ISSN 2079-3316 ПРОГРАММНЫЕ СИСТЕМЫ: ТЕОРИЯ И ПРИЛОЖЕНИЯ №3(34), 2017, с. 31–60

УДК 004.62

Н. С. Живчикова, Ю. В. Шевчук

Производительность Riak KV в задаче сохранения сенсорных данных

Аннотация. Система хранения сенсорных данных является важной частью систем анализа данных. Ее задача — принимать данные с временными метками от удалённых источников, сохранять данные и предоставлять доступ к ним по идентификатору датчика и временному интервалу. С ростом числа датчиков становится актуальной задача масштабирования системы. В данной статье мы экспериментально исследуем применение Riak KV — масштабируемого распределённого хранилища типа «ключ-значение» в качестве основы системы хранения сенсорных данных.

Ключевые слова и фразы: сенсорные данные, производительность записи, распределённая система хранения, временной ряд, Riak, Erlang.

Введение

Задача системы хранения сенсорных данных — сохранение информации, поступающей небольшими порциями от большого числа датчиков, причём так, чтобы была возможность эффективного извлечения массива данных (возможно, большого) по идентификатору датчика и временному интервалу — например, для построения графика.

Система характеризуется тремя параметрами: количеством датчиков S, общим объемом информации, поступающим от датчиков в единицу времени A и длительностью хранения Y. При реализации системы хранения сенсорных данных на одиночном компьютере существуют предельные значения параметров S, A и Y, достижимые без замены компьютера на более производительный с нелинейным возрастанием стоимости из-за выхода за границы возможностей компьютеров массового класса. Если целевые значения параметров S, A или Y превышают возможности одиночного компьютера массового

© Н. С. Живчикова, Ю. В. Шевчук, 2017
© Институт программных систем имени А. К. Айламазяна РАН, 2017
© Программные системы: теория и приложения, 2017

DOI: 10.25209/2079-3316-2017-8-3-31-60

класса, привлекательным решением является переход к распределенной архитектуре — использование нескольких таких компьютеров для достижения целевых характеристик.

Можно решать задачу перехода к распределенной системе хранения «с чистого листа», но разумнее попытаться использовать имеющиеся наработки в этой области. Таким образом мы обратили внимание на распределенные NoSQL системы [1]. Помимо повышения производительности и объема системы хранения, переход на распределенную архитектуру обещает повысить доступность системы и устойчивость к отказам аппаратуры.

Среди распределённых NoSQL систем мы сделали выбор в пользу Riak [2] по следующим причинам:

- Riak может работать как система класса AP в терминах CAP теоремы [3]. Это означает, что в условиях нарушения связности системы (выхода из строя каналов связи или узлов) система жертвует согласованностью данных в пользу сохранения доступности. Мы считаем, что система хранения сенсорных данных должна обладать высокой доступностью для того чтобы минимизировать потерю данных;
- Riak это полностью децентрализованная система, все узлы в ней равноправны, нет узлов со специальными функциями, отказ которых повлечет отказ всей системы. Это еще один плюс с точки зрения доступности системы;
- для сохранения данных на диске Riak использует журнальное хранилище Bitcask [4]), в котором одновременная запись идет в небольшое число файлов, а такой режим обещает хорошую производительность записи [5].

Реализация распределенной системы хранения рассматривается с точки зрения обобщённой модели сохранения сенсорных данных (рис. 1). Исходные данные датчиков $s_1..s_S$ группируются в буферном слое $B_1..B_K$. По мере наполнения буферов накопленные данные сбрасываются во вторичную память $D_1..D_Q$ процессами $P_1..P_M$. В [5] рассмотрена реализация этой модели на базе файловых систем ext4 и f2fs в ОС Linux.

Как и в [5], в данной работе мы исследуем производительность сохранения данных, а характеристиками извлечения не занимаемся.

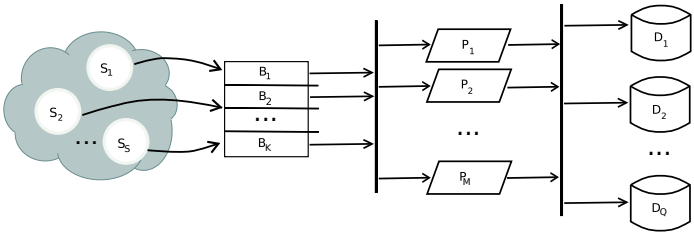


Рис. 1: Обобщенная модель сохранения данных с буферизацией

В то же время, при выборе способа сохранения мы заботимся о том, чтобы эффективное извлечение было возможно.

Статья организована следующим образом. В разделе 1 рассматривается реализация обобщенной модели, использующая Riak в качестве вторичной памяти $D_1..D_Q$. В разделе 2 описываются обнаруженные проблемы и изменения, которые потребовалось внести в Riak, чтобы система справлялась с нагрузкой, характерной для задачи циклического сохранения сенсорных данных. В разделе 3 описаны изменения, которые были внесены в механизм слияния для повышения его эффективности. В разделе 4 представлены результаты тестирования системы. В разделе 5 обсуждаются возможные дальнейшие шаги по повышению эффективности сохранения данных. В разделе 6 предлагается вариант архитектуры масштабируемой системы сохранения сенсорных данных на основе Riak.

1. Реализация обобщенной модели сохранения на базе Riak

Данные в Riak хранятся в форме *объектов*, каждый из которых идентифицируется уникальным *ключом* и содержит *значение*. Существенным отличием объектов Riak от файлов в файловой системе является невозможность пополнения. Строго говоря, пополнение возможно, но это дорогая операция, включающая чтение–модификацию–запись. Поэтому с самого начала мы выбираем вариант однократной записи в объекты. В реализации обобщенной модели на базе файловой системы [5] мы использовали пополнение файлов порциями данных размера B; в случае Riak этому будет соответствовать создание объектов с размером B. Объекты сохраняются с ключом, включающим в себя идентификатор датчика; Riak обеспечивает эффективный доступ к объекту по ключу.

