электронный журнал
МОЛОДЕЖНЫЙ НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК
Издатель ФГБОУ ВПО "МГТУ им. Н.Э. Баумана". Эл No. ФС77-51038.

УДК 004.021

СПОСОБЫ РЕАЛИЗАЦИ И ДЕДУПЛИКАЦИИ В СИСТЕМАХ ХРАНЕНИЯ ДАННЫХ

Багдасарян Е. А., *студент*
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедры «Система обработки информации и управление»

Елисеев А.В., *студент*
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедры «Система обработки информации и управление»

Научный руководитель:
Черненький М. В.,к.т.н.., доцент
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана
bauman@bmstu.ru

Задача сохранения информации и резервного копирования постоянно актуализируется, создаются новые отказоустойчивые аппаратные средства с энергонезависимой оперативной памятью, бесперебойные системы питания и многое другое. Особое внимание разработчики уделяют программным способам обработки и управления данными, а именно файловым системам с возможностью сохранения работоспособности при отказе нескольких дисков, восстановления данных из произвольного момента времени, уменьшения объема используемых при резервном копировании данных, а также обеспечения контроля доступа к данным.

Сейчас существует проблема выбора ФС для использования, а также определение соответствия требований проекта и возможностей системы.

На рисунке 1 показаны основные этапы обработки данных. Стоит отметить, что представлен краткий список основных действий файловой системы над данными (полный насчитывает несколько десятков параметров [4]). Но основываясь на представленных методах, оставшиеся параметры дополняют логику работы системы, так, например, создание RAID массивов возможно при использовании контрольных сумм и менеджера томов, либо являются широко распространенными, и на их основе невозможно различать ФС. Мы не углубляемся в архитектуру хранения данных и алгоритмы поиска, ведь у каждой системы они свои, а тесты показывают [1], что в производительности пока нет однозначных лидеров. Целью данной методики является помощь в выборе критериев

оценки отказоустойчивых файловых систем в зависимости от их назначения. Итак, рассмотрим каждый из выбранных методов подробно.

Рис. 1. Схема вызываемых методов ФС для записи на диск

Рассмотрим путь данных при записи или чтении файла в операционной системе на базе ядра Linux. Для начала уделим внимание формированию вызова на стороне процессов, которые запрашивают данные и с ними работают. Потом постараемся посмотреть на ключевые действия, которые происходят в ядре при работе с данными, рассмотрим, какие изменения произошли с управлением памятью с течением времени и дадим прогноз на ближайшие перспективы развития технологий работы с данными на ядре Linux.

В ОС при вызове программы (например, системной программы cp, которая перемещает или копирует файл из одной директории в другую) используются системные вызовы open(), read() и write(), соответственно для открытия файлов, чтения исходного файла и записи файла назначения. Во всех вызовах используются флаги, учитывающие различные условия работы с файлами, которые пока не рассматриваются (пусть осуществляется канонический доступ к файлу, то есть чтение блокирует весь пользовательский процесс до завершения чтения, а запись заканчивается после копирования информации в кэш страниц). Каждый из вызовов обрабатывается виртуальной файловой системой, и пользовательская программа не может видеть и влиять на процесс обработки системных вызовов, кроме выставления флагов и изменения путей файлов. Функция read() возвращает поток байтов, которые затем передаются в функцию write(). Чтение файла происходит постранично из области оперативной памяти, которая называется кэш страниц. Если процесс делает системный вызов read(), чтобы получить

несколько байтов, а эти данные еще не находятся в оперативной памяти, ядро выделяет новый страничный кадр, заполняет его соответствующей порцией данных из файла, добавляет страницу в кэш страниц и копирует запрошенные байты в адресное пространство процесса. В общем случае, операция чтения не меняется в различных файловых системах, так как ее цель выяснить расположение данных на диске и отправить запрос блочному устройству. Системный вызов write() различные файловые системы обрабатывают по-разному, к примеру, btrfs использует методы Copy-on-write (копирование при записи), где новые данные записываются не на место старых, а только в свободное пространство. Но все эти операции также скрыты от процесса пользователя. Т.е. ядро Linux максимально упрощает работу с операциями чтения или записи, что позволяет, работать пользовательским программам гораздо быстрее и эффективнее, а также скрывает операции с данными и работу с устройствами от пользовательских процессов. Поэтому процесс считает, что выделенное ему адресное пространство является всей доступной памятью системы и не может никак влиять на работу ядра ОС.

