 Оптимальный алгоритм (OPT)

Для оценки эффективности работы алгоритмов замещения в некоторых случаях может быть использован так называемый оптимальный алгоритм. Оптимальный алгоритм[1][2] (он же OPT, алгоритм Белади или «проницательный» алгоритм) работает следующим образом: при необходимости замещения страницы в кэше замещается та страница, которая дольше всего не понадобится в будущем. Данный алгоритм, как правило, не может быть реализован по причине невозможности на практике определить, какие именно страницы в каждый момент времени будут дольше всего неиспользуемыми. Исключения составляют те случаи, в которых заранее известно, к каким страницам и в каком порядке будут обращения и, следовательно, данную информацию можно подвергнуть анализу с последующим построением оптимального алгоритма.

Концепция OPT теоретически может быть использована для оценки реально используемых алгоритмов в качестве «верхней границы» эффективности.

Возможность применения того или иного алгоритма замещения страниц в менеджере буферов СУБД во многом зависит от следующих факторов:

- Размер кэша СУБД (в страницах).
- Размер страниц данных.
- Характер и интенсивность выполняемых транзакций.
- Ограничения на характеристики самого алгоритма замещения: скорость работы, используемый объем памяти и т.п.

Для большинства алгоритмов замещения страниц существуют «плохие данные» - последовательность обращений к страницам, на которой эффективность алгоритма существенно снижается. В качестве примера можно привести цикличные обращения к странице с периодом, большим числа страниц в кэше, для алгоритмов LRU, пиковые нагрузки (т.е. большое количество обращений к странице за малый промежуток времени) для GLOCK и LRD, *Аномалию Белади* для FIFO и т.п. Такие особенности также следует учитывать при выборе алгоритма для СУБД.

2 Постановка задачи

Цель исследования данной дипломной работы - реализация эффективного алгоритма замещения для XML СУБД на примере XML СУБД Sedna, разработанной в рамках Института Системного Программирования РАН.

Задачи, поставленные в данной работе для достижения цели:

1. Реализация известных алгоритмов замещения страниц LRU, CLOCK, GCLOCK (с различными модификациями) для СУБД Sedna.

2. Разработка и реализация алгоритмов, учитывающих специфичные свойства XML СУБД (Sedna), опирающихся на классификацию страниц в кэше по типу хранимых ими данных.

3. Сравнение эффективности работы реализованных алгоритмов.

3 Обзор существующих решений

Понятие *менеджера буферов* как компоненты СУБД является общим для XML и реляционных СУБД. Принципы, положенные в основу работы алгоритмов замещения, предназначенных для реляционных СУБД, могут быть применены и в случае СУБД, изначально спроектированной для хранения XML данных.

Существует множество принципиальных подходов к стратегии замещения страниц в кэше. Более того, алгоритмы, реализующие тот или иной подход, могут обладать большим числом модификаций с различиями в выборе структур данных, ориентированностью на конкретную СУБД и т.п. Перейдем к рассмотрению основных и наиболее ярких представителей алгоритмов замещения.

3.1 LRU

Алгоритм LRU[2] – least recently used (дословно: «наименее недавно использовавшийся») является одним из наиболее распространенных в силу своей наглядности и простоты в понимании. Суть этого алгоритма в том, что при помещении любой страницы в кэш ей в соответствие ставится некоторое текущее значение времени. При необходимости освободить место в кэше для новой страницы, среди всех страниц, находящихся в памяти на данных момент, выбирается та, которая имеет наибольший возраст (т.е. дольше всего не обновлялась).

В качестве текущего времени для страниц данных может быть использовано значение глобального счетчика, увеличивающегося при каждом обращении к произвольной странице.

Как можно заметить, для работы алгоритма необходимо хранить n+1 значений времени, где n – размер кэша в страницах. Временная сложность его работы зависит от реализации: она линейна относительно размера кэша в случае, когда выбор страницы для замещения производится путем сравнения временных меток всех страниц кэша. Если же страницы в кэше организованы в очередь и при каждом обращении целевая страница переставляется в начало такой очереди, сложность такого алгоритма - O(1) относительно числа обращений к страницам.

LRU не учитывает ситуации, когда некоторые из страниц используются с определенным периодом, превышающим размер кэша. Таким образом, между двумя соседними обращениями к себе эти страницы будут успевать покинуть кэш (например, страницы, содержащие индексы; страницы, являющиеся сами по себе целью некоторых повторяющихся транзакций и т. п.). Данный недостаток в некоторых ситуациях может приводить к существенному возрастанию числа физических обращений.

Примером СУБД, использующей LRU в качестве алгоритма замещения, является СУБД MySQL[4].

3.2. CLOCK

Алгоритм CLOCK представляет все страницы в кэше в виде замкнутого однонаправленного списка. В соответствие каждой странице ставится битовый флаг (page reference bit), принимающий два значения (например, значения 0 и 1). При попадании страницы в кэш, битовый флаг устанавливается в значение 0 и страница помещается в хвост списка. В случае обращения к странице, ее битовый флаг устанавливается в значение 1. При необходимости выталкивания страницы из кэша, производится цикличный просмотр списка страниц в кэше, начиная с текущей страницы (header page). Если битовый флаг очередной страницы равен 1, то его значение меняется на 0 и страница помещается в хвост списка, если же страница уже имеет битовый флаг равный 0 – она считается искомой и выталкивается. Следующая за ней страница в списке становится текущей.

Для замещения страницы, к которой только что было обращение (или страницы, которая только что была помещена в кэш) указателю на текущую страницу нужно будет 2 раза полностью обойти список, при условии, что за это время исходная страница не получит обращений к себе. Таким образом, по сравнению с LRU, CLOCK вдвое увеличивается максимальный период обращений к странице, чтобы она могла оставаться в кэше.

Сложность CLOCK в худшем случае – n + 1, где n – размер кэша.

К достоинствам алгоритма стоит отнести его относительную простоту реализации.

Существует множество алгоритмов, использующих организованные в виде CLOCK или LRU списки страниц в качестве основы при построении более сложных моделей замещения. Примерами таких алгоритмов могут служить CAR (CLOCK with Adaptive Replacement) / CART[5], ARC (Adaptive Replacement Algorithm), MQ (будет описан далее) и другие.

Алгоритмы CAR и CART определяют отдельные CLOCK очереди для страниц с краткосрочной и долгосрочной полезностью и LRU очереди для буферов истории страниц. *Буфер истории страниц* - список страниц, в данный момент отсутствующих в кэше, но побывавших там относительно недавно. Алгоритм ARC использует тот же принцип разделения, что и CAR / CART, организуя все списки страниц по принципу LRU.