В реализации обобщенной модели на базе файловой системы производительность системы (при постоянном объёме данных в хранилище V) сильно зависела от числа датчиков S, поскольку количество файлов, в которые происходит запись, было пропорционально S. В реализации на базе Riak производительность *не зависит* от S! Действительно, все объекты в хранилище могут относится к одному датчику, или каждый объект относиться к своему датчику — для Riak это выглядит одинаково, как множество объектов с выбранными клиентом ключами. Естественно, при увеличении S придётся уменьшить период хранения данных Y, чтобы сохранить V постоянным.

1.1. Организация тестирования

Тестирование характеристик Riak как общецелевого хранилища проводилось ранее несколькими группами исследователей. В [6] исследуются характеристики Riak при изменении (как увеличении, так и уменьшении) числа узлов кластера: масштабируемость, доступность и эластичность — линейность реакции на сокращение и увеличение числа узлов кластера. В качестве источника тестовой нагрузки используется `basho-bench`[7]. В [8] исследуется влияние конфигурации Riak на время выполнения запросов, доступность в ситуации разделения и согласованность результатов запросов.

В настоящей работе мы тестируем Riak не сам по себе, а в составе будущей системы хранения. Нас интересует производительность записи в мегабайтах в секунду, в то время как в упомянутых выше исследованиях производительность измеряется как число операций в секунду без указания размера объектов Riak. В качестве тестовой программы мы используем фрагмент будущей системы хранения, отвечающий за сохранение данных в Riak, работающий с искусственным потоком данных. В процессе тестирования мы меняем тестовую программу, конфигурацию системы и, при необходимости, Riak, стремясь получить максимальную производительность записи. Достигнув желаемой производительности, мы реализуем систему хранения сенсорных данных как расширение тестовой программы.

Первоначально мы попытались реализовать тестовую программу на языке Си с использованием клиентской библиотеки Riak для языка Си [9] в синхронном режиме. Довольно быстро выяснилось, что однопотоковая синхронная реализация дает низкую производительность из-за естественных задержек в обработке удаленных запросов.

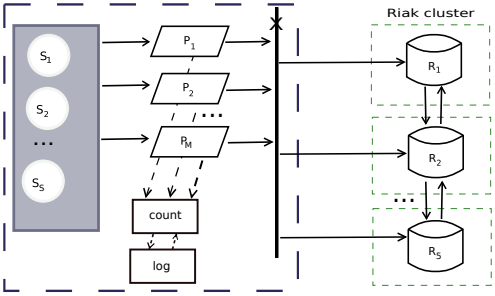


Рис. 2: Организация тестовой программы

Мы рассматривали возможность поднять производительность за счет отправки новых запросов, не ожидая завершения предыдущих — для этого в клиентской библиотеке предусмотрен асинхронный режим. Но более привлекательным решением оказалась реализация тестовой программы на языке Erlang с использованием Riak-клиента для языка Erlang [10]. Erlang позволяет достичь эффекта одновременного выполнения нескольких запросов без использования асинхронного режима. Вместо него используются синхронные запросы в множестве параллельно работающих легковесных процессов Erlang.

Организация тестовой программы показана на рис. 2. Процессы $P_1..P_M$ обращаются к Riak с запросами `put`, в которых значение это блок произвольных данных размера B, а в качестве ключа используется 32-битное число, в котором младшие 24 бита представляют собой номер датчика ($1..S$), а старшие 8 бит идентификатор процесса ($1..M$). Процессы P_i имитируют поочередное циклическое поступление информации от датчиков $1..S$.

По завершении каждого запроса `put` процессы P_i отправляют сообщение процессу `count`, который учитывает объем записанной в Riak информации. Раз в минуту процесс `log` запрашивает у процесса `count` значение счетчика и записывает накопленное значение в лог-файл, а процесс `count` начинает новый цикл накопления с нуля.

Объем актуальных («живых» в терминах Riak) данных V, которые появляются в хранилище Riak в ходе теста, зависит от параметров тестирования следующим образом:

Таблица 1: Характеристики узлов Riak

Процессор	Тип	Intel(R) Core(TM) i5-4670
	Частота	3400 МГц
	Число ядер	4
Оперативная память	Тип	DDR3
	Объем	2048 МБ x 2
	Частота	1333 МГц
Операционная система	Версия	Debian 3.16.7
	Архитектура	amd64
	Версия ядра	3.16.0-4-amd64
Накопитель HDD	Модель	TOSHIBA HDWD105
	Объем	500 ГБ
	Обороты	7200 Об/мин
	Интерфейс	SATA 6Гб/с
Сетевой интерфейс Ethernet	Модель	Intel I217-V
	Драйвер	e1000e
	Стандарт	1000BASE-T