Переходя к описанию работы ядра с оперативной памятью, посмотрим, как виртуальная система обрабатывает системные вызовы и обслуживает кэш страниц [3]. Вся оперативная память делится на страницы в виде 4Кб блоков со своим дескриптором - структурой page, со счетчиком обращений к странице и флагами состояния. Если данные для записи меньше, чем размер страницы памяти, к примеру, индексные дескрипторы файлов, то используется slab-распределение, которое после освобождения объектом своего места на странице оставляет это пространство для объектов такого же типа и размера. Наборы страниц образуют кэши. Кэши используются для хранения всех данных, которые активно используются системой во время работы, такие как различные дескрипторы состояний объектов системы. Главный кэш – kmem_cache содержит все указатели на дескрипторы других кэшей. (Активные кэши со slab можно посмотреть в файле /proc/slabinfo.)

К примеру, файловая система btrfs создает несколько кэшей для поддержки своих функций -

btrfs_free_space_cache

btrfs_path_cache

btrfs_transaction_ cache

btrfs_trans_handle_cache

btrfs_inode_cache

Посмотрим на обработку вызовов read() и write() в виртуальной файловой системе. Функция generic_file_read(), реализующая вызов read(), принимает адрес файлового

объекта, адрес области памяти, в которой должны сохраняться символы, количество символов для чтения и указатель на переменную, содержащую смещение, с которого должно начаться чтение. Получив эти данные и проверив доступ к соответствующим объектам, функция вызывает do_generic_file_read(). Эта функция считывает информацию из кэша страниц, а если там ничего нет, то обращается к диску и запрашивает необходимые блоки для переноса в дисковый кэш. Затем копирует необходимые страницы памяти из кэша страниц в пространство процесса.

Все операции ядро совершает только в оперативной памяти, а не на дисках, так как скорость работы памяти на несколько порядков выше, чем у жестких дисков. Этим обусловлено создание кэша данных, хранимых в оперативной памяти для частого доступа. В результате приложения, использующие одни и те же данные, могут работать быстрее и экономичнее, так как операция доступа к диску не только требует больше времени, но и более энергоемка.

Каждый файл связан с индексным дескриптором, который хранит в себе указатель на структуру address_space, а также адрес данных на диске и является основным параметром для доступа к данным. Используя address_space, ядро получает доступ к данным этого файла в кэше страниц с помощью базисного дерева, которое является частью этой структуры. Базисное дерево помогает найти позицию блока данных, которая нужна процессу. Файлы большого размера разбиваются базисным деревом на страницы и выстраиваются в деревья с узлами с 64-мя потомками, таким образом, можно обрабатывать файлы с размерами до 16 Тбайт. Пример базисных деревьев различной высоты показан на Рисунке 2.