3.3 LFU

LFU[1] - least frequently used («наименее часто использовавшийся»). Суть алгоритма: для каждой странице в кэше вводится так называемый счетчик обращений - reference counter (далее RC). При попадании страницы в кэш, её RC устанавливается в значение 1. При последующих обращениях к странице, её RC каждый раз увеличивается на 1. Как только возникает необходимость выталкивания страницы из кэша, выбирается страница с RC, имеющая минимальное значение.

LFU прост и нагляден, но очевидные недостатки исключают его возможность применения на практике. Серьезным недостатком является, например, критичность LFU к ситуации, когда к одной странице за короткий промежуток времени происходит большое число обращений. RC такой страницы быстро набирает большое значение, что приводит к зависанию страницы в кэше и, как следствие, засорению кэша практически не использующимися страницами. Через некоторое время работы алгоритма LFU с высокой степенью вероятности возникает ситуация, когда одна из страниц в кэше постоянно замещается, а остальные, набравшие большие RC, остаются. Также LFU никак не учитывает продолжительность нахождения страницы в кэше.

Существует множество модификаций LFU, серьезно повышающих его эффективность и использующихся в реальных системах. Перейдем к рассмотрению одной из них.

3.4 GCLOCK

GCLOCK[1] (generalized CLOCK)– результат совмещения принципов работы алгоритмов CLOCK и LFU. Подобно CLOCK, в GCLOCK используется замкнутый односторонний список страниц. Подобно LFU, каждой странице ставится в соответствие счетчик обращений – reference counter - RC. При помещении страницы в кэш её RC инициализируется значением 1, при обращении к странице её RC инкрементируется.

В отличие от CLOCK, для выбора замещаемой страницы может понадобиться более одного полного обхода списка страниц, ведь теперь вместо битового флага, принимавшего значения 0 и 1, используется неотрицательный RC. При обходе списка алгоритм действует по аналогии с CLOCK – если очередная страница имеет RC больший 0, уменьшаем его на 1 и помещаем страницу в конец списка. Замещаем первую встретившуюся страницу с RC равным 0.

Сложность GLOCK линейна относительно размера кэша - работу алгоритма по замещению страницы можно упростить, сведя ее к нахождению первой страницы (после текущей), имеющей минимальный RC, замещению этой страницы и дальнейшему уменьшению значений всех RC страниц в кэше на величину найденного минимального RC.

Дополнительно усовершенствовать GCLOCK можно, введя типизацию страниц в кэше[1]:

Поставим в соответствие типам i страниц веса F(i) и R(i), где

F(i) – значение, которое получает RC страницы типа i при её помещении в кэш,

R(i) – значение, на которое увеличивается RC при обращении к странице в кэше.

Приведем некоторые варианты (V_1, V_2) использования таких весов:

V_1: при инициализации $RC(i) = F_i$,

при каждом последующем обращении $RC(i) = RC(i) + R_i$.

V_2: при инициализации $RC(i) = F_i$,

при каждом последующем обращении $RC(i) = R_i$.

Очевидно, что если $R_i = F_i = 1$ для любого i, то V_2 обращается в простой CLOCK.

Возможность устанавливать разным типам страниц разные начальные веса а также разную скорость набора RC делает алгоритм гибким, особенно в случае типизации страниц для какой-то конкретной системы. В общем же случае можно типизировать, например, страницы с данными и страницы, содержащие какие-либо вспомогательные структуры данных (фрагменты индексов, служебную информацию и прочее).

Также одним из вариантов улучшение алгоритма GCLOCK является ограничение максимально возможного значения RC для страниц.

В качестве примера СУБД, использующей GCLOCK в качестве алгоритма замещения, можно привести PostgreSQL[6] (с ограничением максимального RC страниц равным 5).

3.5 LRD

LRD[1] (least reference density) представляет собой усовершенствованную версию LFU. Помимо RC (reference counter), для страниц в кэше вводятся следующие понятия:

- GRC – Global Reference Counter – глобальный счетчик обращений к станицам, инкрементируется при каждом запросе.
- FC – First reference Counter – номер запроса (значение GRC), в момент которого страница была помещена в кэш.

Возрастом i-ой страницы назовём разность вида GRC – FC(i), несущую в себе информацию о продолжительности нахождения i-ой страницы в кэше. Возраст страницы в дальнейшем будет учитываться при выталкивании.

При необходимости выталкивания страницы из кэша, для каждой i-ой страницы в кэше вычисляется её плотность ссылок на страницу - reference density – RD как отношение числа обращений к странице к её

возрасту: RD(i) = RC(i) / (GRC – FC(i)), GRC – FC(i)>=1. Выталкивается страница, имеющая минимальное значение RD.

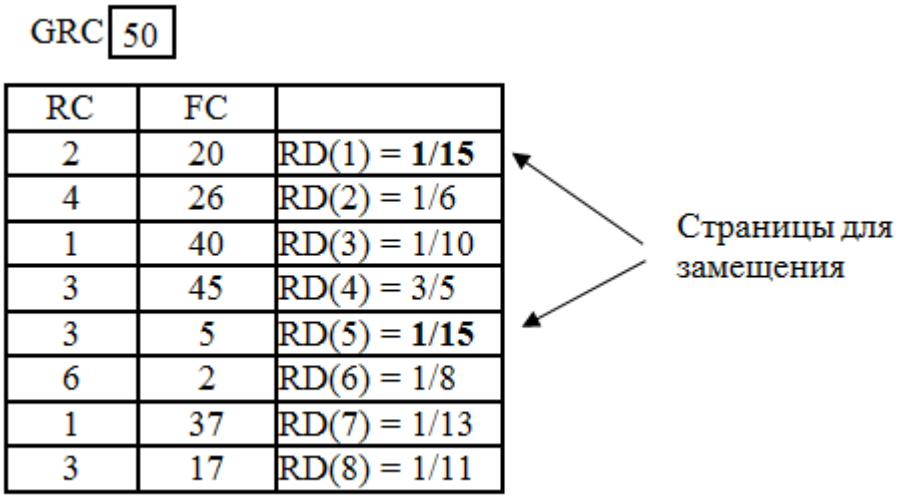

Рисунок 1: Пример работы алгоритма LRD.

Проблема LFU, когда страницы, быстро набиравшие большие значения RC, надолго оставались в кэше, для LRD нехарактерна.

