Алгоритмы управления буферным пулом СУБД при работе с флэш-накопителями

Кузнецов С.Д., Прохоров А.А.
<kuzloc@ispras.ru>, <EmailToProkhorov@gmail.com>

Аннотация. Одним из важнейших механизмов повышения скорости работы современных СУБД является кэширование часто читаемых или записываемых данных в оперативной памяти. Классические алгоритмы замещения страниц БД в кэше стремятся минимизировать промахи буферного пула СУБД. Данный метод оптимизации негласно опирается на тот факт, что скорость записи и чтения данных одинакова. Постепенное совершенствование и удешевление технологии производства флэш-памяти привели к созданию твердотельных накопителей данных(SSD), которые в настоящее время все чаще используются как в персональных компьютерах, так и в системах хранения данных. Флэш-накопители имеют серьезные преимущества по сравнению с традиционными жесткими дисками, главные из которых – более высокая скорость чтения и записи, а также значительно меньшее время доступа к данным. Однако, самые распространенные виды флэш-памяти читают данные с большей скоростью, чем записывают. Из-за данной особенности использование классических алгоритмов замещения страниц при кэшировании дисковых данных неэффективно. В данной статье производится обзор современных алгоритмов управления буферным пулом СУБД, которые предназначены для работы с накопителями на флэш-памяти.

Ключевые слова. Буферный пул СУБД, кэширование дисковых данных, флэш-память, твердотельные накопители данных.

1. Введение

С появлением накопителей информации на флэш-памяти время доступа к данным снизилось с 10 мс., характерных для жестких дисков, до 0,1 мс. Однако, скорость работы современных СУБД по-прежнему лимитируется скоростью работы устройств хранения данных. В связи с этим для повышения скорости работы системы в целом выгодно кэшировать определенный набор данных БД в оперативной памяти.

При разработке алгоритмов управления буферным пулом решается задача минимизации математического ожидания среднего времени доступа к страницам данных. Для носителей информации на магнитных дисках скорость

чтения и записи данных одинакова, поэтому выгодно хранить в буферном пуле наиболее часто используемые страницы(алгоритм Белади) [1]. Такой алгоритм практически не реализуем, поэтому используются его эффективные приближения LRU, LFU, CLOCK и др. [2]

Разработчики флэш-накопителей были вынуждены не только добиваться превосходства над магнитными дисками по техническим характеристикам, но и стремиться к минимизации цены конечного устройства. Именно увеличение плотности хранения данных и экономия на вспомогательных управляющих элементах привели к ряду особенностей, характерных для наиболее распространенных видов флэш-памяти. Чтение и запись в этих устройствах производится блоками в несколько килобайт, удаление блоками размером в сотни килобайт, а запись может осуществляться только после удаления. Данные ограничения привели к асимметрии операций чтения и записи данных.

Для построения буферного пула, оптимизированного для работы с флэш-памятью, необходимо не только поддерживать высокий процент попаданий в дисковый кэш, но и минимизировать процент дорогостоящих промахов по записи. Для решения этих задач в последние годы было разработано несколько алгоритмов, обзор которых изложен в данной статье.

2. Накопители данных на флэш-памяти

Устройства хранения данных на основе флэш-памяти появились в конце 80-ых годов прошлого столетия [3]. Из-за дороговизны и несовершенства технологии производства твердотельные накопители долгое время не могли конкурировать с устройствами на магнитных дисках. Стоимость флэш-памяти в пересчете на 1 Гбайт по-прежнему на порядок выше, чем у дисковых устройств, однако, благодаря высокой скорости обмена и доступа к информации твердотельные накопители получают все большее распространение в современных системах хранения и обработки данных. Сам твердотельный накопитель представляет собой набор модулей флэш-памяти под управлением контроллера устройства.

Устройство флэш-памяти

Флэш-память – разновидность полупроводниковой технологии энергонезависимой электрически перепрограммируемой памяти. Ячейкой хранения данных является транзистор с плавающим затвором [4]. Подобный транзистор может хранить в себе заряд разной величины, поэтому в одной ячейке можно уместить несколько бит информации [5]. Из-за постепенной деградации полупроводника ячейка флэш-памяти обладает ограниченным ресурсом. Для современных накопителей количество циклов перезаписи составляет в среднем 5000.

По структуре компоновки элементов памяти в чипе различают NOR(Not OR) и NAND(Not AND) память. Конструкция NOR позволяет управлять состоянием каждой из ячеек в отдельности, в то время как NAND позволяет управлять только блоками данных. Типичный размер блоков чтения и записи 4 Кбайт, а удаления – 128 Кбайт [6]. Так как NAND память значительно дешевле, именно она используется в современных флэш-носителях.

Особенности работы твердотельных накопителей данных

Так как ячейки флэш-памяти подвержены износу, контроллеры твердотельных накопителей выполняют ряд вспомогательных операций, которые непосредственно не связаны с вводом-выводом, а служат лишь для продления времени работы устройств.

Основной механизм, который используется для этих целей, перемешивание содержимого диска [7]. Если в магнитных дисках логические блоки данных имеют постоянное во времени отображение на физические блоки, то для SSD подобное отображение может изменяться во времени. Так логические блоки часто записываемых файлов с каждой новой записью на накопитель записываются на новые физические блоки. В случае если новый блок также содержит полезные данные, они в свою очередь переписываются на другую позицию согласно алгоритмам контроллера. Таким образом, когда-либо записанный в последовательные физические блоки большой файл, к которому обращения происходят редко, со временем может “расползтись” на разрозненные блоки.