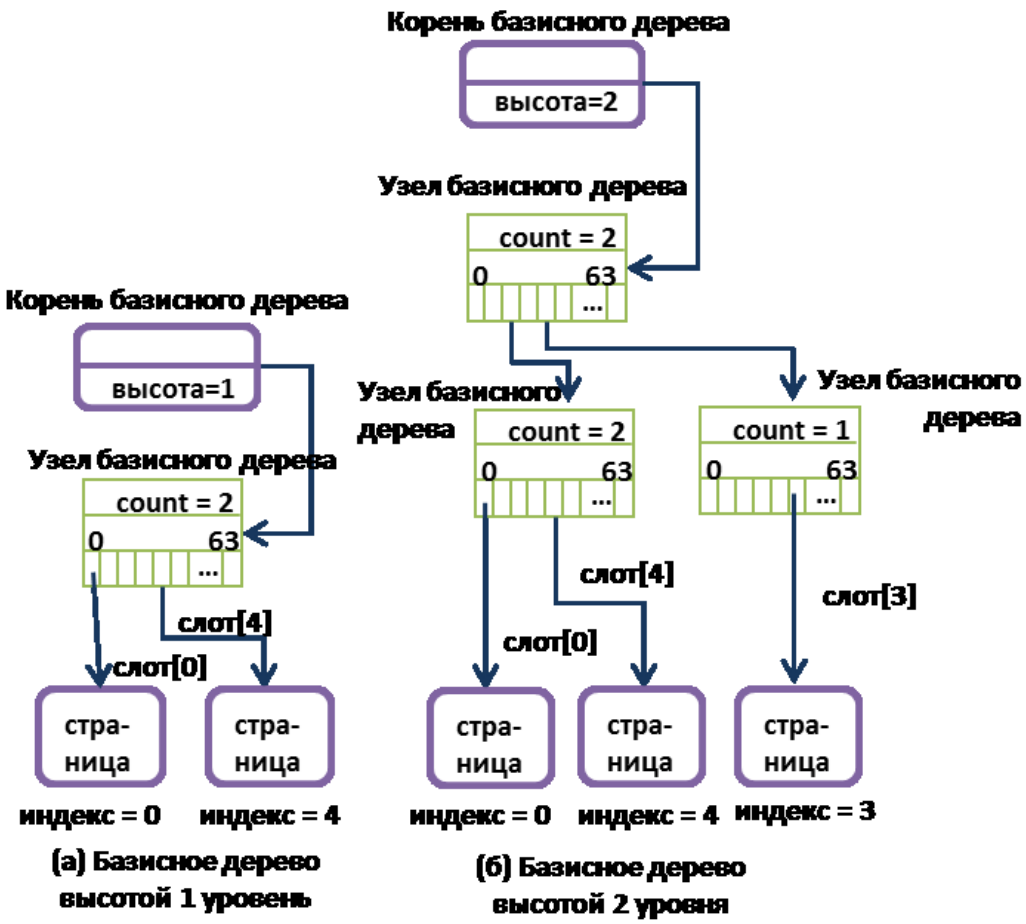

Рис. 2. Структура базисного дерева

При чтении страниц из кэша блоки данных конкретного файла ищутся с помощью базисного дерева, которое имеется для каждого файла в структуре address_space.

При работе с кэшем страниц возникает потребность в записи измененных страниц на диск, чтобы при отключении питания или других сбоях вся проделанная работа не была уничтожена. У ядра существует собственный процесс, который занимается записью блоков, помеченных флагом dirty (которые были изменены в процессе использования). Этот процесс запускается в трех случаях: когда количество «грязных» блоков превышает определенный параметр, когда эти блоки находятся слишком долго в оперативной памяти и при вызове fsync() (процесс заставляет записать данные на диск).

Чтобы упростить процесс поиска «грязных» блоков используются флаги базисных деревьев, каждый узел помечается флагом PG_dirty, если какой-либо из потомков имеет этот флаг. Для страниц, которые не изменялись, используются списки активных и неактивных страниц. В одном содержатся страницы, к которым было обращение недавно, а в другом - страницы, к которым ни один процесс не обращался. Для экономии пространства страницы из неактивного списка постепенно удаляются. Эти списки хранятся в дескрипторах зон оперативной памяти (оперативная память разбита на три

зоны, и для каждой зоны в ядре установлены свои ограничения). Каждая страница памяти из неактивного списка для быстрого перехода по этому списку содержит ссылки на предыдущий и следующий элементы. Ядро старается удалять неактивные страницы из кэша страниц, для этих целей используется процедура с алгоритмом PFRA (Page Frame Reclaiming Algorithm - утилизация страничных кадров)[5]. Этот алгоритм учитывает различные параметры всех страниц оперативной памяти и при необходимости освобождает те, которые можно удалить без влияния на работу ядра. Этот алгоритм запускается в случаях нехватки памяти, выключении компьютера и просто периодически. Благодаря этому алгоритму отсутствует надобность контроля и за размером кэша, так как он меняется динамически в зависимости от загруженности системы. Определенные страницы, которые не имеют своего места на диске, к примеру, промежуточные данные процесса, выгружаются процедурой PFRA и попадают в файл подкачки на диске, хранящий данные до следующего их вызова в оперативную память. Тем самым экономится оперативная память, и пользователь может запускать больше программ для работы.