3.6 MQ

MQ[7] (multiple queue) для своей работы использует несколько списков-очередей страниц Q_0, Q_1, … , Q_n, находящихся в кэше. Каждая из очередей построена по принципу LRU. Каждая страница имеет свой RC, по принципу действия аналогичный RC в предыдущих алгоритмах. Также MQ использует так называемый буфер истории - список страниц, которые в данный момент не находятся в кэше, но которые побывали там относительно недавно (организован также по принципу LRU). Для страниц в буфере истории RC также сохраняется.

Для определения номера i списка Q_i, в котором страница должна находиться в кэше, используется специальная функция F отображения RC страницы на номер i списка в кэше. Функция F должна быть неубывающей и принимать лишь целые неотрицательные значения на натуральных числах.

При обращении к странице, ее RC инкрементируется, после чего определяется номер i как F(RC(page)), и страница (если это необходимо) перемещается в список Q_i. В качестве примера функции F можно

использовать $F = \lceil \log_2(x) \rceil$. Таким образом, страницы, к которым производятся обращения, будут подниматься по иерархии списков Q_i тем выше, чем интенсивнее будут обращения. Q_i будут создаваться/исчезать по мере передвижения страниц.

При очередном обращении к странице возможны 3 ситуации:

1. Искомая страница находится в кэше, тогда для нее вычисляется значение F и производится ее перемещение в начало соответствующего списка (даже если страница не поменяла номер списка). Пример: (здесь и далее в качестве F берем $F = \lceil \log_2(t) \rceil$) страница находится в списке Q_3 и имеет RC = 5. Тогда ее RC становится равным 6, $F = \lceil \log_2(6) \rceil = 3$, страница остается в списке Q_3, но перемещается в его конец (согласно принципам LRU). Если бы RC страницы стал бы равен 9, то $F = \lceil \log_2(9) \rceil = 4$, и страница переместилась бы в начало списка Q_4.

2. Искомая страница находится в буфере истории. Тогда, согласно принципу LRU, помещаем страницу из списка Q_i с минимальным i в историю, а исходную страницу x помещаем в кэш аналогично предыдущему случаю.

3. Искомая страница отсутствует в кэше и её нет в буфере истории, в этом случае перемещаем одну из страниц в кэше в буфер истории, а из последнего по принципу LRU также удаляем страницу. Далее устанавливаем RC(x) = 1 и помещаем ее в кэш, действуя так же, как в пункте 1.

При описанной выше схеме работы, у алгоритма появляется недостаток, присущий алгоритму LFU – страницы с высоким RC могут перестать использоваться, но при этом останутся «висеть» в кэше. Для борьбы с этим явлением в MQ введено понятие expiredTime, связанное с каждой страницей в кэше. При помещении страницы в кэш, expiredTime = currentTime + lifeTime (параметр системы). При каждом запросе для всех страниц проверяется currentTime > expiredTime. При выполнении условия, страница перемещается в начало нижележащего списка (если страница находится в Q_0 – перемещаем в конец Q_0).

MQ хорошо работает со страницами, к которым происходят периодические обращения, за счет накопления значения RC и

продвижения таких страниц в иерархии LRU очередей наверх. Более того, даже если страница была вытеснена из кэша, MQ некоторое время сохраняет статистику страницы в буфере истории, что повышает эффективность работы алгоритма в целом.

3.7 LIRS

Алгоритм LIRS[8] - Low Inter-reference Recency Set – вводит для каждой страницы p следующие понятия:

IRR(p) – Inter-Reference Recency - количество уникальных обращений к страницам между двумя последними обращениями к x. Может быть неопределено.

R(p) – количество уникальных обращений к страницам с момента последнего обращения к p.

Приведенные значения пересчитываются для всех страниц при каждом запросе.

Рисунок 2: Понятия IRR(x) и R(x) для страницы (алгоритм LIRS).

Основная идея алгоритма: если IRR страницы высок, он вероятнее всего, будет высок и далее.

Все страницы делятся на два типа: LIR (Low IRR) страницы и HIR (High IRR) страницы. Все LIR страницы должны содержаться в кэше, и лишь небольшая часть кэша содержит HIR страницы. Количество LIR страниц является настраиваемым параметром системы и, естественно, не может превышать размер кэша. Страницы являются HIR-страницами по умолчанию и получают статус LIR динамически в зависимости от интенсивности обращения к ним.

В случае необходимости освободить место в кэше, для замещения выбирается HIR-страница p, удовлетворяющая следующему условию:

R(p) >= max(IRR(h) , R(h)), h Є HIR. Для вновь помещенной в кэш страницы k производится сравнение со всеми LIR-страницами: если среди LIR-страниц найдется такая, что её R(k) > IRR(k), то эта страница теряет статус LIR и становится HIR-страницей, в то время как страница y становится LIR. Таким образом, в кэше постоянно поддерживаются страницы с высоким показателем IRR, что делает работу алгоритма LIRS эффективной в случае цикличных запросов к некоторым страницам, а также учитывает «возраст» страниц.

К недостаткам LIRS относятся сложная реализация, а также большое число вычислений (требует пересчета IRR и RC для каждой страницы при каждом запросе).

4 Исследование и построение решения задачи

В рамках данной дипломной работы мы будем использовать XML СУБД Sedna[9]. Sedna — изначально спроектированная для хранения и обработки XML-данных СУБД. Разработана в рамках Института Системного Программирования РАН.

Рассмотрим архитектуру Sedna[10].

Управляющий процесс - Governor - главная компонента, поддерживающая жизненный цикл СУБД в целом, а также взаимодействие остальных компонент между собой. Управляющий процесс контролирует функционирующие на данный момент менеджеры баз данных (Storage Manager), создаёт и обеспечивает подключения (сессии) локальных клиентов к базам.

Каждый менеджер базы данных отвечает за поддержку одной базы данных и включает в себя интересующий нас менеджер буферов (Buffer Manager).

Сессия – компонента Sedna, занимающаяся интерпретацией, оптимизацией и исполнением поступающих xQuery запросов. Каждая сессия привязана к соответствующему ей менеджеру базы данных.

В силу особенностей реализации Sedna, менеджер буферов не располагает информацией об обращениях к страницам в рамках тех или иных сессий. Таким образом, при внедрении нового алгоритма в СУБД Sedna в общем случае модифицируются две компоненты – сессия и менеджер буферов (как часть менеджера базы данных). В компоненту сессия необходимо добавить функционал учёта обращений к страницам, в менеджер буферов – целевой алгоритм замещения.

За обращение к странице P мы будем считать смену сессией текущей страницы L, с которой производится работа, на новую страницу P, отличную от L.