Отдельно стоит отметить, что перед записью данных в ячейки памяти, данные из них нужно удалить, что вызывает трудности, связанные с различными размерами блоков записи и удаления для модулей NAND памяти.

Рассмотрим последовательность действий, которые необходимо выполнить контроллеру при обращении на запись данных:

1. считывание блока, содержащего модифицируемые данные во внутренний буфер;
2. модифицирование необходимых байтов;
3. считывание полезных данных, оставшихся в данном блоке удаления;
4. стирание блока в микросхеме флэш-памяти;
5. вычисление нового местоположения блока согласно механизму перемешивания данных;
6. считывание и удаление данных из нового блока, если новый блок не пуст;
7. запись измененного блока, а также всей затронутой полезной информации на новые позиции.

Выполнение всех этих операций приводит к асимметрии чтения и записи данных.

Для того чтобы контроллеру не приходилось выполнять шаг 6, необходимо иметь информацию о том, какие блоки данных не содержат полезной информации. Для этого достаточно запоминать удаленные и помнить ни разу не задействованные области устройства. Сложность данной задачи связана с тем, что при работе с дисковыми устройствами в операции реального удаления данных не было никакой необходимости, и такая команда отсутствовала в интерфейсе обмена данными ОС с накопителями информации. В связи с её острой необходимостью для флэш-носителей, эту команду включили в современные интерфейсы обмена данными и она получила название TRIM.

Так как блоки удаления больше, чем блоки записи, контроллер зачастую вынужден записывать не только один изменившийся блок данных, но и затронутые удалением данные из смежных блоков. Данный эффект усиления записи приводит к ускоренному износу устройства.

Технические характеристики SSD

Рассмотрим сводную таблицу по техническим характеристикам, которая демонстрирует преимущества устройств на флэш-памяти, по сравнению с устройствами на магнитных дисках.

Характеристика	HDD	SSD(NAND) [8]
Время доступа на чтение, мс	20	0,1
Время доступа на запись, мс	20	0,2
Скорость последовательного чтения, Мбайт/сек	140	480
Скорость последовательной записи, Мбайт/сек	135	240
Произвольное чтение блока в 8 Кб, DQ 32, IOPS	150	9000
Произвольная запись блока в 8 Кб, DQ 32, IOPS	150	4000

Глубина очереди ввода/вывода важна и для твердотельных накопителей, так как обеспечивает большую гибкость при записи данных, а также позволяет осуществлять параллельный и асинхронный доступ к отдельным блокам памяти.

Из-за применения более сложных алгоритмов и использованию буферизации современные твердотельные накопители демонстрируют хорошую скорость записи данных, которая лишь вдвое уступает скорости чтения. Из-за применяемых оптимизаций скорость записи может зависеть как от текущей наполненности накопителя, так и от истории предыдущих обращений к устройству.

3. Буферный пул СУБД

Внешние устройства хранения данных работают на несколько порядков медленнее, чем оперативная память. Для снижения времени, которое тратится на обмен данными с внешними накопителями, СУБД поддерживает определенный набор часто используемых данных в буферном пуле. Буферный пул представляет собой область оперативной памяти, которая находится под полным контролем СУБД, т.е. она не может быть выгружена во внешнюю память операционной системой.

Страничная организация хранения данных в БД

Наибольшее распространение получила страничная организация памяти. Место во внешней памяти, предоставляемое для размещения информации в базе данных, логически разделяется на страницы фиксированного размера [9]. Каждая страница данных обычно содержит множество записей лежащих в одной таблице либо узлы существующих индексов. В большинстве систем обработку данных на уровне страниц осуществляет операционная система, а обработку записей внутри страницы ведет СУБД. Такая структура хранения данных обеспечивает эффективное хранение и доступ к записям с учетом всех ограничений, которые накладывают механизмы СУБД.

Большинство современных баз данных используют страницы размером от 2 до 16 килобайт. К примеру, в PostgreSQL страницы имеют размер 8 килобайт. На выбор размера страницы влияют множество факторов. Для увеличения эффективности работы с внешними устройствами требуется использовать страницы как можно большего размера. С другой стороны, для предотвращения косвенных блокировок записей, которые могут возникнуть при блокировке страницы на чтение или запись, необходимо стремится к уменьшению размера страницы.

Страничная организация памяти позволяет рассматривать задачу кэширования дисковых данных в виде кэширования определенного набора страниц БД.

Доступ к страницам данных через буфер

Доступ к страницам БД через буферный пул осуществляется в 2 этапа. Вначале идет проверка на наличие страницы в буферном пуле, а только после этого идет реальное обращение к странице на внешнем носителе. Страница, к которой выполняется обращение СУБД, должна быть обязательно помещена в оперативную память, так как работа со страницами во внешней памяти не допускается. В буферном пуле также могут храниться страницы, содержимое которых отличается от первоначального, такие страницы называются грязными. Они возникают в том случае, если страница сразу запрашивалась на запись, а также в том случае, если первоначально запрошенная на чтение страница подверглась изменениям в последствии.