Если посмотреть развитие способов работы с данными, то в процессе развития ядра Linux, программисты вводили все новые и новые алгоритмы и структуры. Так, для поиска необходимых блоков файла в кэше страниц использовались не деревья, а хэши адресов блоков, по которым происходил поиск. Введения этой технология значительно ускорило работу с кэшем. Также постоянно меняются и адаптируются алгоритмы вытеснения страниц из оперативной памяти, так как нет математически обоснованного алгоритма для вытеснения страниц, и все существующие способы опираются только на экспериментальную эффективность. Если смотреть на весь алгоритм кэширования с точки зрения аппаратной составляющей, то в связи с увеличением популярности твердотельных накопителей, можно смело предположить увеличение длины цепочки процессор - оперативная память-жесткий диск. Между оперативной памятью и жестким диском будет стоять SSD диск, который будет хранить в себе файл подкачки для быстрого доступа к страницам с данными, и содержать в себе расширенный кэш страниц [6]. Также стоит уделить внимание изменению архитектуры компьютера с введением дополнительного процессора для управления памятью, как это сделали с видеокартами.

На сегодняшний день как правило крупные компании хранят свои данные на специальном сервере поддержки системы хранения данных. Главным требованием, предъявляемым к такому серверу, является надёжность доступа к данным. Поэтому, диски с данными организуются специальным образом, образуя RAID-массивы, предохраняющие систему от выхода одного или нескольких дисков из строя. RAID-массивы создают

избыточность данных, так как необходимо, помимо основных данных, записывать контрольные суммы, используемые при восстановлении вышедшего из строя диска. В итоге места для хранения информации требуется больше, к тому же каждый год увеличивается и объём записываемых данных. Для того чтобы сэкономить место на дисках, применяется дедупликация данных.

Дедупликация представляет собой технологию поиска повторяющихся областей на уровне блоков или файлов и замены их соответствующими указателями (рисунок 3). В итоге количество повторяющихся фрагментов резко сокращается, но и увеличивается число метаданных (ссылок на блоки данных) в дисковом хранилище.

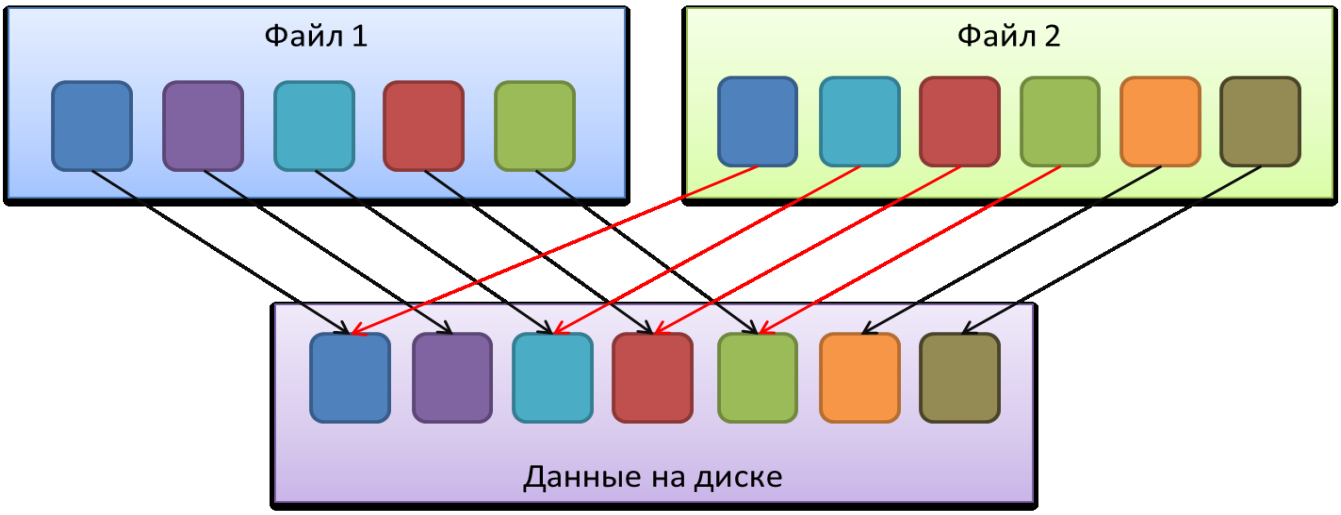

Рис. 3. Технология дедупликации

Различают дедупликацию на файловом и блочном уровнях. При файловой дедупликации области данных представляют собой файлы. Если в систему хранения попадает файл, идентичный присутствующему, то в неё записывается ссылка на имеющийся файл (рисунок 4). Но если пришедший файл отличается от искомого хоть на один байт, то он считается другим и записывается в систему хранения целиком. По этой причине применяется дедупликация на блочном уровне (рисунок 5). При использовании этой технологии каждый файл разбивается на блоки фиксированной либо переменной длины, сравниваются блоки данных. Уникальные данные записываются, а вместо повторяющихся данных записываются ссылки на присутствующие блоки. Такой подход обеспечивает больший коэффициент сжатия, предоставляя меньше места для хранения большего количества информации.