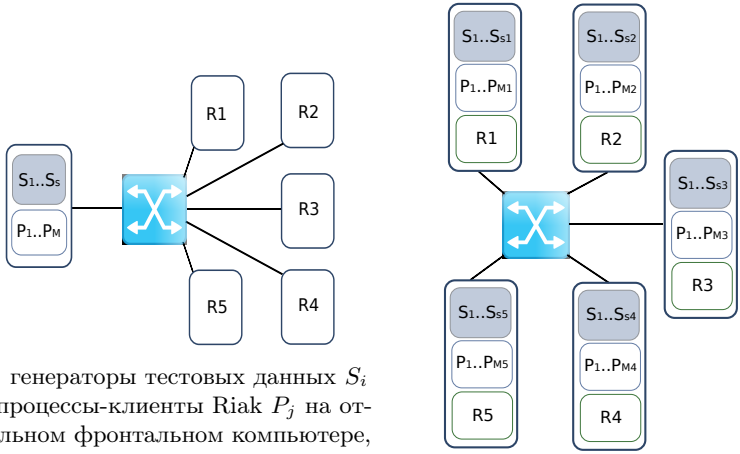
$$V = \frac{SBM}{C}, \tag{1}$$

где S — число датчиков, B — размер буфера данных, соответствующий размеру объекта Riak, M — число процессов, обращающихся к Riak с запросами `put`, C — суммарный объём файловых систем узлов кластера Riak, V — доля живых данных в общем объёме данных C.

Вне зависимости от продолжительности теста объем живых данных не превышает вычисленного значения V, так как в установившемся режиме количество объектов в хранилище не изменяется, вновь сгенерированные данные заменяют прежде ассоциированные с ключами значения.

Объем занятого дискового пространства в файловых системах узлов, где хранит свои файлы Bitcask ($\sum_{i=1}^{N} U_i$), может быть значительно выше, чем V; далее мы остановимся на этом подробнее.

Конфигурация узлов кластера, на которых работает Riak, приведена в таблице 1. Мы использовали два варианта тестовой установки. В первом варианте единственный генератор тестовых данных нахо-



(а) генераторы тестовых данных S_i и процессы-клиенты Riak P_j на отдельном фронтальном компьютере, связанном с компьютерами-узлами Riak $R_1 \ldots R_5$ коммутатором Gigabit Ethernet

(б) генераторы S_i и клиенты Riak P_j на каждом узле Riak

Рис. 3: Схемы тестовой установки

дился на фронтальном компьютере (рис. 3). По мере улучшения производительности установки в ходе работы узким местом стал сначала сетевой интерфейс фронтального компьютера, а потом и интерфейсы узлов кластера Riak, через которые не только поступают данные на сохранение, но и происходит обмен между узлами кластера. Тогда было принято решение о распределённом приёме данных системой (раздел 6), и дальнейшее тестирование происходило на втором варианте установки, где на каждом узле кластера работает свой генератор тестовых данных, имитирующий поступление данных на этот узел от внешнего источника через дополнительный сетевой интерфейс (рис. 3б).

Riak сконфигурирован с коэффициентом репликации `n_val=1`, чтобы получить оценку производительности сверху. Включение репликации снизит производительность и повысит расход ресурсов системы (в первую очередь дискового пространства и пропускной способности сети). Тестов для исследования этих эффектов в данной работе не проводилось. В Riak очень гибкие механизмы настройки репликации; провести эти тесты имеет смысл после того, как выбрана конфигура-

ция репликации исходя из требований конечного пользователя.

Все тесты проводились с включённым механизмом AAE (active anti-entropy) Riak. Отключение AAE даёт прирост производительности около 10%, но пользоваться этим можно только в случаях, когда потеря данных не критична и репликация не используется.

При построении некоторых из представленных ниже графиков использовалось сглаживание данных с помощью фильтра «экспоненциально взвешенное скользящее среднее» [11]. Факт использования сглаживаения отображается в легенде графика записью $\mathsf{EWMA} = n$, отражающей константу сглаживания $\alpha = 1/n$.

2. Поведение Riak при интенсивной циклической записи

Первые же попытки запуска тестовой программы на Erlang дали неожиданный результат. Вскоре после запуска теста объем использованного дискового пространства в файловой системе (U) достигает 100% и Riak перестает обрабатывать запросы на сохранение: запрос `put` возвращает ошибку `{error,enospc}`. Этого не должно было происходить, так как заданный параметрами эксперимента объем живых данных V был значительно меньше размера файловой системы C.

Мы обнаружили две причины, вызывающие такое поведение.

2.1. Причина 1: дисбаланс между сохранением и слиянием

В наших экспериментах Riak сконфигурирован с хранилищем Bitcask[4]. Bitcask это хранилище журнального типа: данные, поступающие в запросе на сохранение (`put`), записываются в один большой файл. Этот файл ротируется, когда достигается пороговое значение (1ГБ). В оперативной памяти хранится индекс (*keydir* в терминологии Bitcask), содержащий соответствие между ключами и позициями в текущем и ротированных файлах, где находятся соответствующие ключам данные.

Если запрос (`put`) содержит ключ, уже существующий в хранилище, соответствующая ключу запись в индексе обновляется и начинает указывать на новую позицию в файле. Предыдущие данные устаревают (становятся «мёртвыми» в терминах Bitcask); они продолжают храниться в файле, но запросы на получение значения по данному

ключу (`get`) будут обращаться не к ним, а к новому значению согласно *keydir*.

Чем больше ключей обновляется и чем чаще это происходит, тем больше возникает мёртвых данных и тем быстрее возникает потребность в удалении устаревших данных и освобождении места для новых. Процесс освобождения хранилища от мертвых данных в Riak называется «слиянием».

Процесс слияния работает параллельно с Bitcask API, который обрабатывает клиентские запросы `put`, `get` и т.д. Процесс слияния работает с ротированными файлами; он прочитывает последовательно один или более ротированных файлов, пропускает мёртвые данные, а живые данные записывает в новый файл — результат слияния. Обработанные файлы после слияния удаляются, что приводит к освобождению пространства в файловой системе для сохранения новых данных.