Если размер буферного пула превысил выделенное под него место, для вставки новой страницы необходимо выбрать страницу-жертву, которая будет

удалена из пула. Если выбранная страница помечена грязной, перед удалением из буфера её необходимо записать на внешний носитель.

Базовые принципы замещения страниц

Для уменьшения времени работы с флэш-носителем можно выделить следующие принципы построения буферного пула СУБД [10].

1. Минимизация операций записи.
2. Использование страниц как можно большего размера для увеличения скорости чтения/записи с устройства.
3. Сокращение общего количества обращений к накопителю, за счет высокого процента попаданий в буфер.
4. Преобразование последовательности обращений на запись к флэш-устройству для уменьшения коэффициента усиления записи и ускорения выполнения операций.

Данные принципы невозможно рассматривать изолированно друг от друга, а также не принимая во внимание ограничения со стороны базовой функциональности СУБД.

Для уменьшения количества записей во внешнюю память можно вытеснять из буфера в первую очередь чистые страницы, но при этом будет возрастать процент промахов буфера.

Использование страниц большего размера уменьшает эффективность кэширования данных в целом. При обращении к записям, кэшируются не только они, но и все те записи, что находятся в той же странице. Так как побочные записи могут быть не нужны, оперативная память на них используется напрасно.

Страницы больших размеров также могут негативно сказаться на скорости обработки транзакций. Логическая синхронизация доступа к данным, которая используется при сериализации транзакций, не обеспечивает физической синхронизации доступа к страницам буферного пула. Одновременное изменение страницы данных различными транзакциями может привести к потере информации. В связи с этим, для обеспечения корректного доступа к страницам данных используется механизм блокировок. Перед каждым обращением к записи, содержащейся в странице, сама страница блокируется на чтение или запись. При блокировке страницы на запись её содержимое не может быть ни прочитано, ни изменено другими транзакциями. Увеличение размеров страниц приведет к увеличению количества взаимных блокировок транзакций.

Для ускорения выполнения обращений к носителю на запись, необходимо массово выталкивать страницы [11], расположенные в одном записываемом или удаляемом блоке данных флэш-памяти. Так как точно определить физическое расположение данных невозможно, используется предположение о том, что логически подряд расположенные страницы, имеют близкие

физические адреса. Техника совместного выталкивания близко расположенных страниц получила название кластерной записи, а непересекающиеся наборы таких страниц – кластерами. Стоит отметить, что использование данной техники напрямую способствует увеличению физической локальности логически близких данных, ввиду особенностей работы флэш-памяти.

Современные механизмы сериализации транзакций позволяют выталкивать на внешний носитель грязные страницы ещё не завершенных транзакций, что делает процесс кэширования грязных страниц независимым от основных механизмов СУБД.

Для увеличения процента попаданий в буферный пул возможно использование техники предварительного чтения страниц данных. Использование данной методики также может привести к ускорению работы с флэш-носителями. Однако, из-за того что скорость произвольного доступа к данным у флэш-устройств на несколько порядков выше, чем у дисковых накопителей, а точность угадывания страниц, которые могут пригодиться впоследствии, не высока, от использования данной техники можно отказаться.

4. Алгоритмы управления буферным пулом, учитывающие особенности флэш-памяти

Рассмотрим три современных алгоритма управления буферным пулом СУБД, оптимизированных для работы в системах с флэш-памятью [12].

Алгоритм CFDC(Clean First – Dirty Clustered)

Алгоритм CFDC [13] разбивает буфер на 2 области: W – рабочая область, содержащая горячие страницы, которые запрашиваются либо часто, либо в последнее время, I – приоритетная область, из которой выбираются страницы для выталкивания на внешне устройства.

Параметр λ регулирует долю страниц буферного пула, которые используются приоритетной областью P. Таким образом, если в буфере b страниц, то P содержит λ*b страниц, а оставшиеся (1- λ)*b находятся под управлением W. Отметим, что на область W не накладывается особых требований по управлению страницами, в ней могут находится как чистые, так и грязные страницы, а главное её назначение – предотвращать попадание наболее часто используемых страниц в приоритетную область.

Перемещение страниц из одной области буфера в другую:

1. При попадании в область W, необходимо отметить факт обращения к странице.
2. При попадании в область P, страница переносится в область горячих страниц.
3. В случае добавления новой страницы в область W, из неё в область P выталкивается страница-жертва(min(W)).
4. Если произошел промах буфера, то происходит выталкивание одной из страниц на внешний носитель. При этом страница-жертва всегда выбирается из области P, а свежая страница помещается в область W.
5. Выталкивание страницы-жертвы из буфера.

Более формально процесс обращения к страницам можно представить следующим алгоритмом.

Algorithm 1 CFDC

```
function REQUEST(Page p)
    * Переменные окружения:
    * Буфер Buffer, рабочая область W, приоритетная область P,
    * |W| = (1 − λ) · b ∧ |P| = λ · b.

    if p ∈ Buffer then
        if p ∈ W then
            Отметить попадание p в W;
        else
            Вытолкнуть min(W) в P, переместить p в W;
        end if
    else
```

```
        Страница q ← SelectVictimPr();
        if q is Null then
            q ← выбрать страницу-жертву из W;
        end if
        if q is Dirty then
            Записать q на внешний носитель;
        end if
        очистить q, переписать содержимое p из внещней памяти в q;
        p ← q;
        if p ∈ P then
            Вытолкнуть min(W) в P, переместить p в W;
        end if
    end if
    Return p;
end function
```

Рассмотрим механизм определения страницы жертвы, который используется в данном алгоритме. Приоритетная область P содержит 3 структуры: LRU список L чистых страниц, очередь с приоритетом Q, содержащая кластеры с грязными страницами, хэш-таблица для доступа к страницам кластеров.