В рамках данной работы мы не рассматриваем возможность реализации алгоритма MQ для Sedna. Так как менеджер буферов не контролирует обращения к страницам со стороны сессий, для реализации этих алгоритмов возникла бы необходимость в организации

сложных структур данных (иерархия LRU очередей для MQ) в межпроцессном пространстве. Задача по созданию таких структур данных трудна и выходит за рамки данной работы.

При выборе алгоритмов для Sedna мы отдавали предпочтение алгоритмам замещения, достаточно простым в реализации. Например, такие алгоритмы как CLOCK, GCLOCK при учёте обращения к страницам обладают свойством локальности[1]. Для LRU и LRD потребуется наличие единственной общей переменной (представляющей текущее «время» обращений), разделяемой всеми транзакциями и менеджером буферов. Реализация подобной переменной сложности не представляет.

4.1 Особенности сериализации поддеревьев документов СУБД Sedna.

В Sedna используется специальная стратегия хранения XML документов с использованием *описывающей схемы*[10]. В отличие от статичных схем структур документов (таких как DTD и XML Schema), описывающая схема для документа строится и поддерживается динамически. Соответственно, принцип описывающих схем применим к произвольным входным документам, даже при отсутствии у них схемы, и обладает большей гибкостью по сравнению со статичными схемами.

Схема структуры документа служит отправной точкой для любых операций над документом, таких как извлечение данных или их обновление. Рассмотрим пример описывающей схемы для следующего XML документа:

<library>
<book>
<title>Foundations of Databases</title>
<author>Abiteboul</author>
<author>Hull</author>
<author>Vianu</author>
</book>
<book>

[1] То есть для обработки обращения к странице не требуется информация о других страницах. Например, CLOCK и GCLOCK используют соответственно только битовый флаг и счетчик обращения целевой страницы.

<title>An Introduction to Database Systems</title>
<author>Date</author>
<issue>
<publisher>Addison-Wesley</publisher>
<year>2004</year>
</issue>
</book>
...
<paper>
<title>A Relational Model for Large Shared Data Banks</title>
<author>Codd</author>
</paper>
</library>

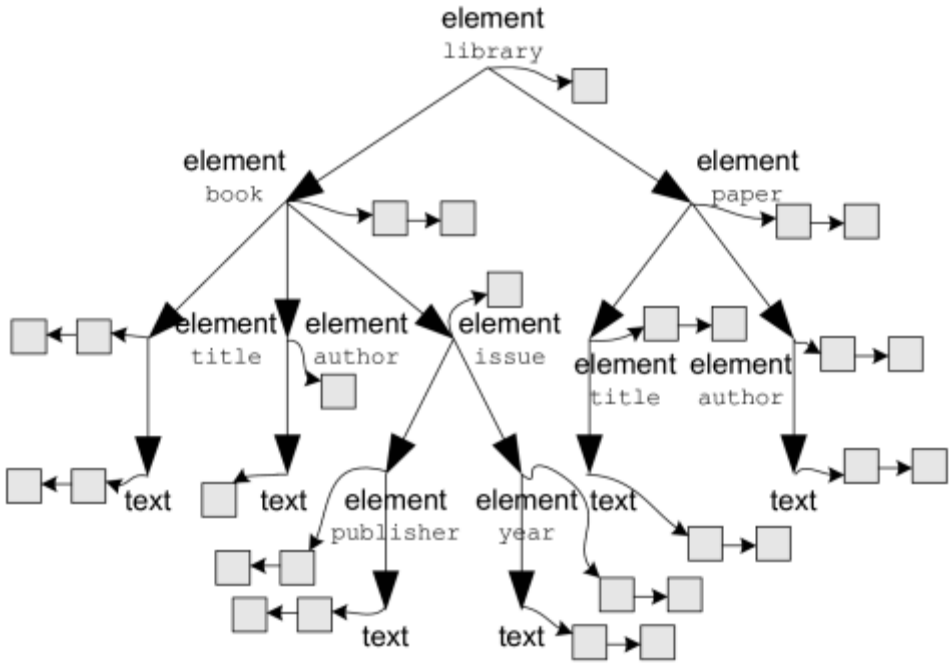

Рисунок 3: Пример описывающей схемы для XML документа.

Описывающая схема документа представляет собой дерево, иерархия узлов которого отражает структуру целевого XML документа. Каждый узел схемы имеет ссылку на страницы, в которых содержатся соответствующего типа узлы документа. Страницы, содержащие однотипные узлы, организованы в двунаправленный список.

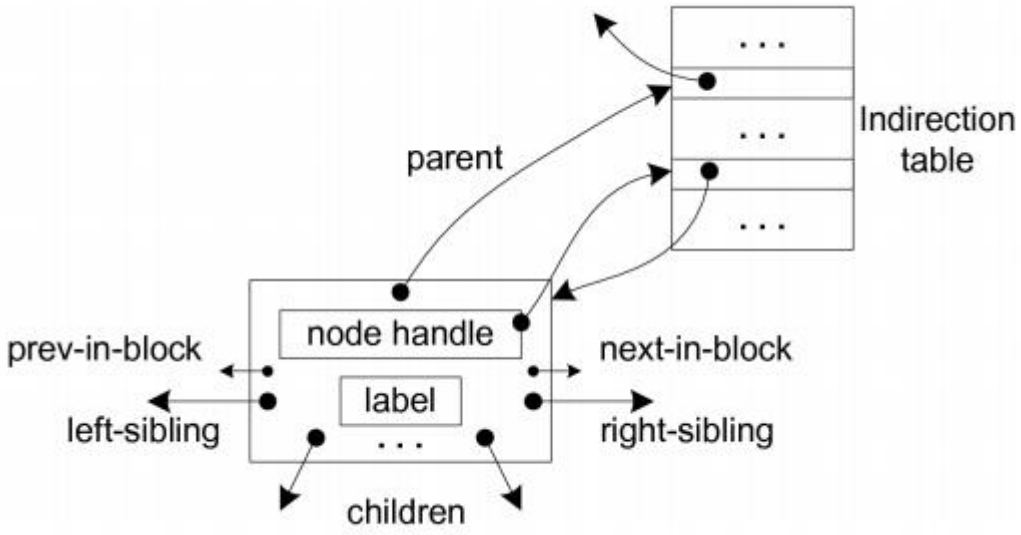

Рисунок 4: Структура заголовка узла.