Эта схема работает хорошо, пока запросы `put` происходят сравнительно редко. Но когда мы нагружаем Riak интенсивным потоков запросов `put`, заменяющих значения существующих ключей, объем мёртвых данных растет быстрее, чем процесс слияния успевает их удалять. В конце концов процесс слияния обнаруживает, что в файловой системе недостаточно места для создания нового файла–результата слияния и мёртвые данные перестают удаляться. Это и есть наблюдаемое в тесте «безвыходное положение» — недостаточно места для работы слияния и нет места для сохранения вновь прибывающих данных.

2.2. Причина 2: незакрытые дескрипторы удаленных файлов

Было бы естественно ожидать, что в тот момент, когда процесс слияния удаляет обработанные файлы, на диске должно освободиться место, соответствующее размеру удаленных файлов. Но в случае Riak это происходит далеко не всегда. Мы видим в логе сообщение о том, что файл удален в ходе слияния, утилита `ls` больше не видит этот файл, но по данным утилиты `df` заполнение файловой системы U продолжает увеличиваться, хотя в результате слияния должно уменьшиться. Это известный эффект: файл, на который нет ссылок из каталогов, не удаляется (занимаемые им блоки не освобождаются)

файловой системой, пока есть процессы, которые этот файл используют (имеют открытый дескриптор для доступа к этому файлу).

Оказывается, происходит вот что. Bitcask кэширует файловые дескрипторы. Любой дескриптор, открытый для выполнения операций `get` или `put`, остается в кэше в расчете на то, что он понадобится еще раз. Очевидно, эти дескрипторы должны быть закрыты и удалены из кэша, когда процесс слияния удаляет файл с диска, но Riak так не делает и убедить его делать так в текущей реализации непросто.

Bitcask реализуется большим числом Erlang-процессов: отдельный процесс отвечает за слияние, каждый клиент Bitcask API это также отдельный процесс. Файловые дескрипторы кэшируются в области приватных данных («process dictionary») процесса-клиента Bitcask API. Один файл может быть закэширован более чем одним процессом-клиентом. Например, сначала файл был открыт в процессе выполнения запроса `put`, дескриптор остался в кэше процесса, вызвавшего функцию `put` Bitcask API. Потом этот файл был ротирован, передан на слияние и, наконец, удалён. Но дескриптор в кэше остался открыт, а файловая система не может освободить блоки файла, пока файл открыт каким-либо процессом. Итак, файл реально не удаляется, занятые им блоки не освобождаются.

Для немедленного освобождения блоков нужно при удалении файла сообщить всем процессам-клиентам о том, что файл удален и открытые для доступа к нему дескрипторы должны быть закрыты и удалены из кэша. К сожалению, простого способа сделать это в программе на Erlang нет.

Мы встречали упоминание этой проблемы (датируется 2013 годом [**12**], [**13**]), но проблема по-прежнему актуальна и для самой последней на данный момент версии Riak KV 2.2.3. Возможно, большинство пользователей Riak не замечают её, потому что их хранилища Riak существенно больше по объему, число файлов в них велико и запросы `get` естественным образом вытесняют из кэша старые дескрипторы, так что эффект от проблемы гораздо менее заметен.

В нашей тестовой конфигурации хранилище сравнительно небольшое (всего 100ГБ на каждый узел), при пороге ротации в Bitcask

равном 1ГБ число файлов оказывается тоже небольшим (100) и на естественное вытеснение дескрипторов из кэша рассчитывать не приходится, тем более что запросов `get` в тесте нет вообще. Поэтому в нашей конфигурации проблема проявляется остро и требует решения.

2.3. Устранение дисбаланса между записью и слиянием

Чтобы не попадать в описанное выше «безвыходное положение», можно снижать частоту запросов `put`, когда заполнение файловой системы U приближается к 100%, чтобы освободить ресурсы подсистемы ввода-вывода и процессора для работы процесса слияния. Вариант с регулированием частоты запросов на стороне клиента Riak (в нашем случае это тестовая программа) не подходит, так как при работе с распределенным хранилищем клиент не может знать, какой узел кластера будет обрабатывать очередной запрос `put`, а значит не может достоверно вычислить заполнение диска. Мы решили реализовать регулирование потока запросов на стороне Riak, используя тот факт, что клиенты работают с хранилищем синхронно: не отправляют следующий запрос, пока не получат от Riak ответ на текущий запрос. Таким образом, достаточно добавить переменную задержку в обработку запроса `put`, и варьировать эту задержку в зависимости от заполнения файловой системы U на момент обработки запроса.

Для этой цели мы определяем два дополнительных параметра в конфигурации Riak — пороговые значения заполнения файловой системы U:

U_l нижний порог: в этой точке и ниже нее задержка равна нулю;

U_h верхний порог: в этой точке задержка достигает 1 секунды.

Для текущего уровня заполнения файловой системы U задержка вычисляется по формуле:

$$T = \max\left(0, \frac{U - U_l}{U_h - U_l}\right)$$

Введение задержки дало желаемый результат: процесс слияния успевает справляться с удалением мертвых данных и тупиковых ситуаций больше не возникает. В тесте использовались значения $U_l = 95\%$ и $U_h = 100\%$.

2.4. Устранение незакрытых дескрипторов удаленных файлов

Когда процесс слияния удаляет файл, нужно дать знать клиентским процессам, что файл удален и его нужно удалить из кэша.