Для обеспечения оптимальной работы запросов к внешнему устройству на запись необходимо придерживаться принципа локальности данных при выталкивании страниц. В качестве первого признака, указывающего на локальность данных, принимается близость логических адресов. Весь диапазон номеров страниц, разбивается на непересекающиеся интервалы равной длины, каждому из отрезков соответствует определенный кластер. При этом кластер может содержать только страницы из соответствующего диапазона. Для кластеров размером 4 страницы, получаем следующее разбиение.

Кластер	1				2				…	K+1				…
Стра-ницы	1	2	3	4	5	6	7	8	…	4*k	4*k+1	4*k+2	4*k+3	…

Вторым признаком локальности данных считается порядок попадания страниц в кластер. К примеру, если во второй кластер страницы попали в порядке {5,6,7,8}, он будет приоритетнее для выталкивания, чем первый кластер с порядком {3,1,4,2}.

Так как кластеры могут содержать произвольное число страниц(от 0 до 4), то с точки зрения эффективности записи на внешние носители, большие кластеры выталкивать эффективней. Также необходимо избегать долгосрочного хранения в буфере грязных страниц.

На основании этих факторов для кластера c, содержащего n страниц, используется следующая оценочная функция

$$P(c) = \frac{\sum_{i=1}^{n-1} |p_i - p_{i-1}|}{n^2 \times (globaltime - timestamp(c))},$$

где $p_0, \dots, p_{n-1}$ – порядок попадания страниц в кластер, globaltime – счетчик времени, отсчитывающий события выставки грязных страниц в буфер, timestamp(c) – значение globaltime в момент создания кластера. Сумма в числителе обозначается как IPD – inter-page distance.

Выталкивание страниц осуществляется по следующему алгоритму.

Algorithm 2 CFDC select victim

```
function SelectVictimPr
    * Переменные окружения:
    * Приоритетная область P, содержащая LRU список чистых
    * страниц L, а также очередь с кластерами грязных страниц
    * с их приоритетами Q.

    if L ≠ ∅ then
        v ← LRU страница из L;
    end if
    if v = null then
        cluster c ← кластер с наименьшим приоритетом из Q;
        if c ≠ null then
            v ← LRU страница из c;
            if v ≠ null then
                c.ipd ← 0;
            end if
        end if
    end if
    Return p;
end function
```

Так как операции с чистыми страницами выполняются значительно быстрей, страница-жертва в первую очередь выбирается их списка L. Если L – пуст, то выбирается кластер с наименьшим приоритетом. Следует отметить, что не весь выбранный кластер выталкивается на диск, а лишь одна LRU страница из него. Так как при этом у кластера сбрасывается IPD, все последующие выталкиваемые страницы будут выбраны из этого же кластера.

Разбиение буфера на 2 части также позволяет сэкономить процессорное время, которое тратится на выбор страниц жертвы, так как в этих расчетах не участвуют страницы из рабочей области.

Как видно, данный алгоритм стремится в первую очередь выталкивать из буфера чистые страницы, сохраняя только те из них, которые содержатся в области горячих страниц W. Данный подход эффективен, если запись

осуществляется значительно медленнее чтения, однако, оценить потери от снижения общего процента попаданий в буфер при таком подходе невозможно. Также недостатком данного алгоритма является негибкая политика по разделению буфера на 2 части.

Исследования эффективности данного алгоритма, проведенные его авторами, показывают 25% уменьшение времени исполнения TPC-C тестов, относительно алгоритма LRU, при снижении количества обращений на запись к носителю на 20%. Наилучшие показатели данный алгоритм демонстрирует при значении параметра λ = 75%. Также эффективность данного алгоритма возрастает при использовании флэш-устройств, у которых более высокая разница скоростей чтения и записи.

Алгоритм CASA(Cost-Aware Self-Adaptive)

В отличие от алгоритма CFDC, CASA [13] предполагает динамический расчёт времени выполнения операция чтения и записи. Данный алгоритм разбивает буфер на 2 LRU списка. В одном из них хранятся чистые страницы, в другом грязные. Общая схема работы:

Параметр $\tau \in \mathbb{R}$ определяет количество страниц, которое отводится для хранения чистых страниц. Пусть c_R и c_W нормализованные стоимости операция чтения и записи, то есть $(c_R + c_w = 1) \wedge (c_R \div c_w = \frac{read\ time}{write\ time})$

Для определения τ используется следующий эвристический подход. При каждом попадании в буфер алгоритм пытается увеличить размер той области, в которой была найдена искомая страница. Параметр τ при этом увеличивается пропорционально нормализованной стоимости запроса, а также относительному размеру расширяемой области.

Существует только 2 случая, при которых проводится настройка параметра τ.

1. Попадание запроса на чтение в область C (CASA. строка 11).
2. Попадание запроса на запись в область D (CASA. строка 15).

В первом случае τ увеличивается на величину $c_R \times (|D| \div |C|)$, во втором – уменьшается на $c_W \times (|C| \div |D|)$. Ниже представлен алгоритм доступа к страницам буфера.