Рассмотрим структуру заголовка произвольного узла XML документа[10]:

- *Label* – поле, содержащее в себе номер соответствующей блоку описывающей схемы.
- *Node handle* – уникальный идентификатор узла, остающийся неизменным на протяжении всего его жизненного цикла.
- *Left-sibling* и *right-sibling* – ссылки на предыдущий и следующий узел в документе относительно целевого узла.
- *Next-in-block* и *prev-in-block* – ссылки на предыдущий и следующий узел, содержащиеся в этой же странице.

Также произвольный узел содержит ссылку на родительский узел – parent node – а также ссылку на первый дочерний узел child node. Действительно, так как все дочерние узлы организованы посредством ссылок left-sibling и right-sibling в двунаправленный список, хранить ссылки на все дочерние узлы целевого узла не нужно. К тому же такое решение позволяет добиться фиксированного размера заголовка узла.

В дальнейшем под *сериализацией* узла мы будем понимать процесс преобразования узла из внутреннего представления базы данных в последовательность символов (XML формат).

Необходимость в сериализации узлов может возникать при произвольных xQuery запросах к СУБД.

Вернемся к описывающей схеме, изображенной на Рисунке 2. Пусть имеется соответствующий схеме документ, содержащий множество узлов book. Рассмотрим xQuery запрос вида

for $x in doc("library")/library/book
return $x/author

Для каждого узла book необходимо вернуть содержимое его дочерних узлов author. При выполнении такого запроса, согласно схеме, узлы book будут перебираться последовательно, начиная с узла, на который ссылается узел book в описывающей схеме. Для каждого из узлов book будет находиться узел author, посредством использования ссылок child и next-sibling. Затем вновь произойдет обращение к странице, содержащей узлы book, для нахождения следующего узла book по ссылке next-in-block (или next-sibling, если все узлы в странице были перебраны) и т.д.

Таким образом, при сериализации дочерних элементов (author) для узлов определённого типа(book), возникает эффект многократного обращения к страницам, содержащим целевые узлы(book). Период таких обращений может варьироваться в зависимости от расстояния до нужного дочернего элемента[2] и числа необходимых дочерних элементов. Произвольная страница, содержащая N целевых узлов(book), наберет число обращений к себе, равное N и перестанет использоваться, а следовательно, быть нужной в кэше.

Приведенная выше особенность Sedna имеет негативное влияние на эффективность работы алгоритмов замещения, которые в своей работе опираются на частоту обращений к страницам. В качестве примера можно привести GCLOCK – страницы, набравшие большое значение счетчика обращений за счет описанного эффекта, перестают использоваться, однако ещё долго «висят» в кэше «по инерции», занимая место.

[2] Имеется ввиду число узлов, которое необходимо пройти от целевого узла до дочернего, используя ссылки child и next-sibling.

Предлагается алгоритм замещения, учитывающий особенность распределения обращений к блокам во время сериализации. В основу алгоритма легли принципы, применяющиеся в алгоритмах LIRS и LRU.

Пусть с каждым блоком в кэше ассоциированы следующие значения:

- CALLS – число обращений к странице, прошедших с момента её помещения в кэш. Учитываются только первые K обращений, то есть CALLS <= K.
- LAST_TIMESTAMP – последнее «время» обращения к странице. Под «временем обращения» будем понимать значение глобальной переменной CURRENT_TIMESTAMP, аналогичной использующейся алгоритмами LRU и LRD.
- SUM_DELTA – инициализируется значением 0. При каждом обращении к странице таком, что текущее значение CALLS < K, увеличивается на величину (CURRENT_TIMESTAMP – LAST_TIMESTAMP).

Таким образом, значение SUM_DELTA / CALLS для данной страницы даёт возможность оценить величину среднего периода обращений к ней. Данная величина формируется за первые K обращений к странице, в дальнейшем остаётся неизменной. Если не ограничивать параметр CALLS у страницы, с увеличением CALLS величина SUM_DELTA / CALLS для всех страниц в кэше становится примерно одинаковой, что негативно отражается на эффективности работы алгоритма.

При необходимости замещения, выбирается страница, имеющая максимальное значение разности (CURRENT_TIMESTAMP – LAST_TIMESTAMP) – SUM_DELTA / CALLS, то есть наибольшее превышение своего среднего периода обращений.

Для страниц, которые за короткий промежуток времени набрали большое число обращений и перестали использоваться, величина SUM_DELTA / CALLS будет мала, такие страницы будут быстро вытесняться из кэша. Страницы с долгосрочной полезностью, к которым обращения происходят регулярно, но с большим интервалом, будут удерживаться в кэше за счет большого значения SUM_DELTA. Кроме того, алгоритм способен адаптироваться к ситуациям, когда обращения к странице с течением времени меняют характер своего поведения.

В дальнейшем предлагается обозначать описанный алгоритм как DELTA.

4.2 Типизация страниц в кэше СУБД Sedna

Рассмотрим ещё один алгоритм для СУБД Sedna, в своей работе в большей степени опирающийся на разделение страниц в кэше по типам содержащихся в них данных. Число же произведенных к страницам обращений будет иметь меньшее значение при принятии решения о выталкивании той или иной страницы за счет динамического ограничения сверху максимально возможного веса страницы в зависимости от типа блока. Таким образом, алгоритм учитывает специфичное свойство Sedna, из-за которого страницы могли неоправданно быстро набирать вес за короткий промежуток времени.

Введём следующие типы страниц:

- Узловые индексные страницы (INDEX NODE)
- Листовые индексные страницы (INDEX LEAF)
- Страницы данных (DATA)
- Остальные страницы (OTHER)

Каждая страница в кэше обладает следующими параметрами:

- CALLS – текущее число обращений к странице.
- CURRENT_WEIGHT – текущий «вес» страницы. Увеличивается вместе с CALLS на 1 при каждом обращении к странице.
- MAX_WEIGHT – величина, задающая максимально возможное значение для параметра CURRENT_WEIGHT.

Выталкивание страниц из кэша ведется по принципу GLOCK, в качестве веса страницы используется его параметр CURRENT_WEIGHT. Менеджер буферов обладает статистикой по типам страниц в кэше, как показано в Таблице 3 (значения в таблице приведены в качестве примера).

Статистика обновляется «с запаздыванием», при каждом выталкивании система запоминает число обращений, произведенных к удаляемой из кэша странице.

Величина SUM_WEIGHTS / REPLACEMENTS характеризует среднее число обращений к страницам данного типа. Чем больше эта величина, тем большее значение MAX_WEIGHT получают страницы данного типа при помещении в кэш. Смысл данного решения в том, чтобы дать возможность набрать высокий вес новым страницам того типа, к которому обращения производились чаще всего. При этом начальный вес при помещении страницы в кэш – величина постоянная для всех типов страниц и равная 1.