Первая мысль, которая приходит, это использование схемы на основе событий. Процесс слияния мог бы посылать клиентским процессам широковещательное сообщение «файл X удален». Правда, в Erlang нет такого понятия как «широковещательное сообщение». Придется поддерживать список идентификаторов клиентских процессов в памяти процесса слияния; это не очень удобно, но возможно. Далее, клиентский процесс не является частью Bitcask: это процесс из Riak, который вызывает функции Bitcask API. Если этому процессу послать сообщение, он сможет отреагировать на него (удалить файл из кэша) только при очередном запросе `put` или какой-либо другой операции Bitcask API — то есть в общем случае мы получаем непредсказуемую задержку. Кроме того, процесс может пользоваться сообщениями в своих целях и незнакомое сообщение может его дезориентировать. Таким образом, сигнализация при помощи сообщений в данном случае неприменима.

Правильным решением представляется реорганизация кэша дескрипторов: вместо множественного кэширования файловых дескрипторов внутри клиентских процессов, реализовать общий для всех процессов кэш в виде именованной ETS-таблицы. Эта таблица будет доступна как клиентским процессам (пополняющим кэш и использующим существующие в нем дескрипторы), так и процессу слияния, который сможет удалять дескрипторы из кэша без участия клиентских процессов. Но это довольно серьезная модификация Bitcask.

Для целей настоящих экспериментов оказалось достаточно следующего временного решения. Каждый клиентский процесс при каждом входе в API, но не чаще чем раз в 30 секунд, проверяет свой кэш на наличие мертвых дескрипторов: файлы проверяются на существование при помощи системного вызова `stat`. Это решение неполноценное, поскольку проверка делается только при входе клиентского процесса в Bitcask API, а рассчитывать на регулярный вход клиента в API в общем случае нельзя. Но в наших экспериментах пользовательские

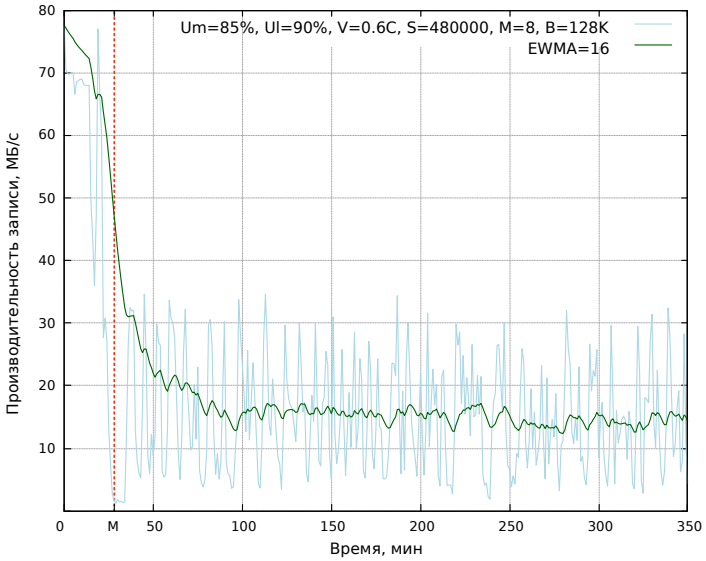


Рис. 4: Изменение производительности записи в ходе теста. Точка M — начало процесса слияния.

процессы часто обращаются к Bitcask, и поэтому задержка между удалением файла и освобождением дескриптора никогда существенно не превышает период проверки кэша.

3. Повышение эффективности слияния

С модификациями из раздела 2 тест перестал аварийно завершаться и стало возможно изучение производительности.

Рассматривая изменение производительности записи в ходе теста (рис. 4) мы видим очень неплохую скорость в начале теста, а затем резкий спад, совпадающий с моментом начала работы слияния. В начале теста, пока уровень заполнения U низкий, слияние не требуется, файловая система получает только запросы на запись в немногочисленные файлы Bitcask. С активацией процесса слияния появляются запросы на чтение ранее записанных файлов, которые конкурируют с запросами на запись и являются важной причиной снижения производительности.

Из опыта работы с Riak складывается ощущение, что разработчики ориентировались в первую очередь на приложения, в которых преобладают запросы на получение данных (`get`), а запросы на удаление (`delete`) или обновление (`put`) данных сравнительно редки и необходимость в слияниях возникает нечасто. В конфигурации Riak даже есть параметр, позволяющий запускать операции слияния только в определенное время суток (`bitcask.merge.window` [14]).

В задаче циклического сохранения сенсорных данных ситуация обратная. Мы имеем дело с плотным потоком запросов `put`, обновляющих данные для существующих ключей, так что необходимость в слиянии существует постоянно. Таким образом, для повышения производительности необходимо работать над повышением эффективности слияния.

3.0.1. *Управление очередью кандидатов на слияние*

В riak за слияние отвечают несколько процессов Erlang. В каждом *vnode*[1] отдельный Erlang-процесс следит за ротированными файлами в хранилище и на основании параметров конфигурации [14] принимает решение, какие файлы будут участвовать в слиянии. Список файлов отправляется в очередь на слияние. Непосредственно слиянием занимается один Erlang-процесс на каждом физическом узле. В порядке очереди он последовательно прочитывает ротированные файлы, пропускает мертвые данные, а живые данные переписывает в новый файл — результат слияния.