Algorithm 1 CASA

```
1:  function REQUEST(Page p, Operation op)
2:      * Переменные окружения:
3:      * Буфер Buffer, емкостью b страниц; список свободных страниц E;
4:      * два LRU списка C - для чистых, D - для грязных страниц;
5:      * τ - размер списка C, где 0 ≤ τ ≤ b; нормализованые стоимости
6:      * чтения и записи c_R и c_W: c_R + c_W = 1 ∧ c_R ÷ c_W = read time / write time.
7:      * Изначально |C| = 0; |D| = 0; |E| = b; τ = 0.
8:
9:      if p ∈ Buffer then
10:         if p ∈ C ∧ op = R then
11:             τ ← min(τ + c_R × (|D| ÷ |C|), b);
12:             Переместить p на MRU позицию C;
13:         else if p ∈ C ∧ op = W then
14:             Переместить p на MRU позицию D;
15:         else if p ∈ D ∧ op = W then
16:             τ ← max(τ − c_W × (|C| ÷ |D|), 0);
17:             Переместить p на MRU позицию D;
18:         else
19:             * p ∈ D ∧ op = R
20:             Переместить p на MRU позицию D;
21:         end if
22:     else
23:         страница-жертва v ← null;
24:         if |E| > 0 then
25:             v ← any from E;
26:         else if |C| > τ then
27:             v ← LRU страница C;
28:         else
29:             v ← LRU страница D;
30:             Записать страницу v на внешний носитель;
31:         end if
32:         Переписать содержимое p из внешней памяти в v;
33:         p ← v;
34:         if op = R then
35:             Переместить p на MRU позицию C;
36:         else
37:             Переместить p на MRU позицию D;
38:         end if
39:     end if
40:     Return p;
41: end function
```

Данный алгоритм был усовершенствован внедрением кластерной записи грязных страниц. Полученный алгоритм получил название SAWC(Self Adaptive with Write Clustering). Сброс всего кластера, который содержит страницу-жертву, на внешнее устройство может привести к выталкиванию часто используемых страниц. Для избежания подобных ситуаций используется информация о частоте доступа к страницам данных. Для записи на накопитель из кластера выбираются только те страницы, которые использовались не чаще страницы-жертвы.

Кластерная запись описывается следующим алгоритмом.

Algorithm 2 CASA clustered

```
1: procedure CLUSTEREDWRITE(Page p)
2:     * Переменные окружения:
3:     * Хэш-таблица с кластерами грязных страниц T.
4:
5:     Записать страницу p на внешний носитель;
6:     кластер c ← кластер из T, содержавший p;
7:     if c exists then
8:         for all q ∈ c ∧ q ≠ p do
9:             if Freq(q) ≠ Freq(p) then
10:                Записать страницу q на внешний носитель;
11:                Удалить q из c;
12:            end if
13:        end for
14:        if |c| = 0 then
15:            Удалить c из T;
16:        end if
17:    end if
18: end procedure
```

Динамический расчет областей чистых и грязных страниц позволяет адаптироваться к текущей нагрузке на БД, увеличивая процент попаданий в буфер. Асимметрия чтения и записи учитывается при определении размеров областей. Так как алгоритм опирается на эвристический механизм разделения буфера, трудно теоретически оценить его эффективность. Также недостатком данного алгоритма является предположение о постоянной величине времени чтения и записи данных на флэш-носитель, которые, как известно, могут меняться в зависимости от текущей нагрузки на носитель и его заполненности.

Согласно исследованиям, которые проведены авторами данного алгоритма, при запуске TPC-C и TPC-H тестов произвольности он демонстрирует

снижение времени выполнения на 20%, относительно алгоритма LRU, и лишь немного уступает CFDC.

Алгоритм FD-Buffer

Данный алгоритм рассматривает задачу оптимизации работы буферного пула в терминах среднего времени доступа к странице. Пусть P_{total} вероятность промаха буфера, E_{dirty} доля грязных страниц среди всех выталкиваемых, а C_{read} и C_{write} время выполнения операций чтения и записи страниц. Тогда математическое ожидание среднего время доступа к странице можно выразить в следующем виде.

$$Cost_{io} = P_{total} \cdot C_{read} + P_{total} \cdot E_{dirty} \cdot C_{write}; (1)$$

Введем переменную $R = \frac{C_{write}}{C_{read}}$,

которая отвечает за асимметрию операций чтения и записи, и преобразуем формулу 1 к нормализованному виду.

$$Cost'_{io} = P_{total}\left(1 + E_{dirty} \cdot R\right); (2)$$

Для подсчёта вероятности промаха используется метод, который предложил Мэтсон Р.Л. в 1970 году [14] для буферов, обладающих свойством вложенности. Буфер обладает свойством вложенности, если при увеличении размера буфера с M до M+K, новый буфер с 1-ой по M-ую позицию будет полностью аналогичен первоначальному.