Таблица 3: Используемая менеджером буферов статистика для каждого типа страниц.

Тип страниц	REPLACEMENTS (Число замещенных страниц)	SUM_WEIGHTS (Суммарный вес замещенных страниц)
INDEX NODE	50	3000
INDEX LEAF	10	1000
DATA	150	4500
OTHER	30	900

Если к новой странице с типом, обладающим высоким значением SUM_WEIGHTS / REPLACEMENTS, не будет произведено большое число обращений, она быстро покинет кэш по принципу GCLOCK, несмотря на высокое значение MAX_WEIGHT.

В рамках данной работы будем называть описанный алгоритм PPCLOCK – Page Priority CLOCK.

5 Описание практической части

Рассматриваемые в работе алгоритмы реализованы на языке C, который также являлся основным языком программирования при разработке СУБД Sedna[11].

Мы отказались от сравнения алгоритмов замещения по времени выполнения xQuery запросов. В качестве критерия для сравнения эффективности алгоритмов замещения было выбрано число обращений к внешним устройствам хранения данных. В отличие от времени выполнения, число обращений к внешним устройствам не зависит от конфигурации аппаратного и программного обеспечения ЭВМ, на которой происходит тестирование, а также от способа реализации алгоритма замещения.

Под *сценарием тестирования* здесь и далее мы будем понимать совокупность xQuery запросов к БД, обеспечивающих загрузку целевого xml документа, создание вспомогательных структур данных (индексов) и последовательное выполнение запросов к загруженному документу с возможностью сбора статистики по числу замещённых блоков.

Для тестирования новых алгоритмов замещения были разработаны две программы: для автоматической генерации сценариев тестирования и для исполнения сценариев в Sedna. Для реализации данных программ использовался язык Java.

В качестве целевых XML документов для запросов из сценариев тестирования были выбраны документы, описывающие абстрактный web-форум, со структурой, аналогичной структуре следующего документа:

```
<Forum>
 <Authors>
  <Author id="1" name="author1" param1="value1"/>
  <Author id="2" name="author2" param1="value2"/>
  ...
 </Authors>
 <Topics>
  <Topic id="1" name="topic1">
```

```
    <Discussions>
     <Discussion id=”1” name=”discussion1”>
      <Messages>
       <Message id=”1” name=”message1” author_id=”1”/>
       <Message id=”2” name=”message1” author_id=”2”/>
       <Message id=”3” name=”message1” author_id=”3”/>
      </Messages>
     </Discussion>
    </Discussions>
   </Topic>
  </Topics>
</Topic>
```

Узлы Author, Topic, Discussion, Message проиндексированы по параметру id, узлы Message также проиндексированы по параметру author_id.

Произвольный сценарий тестирования характеризуется общим числом исполняемых запросов к БД, а также типом запросов и их количественным соотношением. В сценариях мы использовали простейшие xQuery запросы по целевому документу. Список использованных запросов можно посмотреть в приложении А.

Сценарии тестирования представляют из себя набор запросов по образцу выше, взятых в различных количественных соотношениях между собой, выполняющихся в определенном порядке (при генерации сценария тестирования задаётся случайный порядок выполнения запросов, далее при каждом выполнении сценария порядок выполнения запросов сохраняется).

Было рассмотрено 2 набора сценариев тестирования. В первом из них запросы обрабатывали большие объемы данных по сравнению с размером всей базы в целом: для обработки одного запроса мог быть задействован весь кэш целиком (например, возврат всего поддерева Topic). Размер тестовой БД вместе с индексами составлял около 1 Гб, размер кэша – 1600 страниц (около 100 Мб). Каждый из сценариев statements включал в себя 500 запросов. Описанный набор сценариев помечен как *huge*.

Второй набор сценариев (*standart*) включал в себя запросы, в большинстве своём исполняющиеся быстро и не требующие много места в кэше под затрагиваемые ими блоки. Размер тестовой БД – 2Гб, размер кэша – 3200 страниц. Сценарии statements содержали 5000 запросов.

Сценарии load_document и create_indexes производят, соответственно, загрузку целевого документа в базу и создание индексов по нему для каждого из наборов сценариев тестирования.

Результаты тестирования приведены в Таблице 4. В таблице 5 содержится сравнительная характеристика реализованных алгоритмов по сравнению с изначально имевшимся алгоритмом FIFO.

Согласно полученным результатам, наименьшего числа замещений по сравнению с FIFO удалось добиться в случае применения алгоритма LRU (1.65%) и PPCLCOK (1.56%). Хуже всего PPCLOCK работает на сценариях тестирования *statements-01_huge* и *statements-04_huge*. Первый из них представляет собой набор запросов вида 10 и 11 (см. Приложение А), взятых в равном количестве. Сценарий *statements-04_huge* состоит из запросов вида 7 – 12, также взятых в равном количестве. Таким образом, указанные сценарии целиком состоят из запросов, выполняющих подсчет значения некоторой аналитической функции (count, max) над определенным множеством узлов (Author, Messages, Discussion). Если исключить эти 2 сценария из рассмотрения, получим результат для PPCLOCK равный 3% (для LRU в случае исключения указанных сценариев – 2%).

Дополнительно, если исключить из рассмотрения сценарии тестирования load_document и create_indexes[3], результат PPCLOCK составит 3.94%, LRU – 1.58%, то есть PPCLOCK превосходит LRU более чем в два раза.

[3] На основании того факта, что загрузка документов в базу и создание индексов производятся гораздо реже, чем выполнение различных запросов на получение данных.

Таблица 4: Результаты тестирования: количество произведенных каждым алгоритмом замещений для каждого из сценариев тестирования. Выделены лучшие результаты для каждого из сценариев тестирования.