Файлы, которые находятся в очереди, принадлежат разным vnode и содержат разную долю мёртвых данных D. Эффективность слияния должна быть прямо пропорциональна D: чем больше доля мёртвых данных, тем меньше живых данных потребуется переписать в файл – результат слияния, и тем больше пространства освободится в ре-

[1] vnode в Riak — виртуальный узел, отвечающий за хранение объектов с ключами, попадающими в назначенный данному идентификатору vnode сегмент хэш-пространства системы. Количество виртуальных узлов в системе задаётся конфигурацией Riak и обычно в несколько раз превышает число физических узлов. Виртуальные узлы распределяются по физическим узлам кластера и могут перераспределяться при добавлении новых узлов или отключении узлов от системы

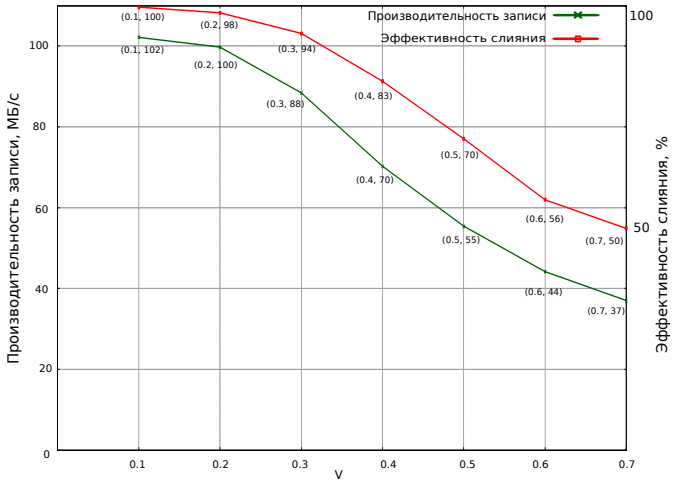


Рис. 5: Зависимость производительности записи и средней доли мёртвых данных D от доли живых данных в хранилище V

зультате слияния. И действительно, на рис. 5 видна связь между производительностью и средней долей мёртвых данных в выбранных на слияние кандидатурах, позволяющая заключить, что рост производительности напрямую связан с повышением эффективности слияния. Таким образом, можно рассчитывать на повышение эффективности слияния, если не передавать на слияние все поступившие в очередь кандидатуры в порядке FIFO, а выбирать кандидатуры с наибольшей долей мёртвых данных D.

3.0.2. *Запуск слияния по заполнению файловой системы U*

В конфигурации Riak существуют параметры для настройки запуска процесса слияния [14]. В качестве критерия запуска используется фрагментация файлов Bitcask: доля мертвых ключей или мертвых данных в каждом из файлов. Как выяснилось, для приложений с интенсивной записью существующие параметры не позволяют адекватно управлять процессом слияния. Если установить пороговые значения слишком высоко, возникает риск оказаться в тупиковой ситуации из-за переполнения файловой системы. Если установить пороговые значения слишком низко, слияние будет работать вхолостую и мешать обрабатывать запросы `put`. Нам хотелось бы, чтобы процесс

слияния поддерживал заполнение файловой системы U на некотором уровне, скажем 90%, но необходимые для этого пороговые значения фрагментации невозможно установить статически, они зависят от количества файлов Bitcask, которое в результате слияний меняется непредсказуемо.

С течением времени доля мёртвых данных в файлах Bitcask только увеличивается. Таким образом, чем позднее мы запускаем процесс слияния, тем эффективнее он работает — тем больше мёртвых данных удаляется при равных затраченных на слияние усилиях. Это наблюдение наводит на мысль, что вообще не нужно запускать слияние, пока заполнение файловой системы U не достигнет заданного администратором системы порога. Riak должен отслеживать заполнение файловой системы, используя утилиту `df` или системный вызов `statfs` и запускать слияние только тогда, когда пора освобождать место на диске.

3.0.3. *Быстрое слияние полностью мёртвых файлов*

Важный частный случай представляют собой файлы, в которых все 100% данных мёртвые ($D = 1$). Для таких файлов слияние как таковое не требуется: файл не нужно читать, чтобы извлечь оставшиеся живые данные, его можно просто удалить, что гораздо эффективнее обычной процедуры слияния.

3.0.4. *Алгоритм рыболова*

В реализации управления слиянием мы будем использовать алгоритм, схожий с поведением рыболова, который ловит рыбу и ест её, чтобы утолить голод. Опишем его неформально. Пока рыболов сыт, он отпускает мелкую рыбу, берёт только самую крупную. Если время идёт, а крупная рыба всё не попадается, голод начинает беспокоить рыболова, он снижает требования и перестаёт отпускать среднюю рыбу. Если средней рыбы ловится достаточно для утоления голода, рыболов питается ей, а мелкую продолжает отпускать. Если средней рыбы ловится мало, голод не удовлетворён, рыболов продолжает снижать требования, он готов брать всё более и более мелкую рыбу. Со временем рыболов приноравливается к рыбе, которая ловится в данном месте: он уже знает, какую рыбу брать, какую отпускать, чтобы не голодать, но и не есть мелкую рыбу без необходимости.

Будем обозначать размер рыбы D: самой крупной рыбе соответствует $D = 1$, самой мелкой $D = 0$. Найденный рыболовом порог отпускания рыбы обозначим D_m. Лучшую рыбу ($D = 1$) рыболов никогда не отпускает, даже если совсем не голоден, но она попадается не настолько часто, чтобы рыболов скончался от обжорства.

В применении алгоритма к задаче управления очередью на слияние размер рыбы соответствует доле мёртвых данных в файле-кандидате на слияние D. Чувство голода определяется как $H = (U - U_m)/(1 - U_m)$, где U – текущий уровень заполнения файловой системы $(0\ldots1)$, U_m – заданный администратором уровень заполнения файловой системы, который должен поддерживать алгоритм.