Для оценки вероятности промаха буфера вводятся специальные счетчики попаданий . При каждом обращении к буферу определяется глубина, при которой $Hit[1, \dots, \infty]$ запрашиваемая страница нашлась бы в пуле. Счетчик, соответствующий данной глубине, увеличивается на единицу. Очевидно, что максимальная глубина пула определяется общим количеством страниц данных БД, но поддерживать информацию обо всех страницах нерационально. Предположим, что мы ограничились поиском только до глубины равной N. Если требуемая глубина пула будет превышать это значение, необходимо увеличивать специальный счетчик $Hit[\infty]$. Обладая такой информацией о попаданиях в пул, можно рассчитать вероятность промаха для буферного пула меньшего размера, чем N.

$$P(M) = 1 - \frac{\sum_{i=1}^{M} Hit[i]}{\sum_{i=1}^{N} Hit[i] + Hit[\infty]}; M \le N; (3)$$

Для поддержки механизма обновления счетчиков достаточно поддерживать виртуальный буфер размера N, а в памяти хранить только те страницы, которые попали на первые M позиций.

Отличие алгоритма FD-Buffer [15] от представленной модели в том, что он поддерживает 2 LRU стека страниц, в одном из них находятся чистые, а в другом грязные страницы. Память между стеками распределяется исходя из принципа минимизации среднего времени обращения к устройству(формула 2). Для оценки вероятности промаха для каждого стека поддерживаются счетчики попаданий, а сама вероятность оценивается с помощью формул, аналогичных формуле 3.

Обозначим стек с чистыми страницами *C*, а с грязными *D*, а их размеры M_c и M_d. Для двух стеков алгоритм доступа к страницам имеет следующий вид.

Algorithm 1 Reads and Writes on FD-Buffer

```
1: function READ(Page p)
2:     Frame v = C.Lookup(p);
3:     if v ≠ NULL then
4:         C.Update(v);
5:     else
6:         v = D.Lookup(p);
7:         if v ≠ NULL then
8:             D.Update(v);
9:         else
10:            v = FindVictimForClean(C, D);          ▷ Algorithm 2.
11:            Загрузить страницу p из носителя в буфер v;
12:        end if
13:    end if
14:    Return frame v;
15: end function

16: function WRITE(Page p)
17:    Frame v = C.Lookup(p);
18:    if v ≠ NULL then
19:        C.Remove(v);
20:        D.Add(v);
21:    else
22:        v = D.Lookup(p);
23:        if v ≠ NULL then
24:            D.Update(v);
25:        else
26:            v = FindVictimForDirty(C, D);          ▷ Algorithm 2.
27:            Загрузить страницу p из носителя в буфер v;
28:        end if
29:    end if
30:    Return frame v;
31: end function
```


Так как размер каждой части буфера определяется динамически, страница жертва выбирается в первую очередь из того стека, размер которого превышает рассчитанный оптимум. В случае если рассчитанный размер не соответствует реальному, помимо выталкивания страниц происходит передача освободившегося места другому буферу.

Для поиска выталкиваемой страницы используются следующие алгоритмы.

Algorithm 2 Page evictions in FD-Buffer

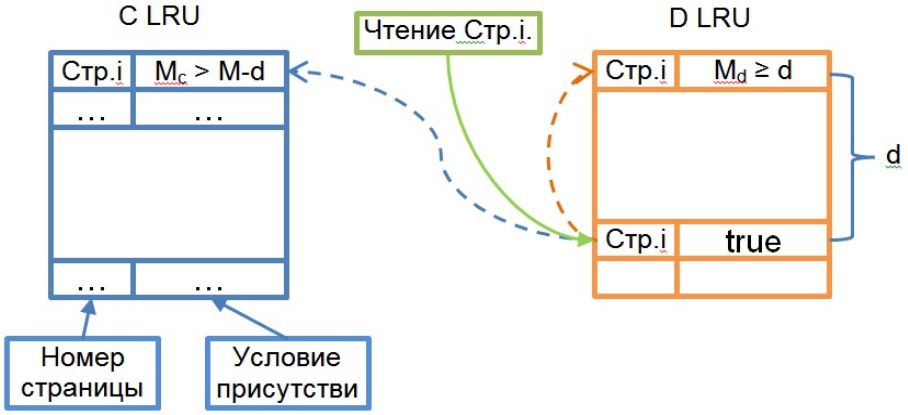
```
1: function FINDVICTIMFORCLEAN(C, D)
2:     if |D| > M_d then
3:         v = D.GetVictim();
4:         D.Remove(v);
5:         C.Add(v);
6:         Записать страницу из v на внешний носитель;
7:     else
8:         v = C.GetVictim();
9:     end if
10:    Return frame v;
11: end function

12: function FINDVICTIMFORDIRTY(C, D)
13:    if |C| > M_c then
14:        v = C.GetVictim();
15:        C.Remove(v);
16:        D.Add(v);
17:    else
18:        v = D.GetVictim();
19:        Записать страницу из v на внешний носитель;
20:    end if
21:    Return frame v;
22: end function
```

Использование 2 стеков сильно усложняет механизм подсчета количества попаданий в пул. Каждый запрос на чтение и запись может попасть как в стек чистых, так и грязных страниц. Обозначим вероятности попаданий в них как P_c и P_d. Тогда для вероятности промаха в целом получим $P_{total} = P_C \times P_D$. Доля грязных страниц, среди всех вытолкнутых, это вероятность промаха по записи в стеке грязных страниц, т.е. $E_{dirty} = P_d^w$. Преобразуем нормализованную стоимость обращения к буферу к следующему виду.

$$Cost'_{io} = P_c \times P_d(1 + P_d^w \cdot R); (4)$$

Как было отмечено ранее, для поддержки механизма подсчета промахов необходимо эмулировать стек большего размера, чем под него реально выделено места. Так как используется 2 стека, то при перемещении страниц

между ними возможна ситуация, когда страница окажется сразу в обоих стеках. Рассмотрим одну из них подробней. Расширим описатель буфера полем с условием присутствия странице в стеке, его смысл будет раскрыт далее.