	FIFO	CLOCK	LRU	PPCLOCK	DELTA	GCLOCK-5
load_document_huge	13938	13795	13792	13795	13793	13860
create_indexes_huge	9718	9346	8796	9721	8794	9657
statements-01_huge	196707	198491	197587	205900	199030	199158
statements-02_huge	186351	186586	186315	182920	186292	186450
statements-03_huge	72104	71946	71588	70931	71532	71978
statements-04_huge	989232	991116	991321	1165217	983543	991927
statements-05_huge	612333	614178	612260	622411	612414	615681
statements-06_huge	684276	685741	684023	588175	681507	690454
statements-07_huge	101163	100943	100539	97884	100512	101023
statements-08_huge	2188	2195	2216	2199	2191	2201
load document_standart	24444	24403	24395	24403	24397	24430
create indexes_standart	14763	14509	14237	14845	14253	14516
statements-01_standart	45325	44020	43409	41982	45995	44258
statements-02_standart	197551	195370	194295	191250	194502	196768
statements-03_standart	79275	77704	76897	74066	79943	79076
statements-04_standart	23453	23273	23040	22686	23951	23644
statements-05_standart	52796	51419	50841	48911	53559	51875
statements-06_standart	712848	712102	710688	708452	804394	713564
statements-07_standart	43694	42992	42814	42677	43025	43159
statements-08_standart	77192	75648	75062	74319	79454	76697

Алгоритм DELTA показал неплохие результаты на первом (*huge*) наборе сценариев тестирования, примерно равные результатам LRU (1.17% у DELTA, 1% у LRU), однако на втором наборе оказался неэффективен. В частности на сценарии *statements-06_standart* было произведено на 12% больше замещений, чем у алгоритма FIFO. Сценарий *statements-06_standart* состоит из запросов 1, 3, 4, 5, 6 (по 5% соответственно от общего числа запросов) и 7, 8, 9, 10 (примерно по 18%). Ограничение на число CALLS для DELTA было выбрано равным 10^4.

[4] Как наиболее эффективное из рассмотренных значений MAX CALLS.

Также для алгоритма GLOCK получены ожидаемо низкие результаты. Тестирование проходила модификация GLOCK с ограничением максимального веса страниц равным 5[5].

Таблица 5: Результаты тестирования: насколько меньше (в процентах) было произведено замещений алгоритмами относительно алгоритма FIFO.

	CLOCK	LRU	PPCLOCK	DELTA	GCLOCK-5
load_document_huge	1,026	1,047	1,026	1,04	0,56
create_indexes_huge	3,828	9,488	-0,031	9,508	0,628
statements-01_huge	-0,907	-0,447	-4,673	-1,181	-1,246
statements-02_huge	-0,126	0,019	1,841	0,032	-0,053
statements-03_huge	0,219	0,716	1,627	0,793	0,175
statements-04_huge	-0,19	-0,211	-17,79	0,575	-0,272
statements-05_huge	-0,301	0,012	-1,646	-0,013	-0,547
statements-06_huge	-0,214	0,037	14,044	0,405	-0,903
statements-07_huge	0,217	0,617	3,241	0,644	0,138
statements-08_huge	-0,32	-1,28	-0,503	-0,137	-0,594
load document_standart	0,168	0,2	0,168	0,192	0,057
create indexes_standart	1,721	3,563	-0,555	3,455	1,673
statements-01_standart	2,879	4,227	7,376	-1,478	2,354
statements-02_standart	1,104	1,648	3,19	1,543	0,396
statements-03_standart	1,982	3	6,571	-0,843	0,251
statements-04_standart	0,767	1,761	3,27	-2,123	-0,814
statements-05_standart	2,608	3,703	7,359	-1,445	1,744
statements-06_standart	0,105	0,303	0,617	-12,842	-0,1
statements-07_standart	1,607	2,014	2,328	1,531	1,224
statements-08_standart	2	2,759	3,722	-2,93	0,641
Average	0,90865	1,6588	1,5591	-0,1637	0,2656

[5] По аналогии с использующейся модификацией GCLOCK в Postgresql.

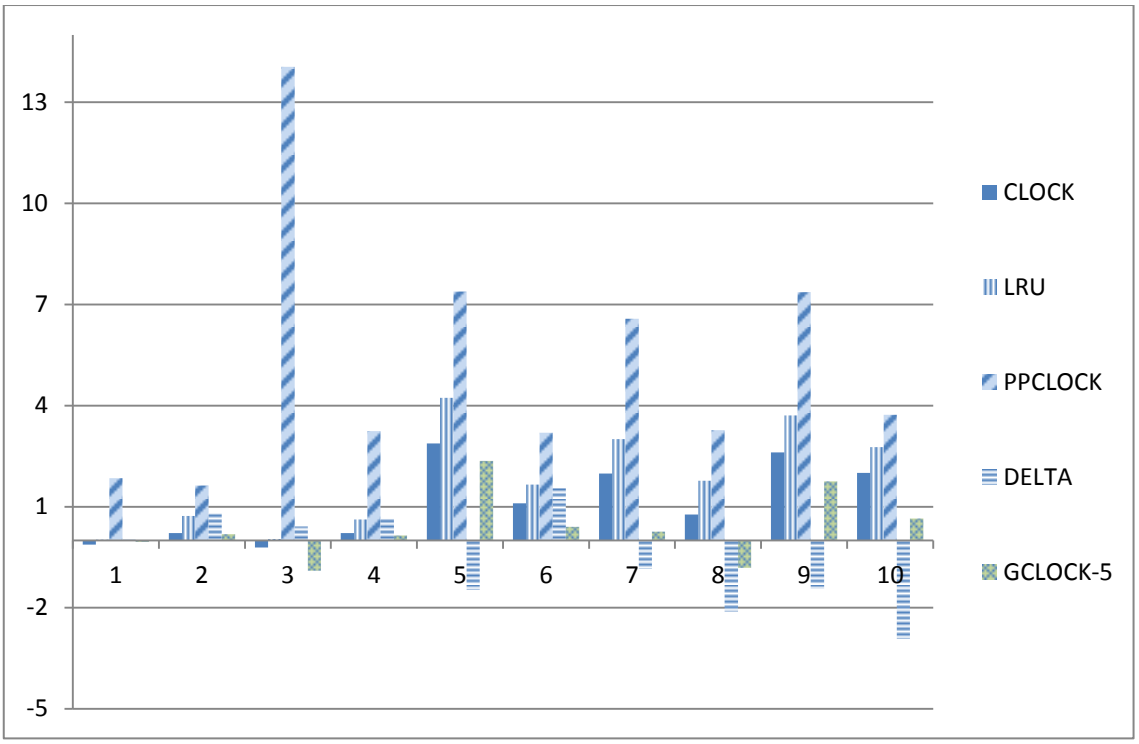

Рисунок 5: Результаты тестирования. Сравнение эффективности алгоритмов замещения на сценариях statements-02_huge, statements-03_huge, statements-06_huge, statements-07_huge, statements-01_standart, statements-02_standart, statements-03_standart, statements-04_standart, statements-05_standart, statements-08_standart (ось абсцисс, последовательность на графике сохранена). Ось ординат: насколько меньше замещений (в %) было произведено.