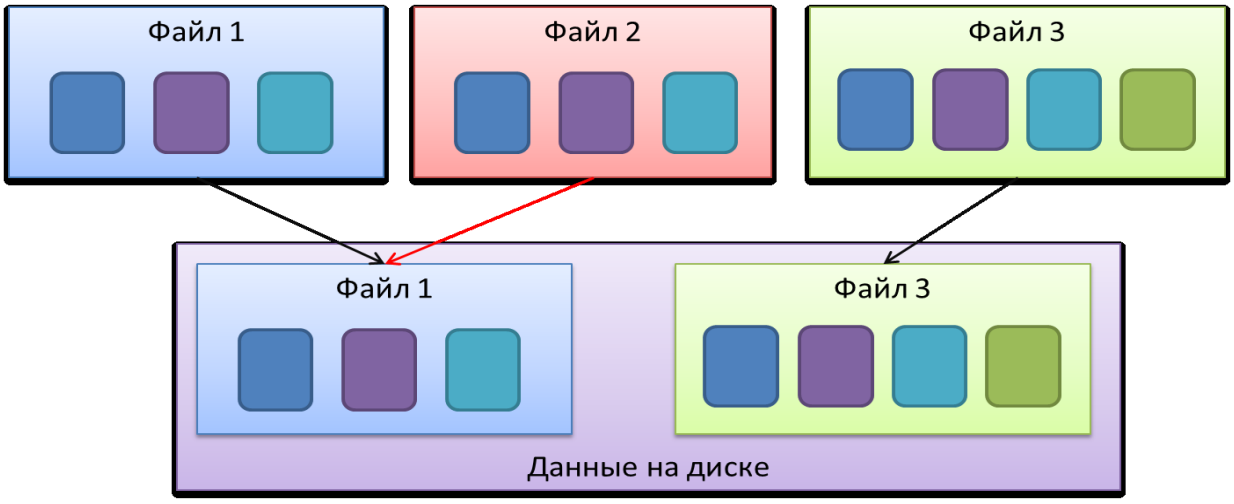

Рис. 4. Файловая дедупликация

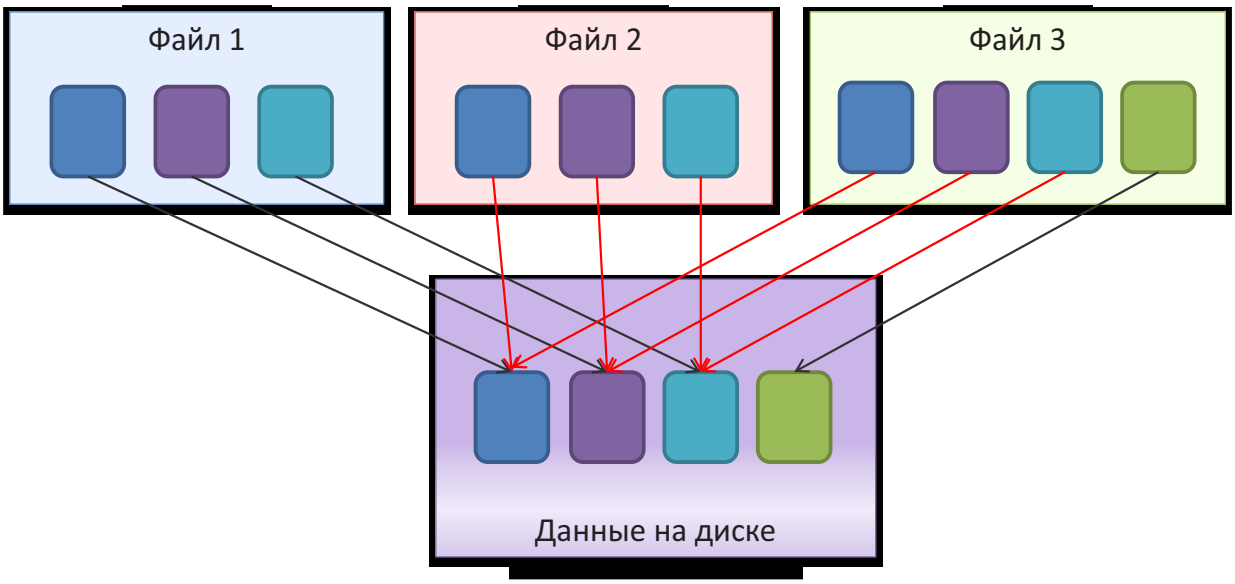

Рис. 5. Блочная дедупликация

С другой точки зрения дедупликация бывает on-line и off-line. On-line дедупликация реализуется до записи данных на диск: сначала блок данных сравнивается с имеющимися блоками, а затем производится запись. Эта операция происходит на стороне клиента, и на сервер передаются не все данные. Такой подход снижает загруженность сети. Однако на клиентской стороне нельзя хранить информацию обо всей системе хранения данных, поэтому сравнение происходит с ограниченным, часто повторяющимся числом блоков данных. В итоге уменьшается коэффициент сжатия. Другой подход, off-line дедупликация, подразумевает выполнение программы дедупликации в самой системе хранения данных в часы меньшей загруженности системы. Данная проверка происходит уже надо всей файловой системой, удаляются повторяющиеся блоки, а вместо них записываются ссылки на уникальные блоки. Этот метод увеличивает коэффициент дедупликации.

Вообще на коэффициент дедупликации влияют сами данные, записываемые в систему хранения. Если система представляет собой хранилище медиа файлов, то повторяющихся блоков будет мало, а следовательно не изменится и коэффициент сжатия. С другой стороны, если на сервер записывается информация о виртуальных машинах, имеющих большое число повторяющихся системных данных, или база данных, то дедупликация существенно экономит место на дисках. Самым эффективным направлением использования данного механизма является система резервного копирования. Каждый раз при записи резервных копий на ленты или жёсткие диски записываются сильно дублируемые данные.