При попытке доступа к странице на чтение, она может находиться в виртуальном стеке D на глубине d. Классическое поведение предполагает перемещение страницы на вершину стека D, однако, если эта страница реально в нем не находится, то она перемещается на вершину стека C. Так как принятое решение повлияет на будущее увеличение счетчиков попадания в буфер, для расчета оптимального размера стеков необходимо запомнить условие, которое определило местонахождение страницы. Для этого в виртуальных стеках запоминаются оба варианта перемещения страницы и запоминаются условия её попадания в них, при этом реально страница хранится только в одном из них.

Проверку на условие наличия страницы в буфере обозначим PCond. Для поддержки счетчиков попаданий в буфер операции чтения и записи модифицируются следующим образом.

Чтение:

Algorithm 3 Handling a read in Two-Stack

function READHANDLER($Pagep$)
  if $(p \notin$ CLRU$) \wedge (p \notin$ DLRU$)$ **then**
    $Hit_c^r[\infty]++, Hit_d^r[\infty]++$;
    Поместить p на вершину CLRU в буфер e_c;
    $PCond_c[e_c] = true$;
  end if
  if $(p \in$ CLRU $-$ буфер $e_c) \wedge (p \notin DLRU)$ **then**
    * Пусть d_c глубина вхождения e_c в стек CLRU;
    * Рассчитываем такую глубину стека i, чтобы при $M_c \geq i$
    * выполнялось $(M_c \geq d_c) \wedge PCond_c[e_c]$;
    $Hit_c^r[i]++, Hit_d^r[\infty]++$;
    Поместить содержимое e_c на вершину CLRU в буфер e'_c;
    $PCond_c[e'_c] = true$;
  end if
  if $(p \notin$ CLRU$) \wedge (p \in$ DLRU $-$ буфер $e_d)$ **then**
    * Пусть d_d глубина вхождения e_d в стек DLRU;
    * Рассчитываем такую глубину стека j, чтобы при $M_d \geq j$
    * выполнялось $(M_d \geq d_d) \wedge PCond_d[e_d]$;
    $Hit_c^r[\infty]++, Hit_d^r[j]++$;
    Поместить содержимое e_d на вершину DLRU в буфер e'_d;
    $PCond_d[e'_d] = (M_d \geq d_d)$;
    Копировать содержимое e_d на вершину CLRU в буфер e_c;
    $PCond_c[e_c] = (M_c > M - d_d)$;
  end if
  if $(p \in$ CLRU $-$ буфер $e_c) \wedge (p \in$ DLRU $-$ буфер $e_d)$ **then**
    * Пусть d_c и d_d глубина вхождения e_c и e_d в CLRU and DLRU;
    * Рассчитываем такую глубину стека i, чтобы при $M_c \geq i$
    * выполнялось $(M_c \geq d_c) \wedge PCond_c[e_c]$;
    * Рассчитываем такую глубину стека j, чтобы при $M_d \geq j$
    * выполнялось $(M_d \geq d_d) \wedge PCond_d[e_d]$;
    $Hit_c^r[i]++, Hit_d^r[j]++$;
    Поместить содержимое e_c на вершину CLRU в буфер e'_c;
    $PCond_d[e'_c] = (PCond_c[e_c] \vee M_c > M - d_d)$;
    Поместить содержимое e_d на вершину DLRU в буфер e'_d;
    $PCond_d[e'_d] = (PCond_d[e_d] \vee M_d \geq d_d)$;
  end if
end function

Запись:

Algorithm 3 Handling a write in Two-Stack

```
1: function WriteHandler(Page p)
2:     if p ∈ CLRU - буфер e_c then
3:         * Пусть d_c глубина вхождения e_c в стек CLRU;s
4:         * Рассчитываем такую глубину стека i, чтобы при M_c ≥ i
5:         * выполнялось (M_c ≥ d_c) ∧ PCond_c[e_c];
6:         Hit_c^w[i] + +;
7:         Удалить буфер e_c из CLRU;
8:     end if
9:     if p ∉ CLRU then
10:        Hit_c^w[∞] + +;
11:    end if
12:    if p in DLRU at entry e_d then
13:        * Пусть d_d глубина вхождения e_d в стек DLRU;
14:        * Рассчитываем такую глубину стека j, чтобы при M_d ≥ j
15:        * выполнялось (M_d ≥ d_d) ∧ PCond_d[e_d];
16:        Hit_d^w[i] + +;
17:        Поместить содержимое e_d на вершину DLRU в буфер e'_d;
18:        PCond_d[e'_d] = true;
19:    end if
20:    if p ∉ DLRU then
21:        Hit_d^w[∞] + +;
22:        Поместить p на вершину DLRU в буфер e''_d;
23:        PCond_d[e''_d] = true;
24:    end if
25: end function
```

Формулы определения вероятности промахов стеков для подсчета оптимальных размеров областей буфера можно выразить следующим образом.