Файл-кандидат направляется на слияние при условии $D \geq D_m$, где

$$D_m = \begin{cases} \min(1 - H, \mathit{ewma}(D_i)) \text{ при } H > 0 \\ 1 \text{ при } H \leq 0 \end{cases}$$

Член $\mathit{ewma}(D_i)$ — экспоненциально взвешенное скользящее среднее от размера всех ранее съеденных рыб — выражает память рыболова о том, какая рыба годится в пищу. Также D_m зависит от голода H, благодаря чему происходит обучение: если голод заставил рыболова взять более мелкую рыбу, чем обычно, *ewma* понизится и в дальнейшем рыболов даже без острого голода будет брать более мелкую рыбу. С другой стороны, если при низком D_m попадается рыба более крупная, *ewma* повышается в надежде, что дальше будут ещё такие же, не стоит торопиться брать мелкую рыбу.

4. Тестирование

Параметры всех проводимых экспериментов связаны отношением 1 (стр. 36). Мы запускаем одну и ту же тестовую программу при разных значениях параметров, чтобы рассмотреть интересующие нас аспекты поведения системы.

4.1. Зависимость производительности от заполнения хранилища V

В первую очередь нам интересно, насколько принятые в разделе 3 меры по повышению эффективности слияния помогли устранить спад производительности с началом слияния. Мы повторили тесты, пока-

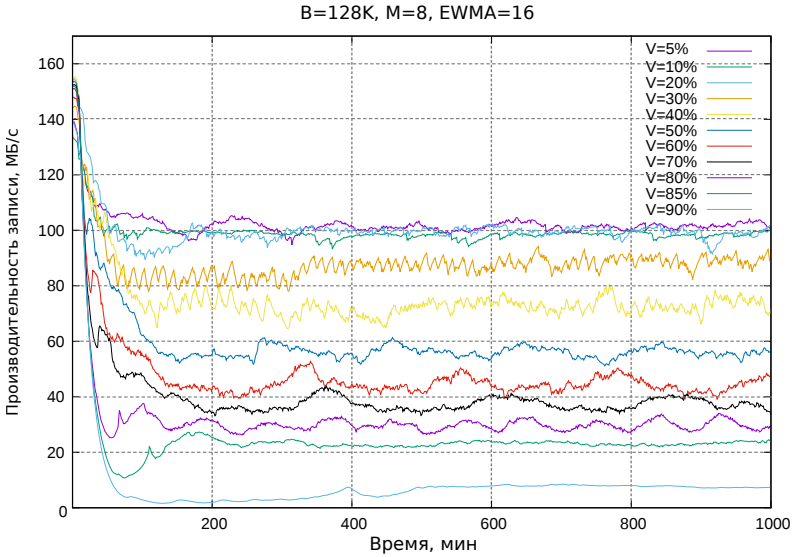


Рис. 6: Изменение производительности в ходе теста при разных уровнях заполнения хранилища данными V

занные на рис. 4,5; результаты представлены на рис. 6. Изменение V достигалось изменением числа датчиков S. Тестирование проводилось на одном узле Riak ($N = 1$): поскольку слияние выполняется автономно каждым узлом, не имеет смысла рассматривать процесс на нескольких узлах.

Действительно, спад скорости по сравнению с рис. 4 стал меньше. Особенно ярко видно улучшение при малых уровнях заполнения V: в этих тестах на слияние поступают файлы с $D = 1$, система работает в области A (рис. 7), где слияние максимально эффективно благодаря оптимизации из раздела 3.0.3.

При $V = 0.9$ система работает в области C (рис. 7): на слияние поступают файлы с небольшой долей мёртвых данных, эффективность слияния низкая и система вынуждена задействовать задержку в запросах `put` (раздел 2.3). Отметим, что после повышения эффективности слияния при $V < 0.9$ система обходится без включения задержки, что можно считать достижением.

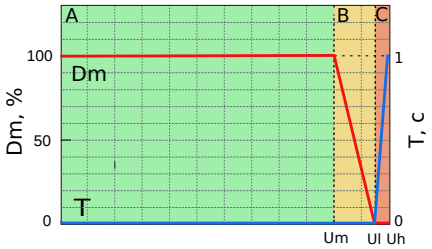


Рис. 7: Области зависимости средней доли мёртвых данных D от уровня заполнения V

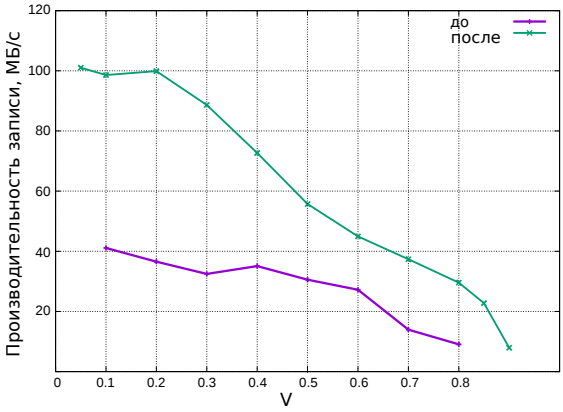


Рис. 8: Зависимость производительности от заполнения хранилища данными V

Наглядно эффект от мер по повышению эффективности слияния показан на рис. 8.

4.2. Зависимость производительности от порога ротации файлов Bitcask R

Максимальную производительность система демонстрирует в области A (рис. 7), где удаётся удовлетворять потребность в освобождении дискового пространства исключительно за счёт слияния файлов с $D = 1$, так что благодаря оптимизации раздела 3.0.3 слияние практически не занимает ресурсов системы. Этот режим напоминает движение судна в режиме глиссирования, когда сопротивление воды движению

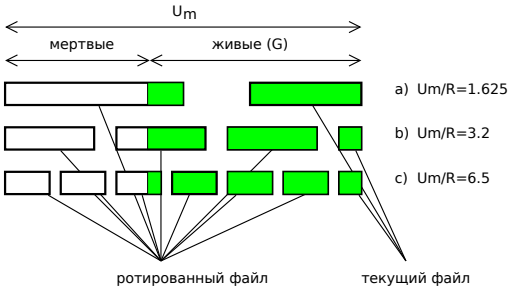


Рис. 9: Распределение живых данных в файлах Bitcask на одном vnode в момент $U = U_m$ при разных значениях порога ротации файлов R

резко уменьшается; будем использовать этот термин для обозначения работы системы в области A.