Рассмотрим несколько способов реализации алгоритма on-line дедупликации, которые можно будет применить для систем хранения данных. Как было упомянуто выше, on-line дедупликация осуществляется в оперативной памяти сервера или на стороне клиента. Будем рассматривать случай применения on-line дедупликации на стороне сервера. Она осуществляется в оперативной памяти. Перед записью блоков данных на диски, они попадают в общий буфер, где производится их сравнение по одному из ниже описанных алгоритмов. Буфер представляет собой участок оперативной памяти объёмом в несколько гигабайт.

Первым вариантом осуществления механизма дедупликации является оперирование для сравнения контрольными суммами, которые используются в файловой системе по умолчанию (рисунок 6). В нашем варианте будет использована файловая система Btrfs, которая использует CRC32C контрольные суммы [2]. CRC-суммы быстро считаются, но у них есть один недостаток – возможно состояние хэш-коллизии: ситуации, когда у двух разных блоков одинаковые контрольные суммы. Поэтому помимо сравнения по хэш-функциям производить побайтовое сравнение блоков с одинаковыми контрольными суммами. Данный вариант похож на вариант реализации NetApp, но он будет применятся только в пределах буфера и в оперативной памяти, что существенно увеличит его быстродействие, правда снизит коэффициент сжатия данных.