6 Заключение

В ходе данной работы были проанализированы и реализованы в XML СУБД Sedna алгоритмы, применяющиеся в существующих реляционных СУБД. Также были разработаны и реализованы новые алгоритмы замещения, принцип работы которых учитывает особенности Sedna. Было произведено тестирование алгоритмов замещения на различных конфигурациях СУБД. Для облегчения процесса тестирования разработан соответствующий инструментарий, позволяющий генерировать и выполнять сценарии тестирования с заданными параметрами.

По результатам тестирования, на большинстве произведенных тестов алгоритм PPCLOCK не уступает по свой эффективности другим алгоритмам, и на многих тестах PPCLOCK показывает лучший результат. В ходе тестирования были выявлены сценарии, на которых эффективность работы PPCLOCK существенно падала (*statements-01_huge* и *statements-04_huge*), что говорит о необходимости в дополнительных исследованиях данного алгоритма. Возможным источником падения эффективности на приведенных сценариях является слишком быстрое увеличение веса определённых типов блоков по сравнению с другими. В качестве решения данной проблемы может быть предложено ограничение сверху максимально возможного значения веса для каждого типа блоков, однако реализация этого решения выходит за рамки данной работы.

В качестве дополнительного преимущества алгоритма PPCLOCK можно отметить простоту его реализации. В ходе тестирования среднее количество страниц, проходимое PPCLOCK до искомой замещаемой страницы, составило 6 (для CLOCK и GCLOCK соответственно 2 и 6), что много меньше числа страниц, которое требовалось бы пройти в худшем случае (1600 и 3200 для 2 наборов сценариев). Следовательно, обход замкнутого списка страниц в PPCLOCK может быть эффективно реализован простым последовательным просмотром страниц в кэше.

Как показали многочисленные произведенные тесты, не вошедшие в данную работу, эффективность алгоритма PPCLOCK возрастает при увеличении размера кэша. Формальное рассмотрение вопроса об

актуальности аномалии Белади для алгоритма PPCLOCK не входит в данную работу, так как представляется достаточно трудоёмкой задачей. К тому же, лежащий в основе PPCLOCK алгоритм GCLOCK успешно используется в реальных СУБД, что говорит о его надежности.

Разработанный алгоритм DELTA на произведенных тестах показал свою неэффективность, что может быть обусловлено несоответствием сделанных в алгоритме предположений реальным закономерностям в работе Sedna.

Список литературы

[1] T. Haerder, W. Effelsberg. Principles of Database Buffer Management. // ACM Transactions on Database Systems. 1984. 9. N 4. P. 560–595.

[2] G. M. Sacco, M. Schkolnick. Buffer Management in Relational Database systems // ACM Transactions on Database Systems. 1986. 11. 4. P. 473–498.

[3] Kirby McMaster, Samuel Sambasivam , Nicole Anderson. How Anomalous Is Belady's Anomaly? // Issues in Informing Science and Information Technology. 2009. 6. P. 825-836.

[4] Oracle. InnoDB Glossary [HTML] (http://dev.mysql.com/doc/innodb/1.1/en/glossary.html#glos_lru)

[5] S. Bansal, D. S. Modha. CAR: Clock with Adaptive Replacement // Proceedings of the 3rd USENIX Conference on File and Storage Technologies. Berkeley, CA, USA: USENIX Association, 2004. P. 187–200.

[6] G. Smith. Inside the PostgreSQL Shared Buffer Cache [PDF] (http://www.westnet.com/~gsmith/content/postgresql/InsideBufferCache.pdf)

[7] Y. Zhou, J. F. Philbin. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches // Proceedings of the 2001 USENIX Annual Technical Conference. Berkeley, CA, USA: USENIX Association, 2001. P. 91-104.

[8] S. Jiang, X. Zhang. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. // Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems. New York, NY, USA: ACM, 2002. P. 31–42.

[9] I. Taranov, I. Shcheklein, A. Kalinin. Sedna: native xml database management system (internals overview) // Proceedings of the 2010 international conference on Management of data. New York, NY, USA: ACM, 2010. P. 1037–1046.

[10] A. Fomichev, M. Grinev, S. Kuznetsov. Sedna: a native XML DBMS // Proceedings of the 32nd conference on Current Trends in Theory and Practice of Computer Science. Berlin, Heidelberg: Springer-Verlag, 2006. P. 272-281.

[11] Institute for System Programming RAS. Sedna documentation [HTML] (http://sedna.org/documentation.html)

Приложение А. Список использовавшихся в сценариях тестирования запросов.

XQuery запросы приведены с точностью до использованных в них значений $VALUE1 и $VALUE2.

1. for $x in (index-scan-between(“forum_messageIdIndex”, $VALUE1, $VALUE2, “SEG”))
   return <result>{$x/@name}</result>

2. for $x in (index-scan-between(“forum_messageIdIndex”, $VALUE1, $VALUE2, “SEG"))
   order by $x/@id
   return <result>{$x/@name}</result>

3. for $x in (index-scan-between(“forum_authorIdIndex”, $VALUE1, $VALUE2, ”SEG”))
   return <result>{$x/@name}</result>

4. for $x in (index-scan-between(“forum_messageAuthorIdIndex”, $VALUE1, $VALUE2, “SEG”))
   return <result>{$x/@name}</result>

5. for $x in (index-scan(“forum_discussionIdIndex”, $VALUE1,”EQ”))
   return <result>{$x/Messages}</result>

6. for $x in (index-scan(“forum_topicIdIndex”, $VALUE1, ”EQ”))/Discussions/Discussion
   return <result>{$x/@name}</result>

7. for $x in (index-scan-between(“forum_messageAuthorIdIndex”, “SEG”))
   return <MessagesCount> {count($x) } </MessagesCount>

8. let $po2 := index-scan(“forum_discussionIdIndex”, $VALUE1, ”EQ”)
   let $mes := $po2/Messages/Message
   return <MessagesCount> {count($mes) } </MessagesCount>

9. let $po2 := index-scan(“forum_topicIdIndex”, $VALUE1, “EQ”)
   let $disc := $po2/Discussions/Discussion
   return <DiscCount> {count($disc) } </DiscCount>

10. let $aut := document(“forum”)/Authors/Author
    return <DiscCount> {max($aut/@points) } </DiscCount>

11. let $mes :=
    document(“forum”)/Topics/Topic/Discussions/Discussion/Messages/Message
    return <MessagesCount>{count($mes)}</MessagesCount>

12. let $po2 := index-scan(“forum_topicIdIndex”, $VALUE1, “EQ”)
    let $mes := $po2/Discussions/Discussion/Messages/Message
    return <MessagesCount> {count($mes) } </MessagesCount>