$$P_c(M_c) = 1 - \frac{\sum_{i=1}^{M_c}(Hit_c^r[i] + Hit_c^w[i])}{\sum_{i=1}^{n}(Hit_c^r[i] + Hit_c^w[i]) + Hit_c^r[\infty] + Hit_c^w[\infty]}; (5)$$

$$P_d(M_d) = 1 - \frac{\sum_{i=1}^{M_d}(Hit_d^r[i] + Hit_d^w[i])}{\sum_{i=1}^{n}(Hit_d^r[i] + Hit_d^w[i]) + Hit_d^r[\infty] + Hit_d^w[\infty]}; (6)$$

$$P_d^w(M_d) = 1 - \frac{\sum_{i=1}^{M_d} Hit_d^w[i]}{\sum_{i=1}^{n} Hit_d^w[i] + Hit_d^w[\infty]}; (7)$$

Так как определение M_C и M_D в режиме реального времени требует много вычислительных ресурсов, пересчет оптимальных размеров стеков выполняется периодически, один раз за несколько обращений к буферу.

Приведенный алгоритм позволяет достаточно точно решать задачу оптимизации среднего времени доступа к странице, поддерживая баланс между различием в стоимости операций чтения и записи страниц данных и высоким процентом попадания в пул. Наличие двух виртуальных буферов, счетчиков попаданий, а также сложных формул расчета вероятности промахов буфера, приводит к большим вычислительным затратам, что может оказаться значительным ограничением в его использовании. Важным дополнением к алгоритму FD-Buffer является внедрение методики кластерной записи.

Для оценки эффективности приведенного алгоритма авторы предлагают рассчитывать среднее время доступа к страницам БД. Так при выполнении теста TPC-C среднее время доступа к страницам снижалось на 30%, по сравнению с алгоритмом LRU. Согласно экспериментальным данным, алгоритм FD-Buffer лишь немного опережает CFDC по производительности и только при условии внедрения кластерной записи.

5. Заключение

Накопители данных на флэш-памяти имеют ряд серьезных преимуществ перед накопителями на жестких дисках, поэтому системы работающие с флэш-памятью демонстрируют большую производительность. Технологическое удешевление стоимости подобных устройств и увеличение плотности хранения информации приводит к некоторым особенностям работы, например к асимметрии скоростей чтения и записи данных. Данные особенности приводят к снижению эффективности буферизации страниц данных – важнейшего механизма повышения скорости работы СУБД. В данной работе рассмотрены три современных алгоритма управления буферным пулом, которые оптимизированы для работы с флэш-памятью. Экспериментальные исследования алгоритмов CFDC и FD-Buffer на тестах производительности TPC-C демонстрируют преимущество до 30% перед классическим алгоритмом LRU. Основной вклад в повышение эффективности вносит использование кластерной записи страниц, при которой грязные страницы выталкиваются из буфера с учетом принципа локальности. К недостаткам представленных алгоритмов можно отнести более высокие вычислительные затраты и потребление памяти. В целом, представленные алгоритмы позволяют использовать преимущества флэш-памяти без потерь производительности связанных с особенностями её работы.

Список литературы

[1] Belady. L., «A study of replacement algorithms for a virtual-storage computer,» *IBM Systems Journal,* p. 78–101, 1966.

[2] Shaul Dar, Michael J. Franklin, Bjorn.T. Jonsson, «Semantic Data Caching and Replacement,» в *VLDB Conference*, 1996.

[3] Masuoka Fujio, Iizuka Hisakazu,, «Semiconductor memory device and method for manufacturing the same,» 1985.

[4] INTEGRATED CIRCUIT ENGINEERING CORP., «Flash Memory Technology,» [В Интернете]. Available: http://smithsonianchips.si.edu/ice/cd/MEMORY97/SEC10.PDF.
[5] Vättö Kristian, «Understanding TLC NAND,» 2012. [В Интернете]. Available: http://www.anandtech.com/show/5067/understanding-tlc-nand/2.
[6] Jesung K., Min J., Sam H., Sang L., Yookun C.;, «A Space-Efficient Flash Translation Layer for CompactFlash Systems,» *Proceedings of the IEEE 48 (2),* p. 366–375, 2002.
[7] Perdue Ken, «"Wear Leveling Application Note,» 2010.
[8] OCZ Technology Group, Inc., «OCZ Vertex 3 SSD Series – 3-rd Generation Harnessing the speed of the SATA III interface.,» 2011.
[9] С.Д. Кузнецов, Основы баз данных, 2007.
[10] Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross,, «SSD Bufferpool Extensions for Database Systems.,» 2010.
[11] D. Seo, D. Shin. , « Recently-evicted-first buffer replacement policy,» *IEEE Transactions on Consumer,* p. 1228–1235, 2008.
[12] «Selected Papers on Flash-Based Database,» *Lab of Web and Mobile data Management,* 2011.
[13] Y. Ou, T. Härder, and P. Jin., «CFDC: a flash-aware replacement policy for database buffer management.,» в *DaMoN*, 2009.
[14] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger., «Evaluation techniques for storage hierarchies.,» *IBM System Journal 9(2),* 1970.
[15] Sai Tung On,Yinan Li,Bingsheng He, Ming Wu, Qiong Luo, Jianliang Xu., «FD-Buffer: A Buffer Manager for Databases on Flash Disks.,» 2010.