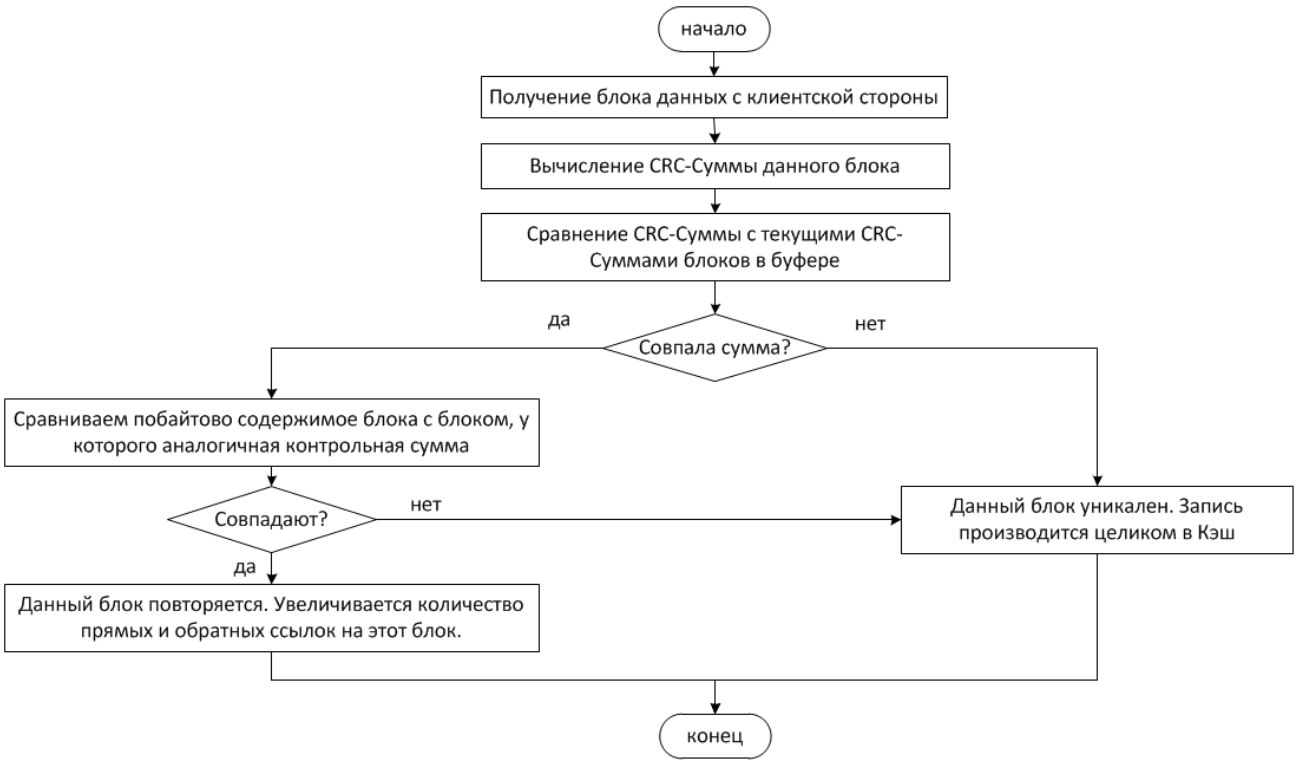

Рис. 6. Блок-схема дедупликации с поблочным сравнением

Второй вариант реализации дедупликации – это использование таких контрольных сумм, у которых вероятность возникновения коллизии очень мала. Вместо стандартных CRC-сумм можно вычислять хэш-функции SHA-256 [10]. При этом вероятность коллизии составляет 2^-256 = 10^-77 – это на 50 порядков меньше, чем ошибка памяти самых надёжных аппаратных средств. Поэтому исчезает потребность осуществлять побайтовое сравнение, и производительность данного алгоритма значительно увеличивается по сравнению с первым вариантом.

Ещё одним вариантом, который повышает коэффициент сжатия данных, является использование журнала повторений при сравнении данных, попадающих в буфер (рисунок 7). Журнал повторений составляется из отсортированных SHA-256 хэш-функций тех блоков, которые чаще всего повторяются в файловой системе. Этот журнал составляется при выполнении off-line дедупликации и полностью хранится в оперативной памяти. Теперь хэш-функция записываемого блока, сравнивается сначала с функциями из журнала повторений, затем с функциями, уже записанных в буфер, а только потом, если хэш уникален, блок записывается целиком, а иначе записывается только ссылка на уже имеющиеся данные.

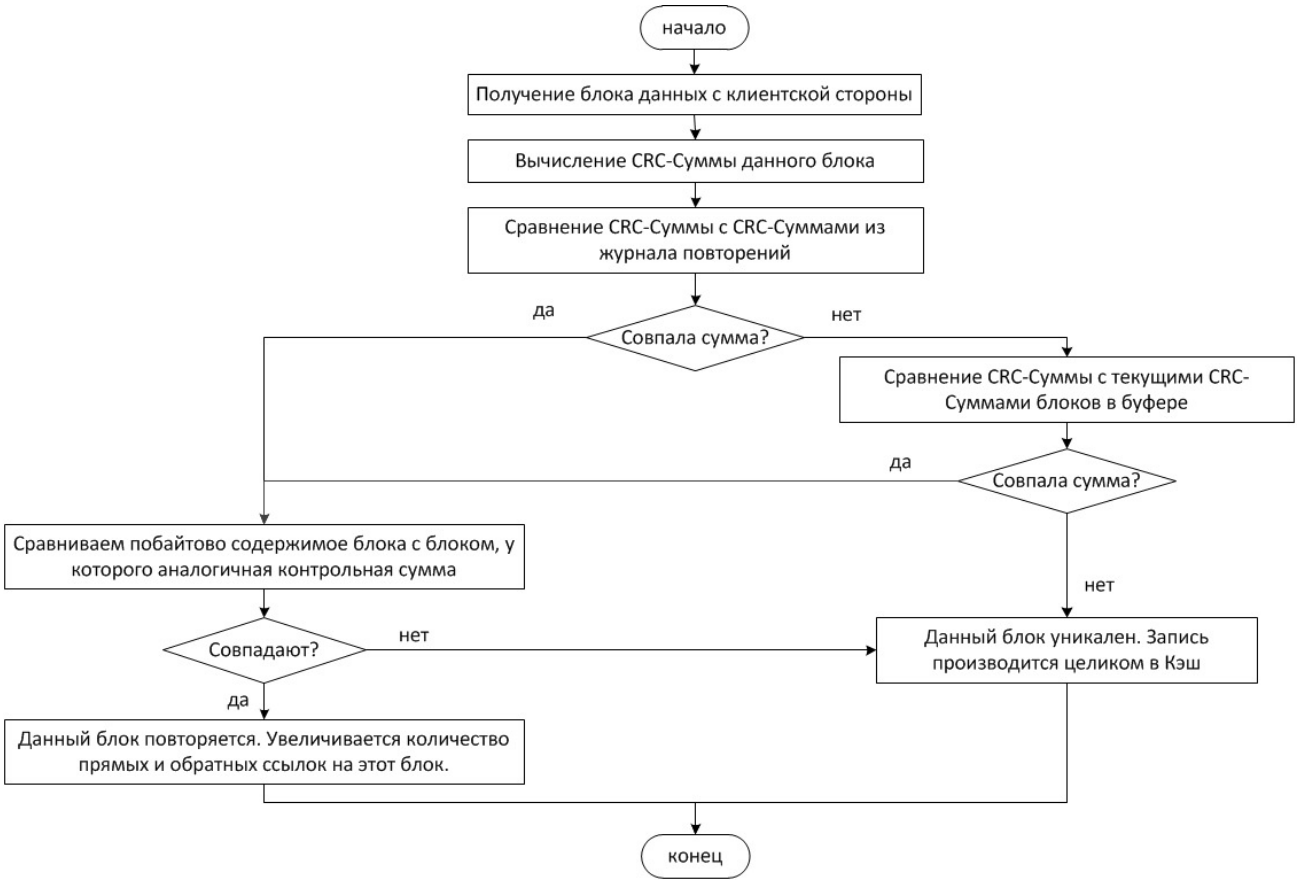

Рис. 7. Блок-схема дедупликации с использованием журнала повторений

Каждый из подходов имеет свои преимущества и недостатки. И выбор того или иного подхода зависит от задач, предъявляемых данной системе хранения данных.

Список литературы

1. Michael Larabel. XFS, Btrfs, EXT4 Battle It Out On Linux 3.4. Режим доступа: http://www.phoronix.com/scan.php?page=article&item=linux_34_fs&num=1. (Дата обращения 21.09.2012г.).
2. Valerie Aurora. A short history of Btrfs. Режим доступа: http://lwn.net/Articles/342892/ (Дата обращения 21.09.2012г.).
3. Р.Лав Разработка ядра Linux. 2-е издание.: пер. с англ.-М.:ООО «ИД Вильямс», 2006. - 448 с.
4. Comparison of file systems. URL: http://en.wikipedia.org/wiki/Comparison_of_file_systems (Дата обращения 21.09.2012г.).
5. Sarp Oral. Efficient Object Storage Journaling in a Distributed Parallel File System. URL: full_papers/oral.pdf (Дата обращения 21.09.2012г.).

6. Rene Mayhofers. SSD Linux benchmarking: Comparing filesystems and encryption methods. Режим доступа: http://www.mayrhofer.eu.org/ssd-linux-benchmark (Дата обращения 21.09.2012г.).
7. Michael Larabel. Real World Benchmarks Of The EXT4 File-System. URL: http://www.phoronix.com/scan.php?page=article& item=ext4_benchmarks&num=1 (Дата обращения 21.09.2012г.).