В тестах раздела 4.1 мы достигали глиссирования снижением объёма живых в данных в хранилище V, то есть снижением эффективности использования дискового пространства. Глиссирование без снижения V означает, что к моменту, когда заполнение файловой системы U на некотором узле Riak достигает порога U_m, хотя бы в одном из vnode на данном узле нашёлся файл с $D = 1$ для эффективного слияния.

Среднее число файлов Bitcask в виртуальном узле (vnode) в момент $U = U_m$ определяется выражением

$$F = \frac{C_i}{n_i}\frac{U_m}{R} \tag{2}$$

где R – порог ротации файлов Bitcask, заданный параметром конфигурации `bitcask.max_file_size`, C_i — объём файловой системы на физическом узле Riak с номером i, n_i — количество vnode на узле с номером i. Тест устроен так, что у всех объектов одинаковое время жизни, то есть чем раньше объект появился в Riak, тем раньше он отмирает. Таким образом, живые и мёртвые данные распределяются по файлам Bitcask как показано на рис. 9.

Рассматривая рис. 9, мы можем выписать условие глиссирования:

$$G \le (F - 1)R \tag{3}$$

где $G = V\frac{C_i}{n_i}$ — объём живых данных в одном vnode. После подстановки значений F и G в условие (3) получаем условие глиссирования в форме зависимости порога ротации R от доли заполнения хранилища живыми данными V:

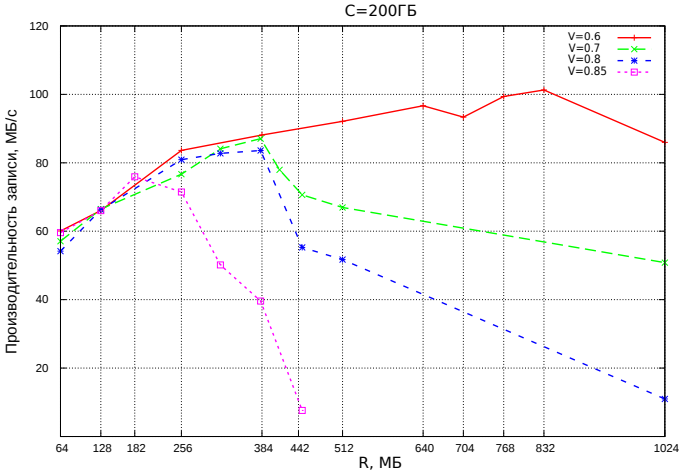


Рис. 10: Зависимость производительности от порога ротации файлов Bitcask R.

$$R \leq \frac{C_i}{n_i}(U_m - V) \tag{4}$$

Отношение (4) носит приближённый характер: в нём не учтены неравномерность распределения данных по vnode, накладные расходы на вспомогательные файлы Bitcask и файловую систему. Тем не менее, оно годится в качестве ориентира при выборе конфигурации системы. При выборе конфигурации следует также учитывать возможность изменения n_i при работе системы в режиме разделения.

Согласно (4), система будет оставаться на глиссировании, если при повышении V пропорционально уменьшать R. Однако при уменьшении R возрастает частота ротации файлов Bitcask, что негативно сказывается на производительности (рис. 10).

Тесты на рис. 8 проводились при $C_i = 100$ГБ, $R = 1$ГБ, $n_i = 64$, $U_m = 0.8$. Это довольно неудачная конфигурация: условие глиссирования (4) выполняется только при $V \leq 0.16$. В действительности тест при $V = 0.2$ ещё демонстрирует максимальную производительность, снижение начинается только при $V = 0.3$. Изучение журнала показывает, что при $V = 0.2$ система ещё находится в режиме глиссирования, а при $V = 0.3$ уже нет. Мы говорили, что условие (4) носит приближённый характер, и на практике наблюдаем небольшую ошибку в меньшую сторону, вероятно вызванную неравномерностью

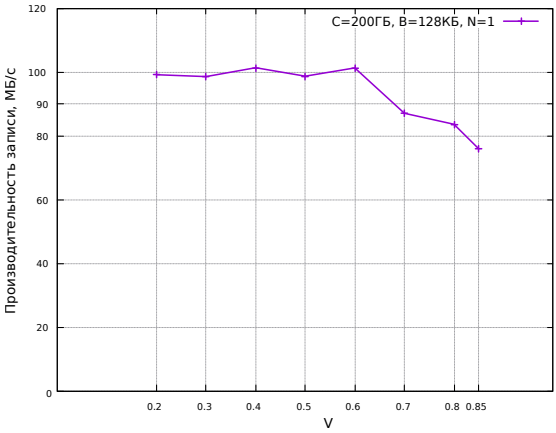


Рис. 11: Зависимость производительности от заполнения хранилища данными V при оптимальных порогах ротации файлов Bitcask R, обеспечивающих глиссирование во всех тестах

распределения данных по vnode. Условие (4) выписано в расчёте на худший случай, когда файлы во всех vnode пополняются равномерно и в момент $U = U_m$ все vnode предлагают кандидатуры на слияние с одинаковым D. Неравномерность приводит к тому, что на некоторых vnode появляются кандидатуры с $D = 1$, хотя среднее D по всем vnode и меньше 1. Наличие хотя бы одной кандидатуры с $D = 1$ в момент $U = U_m$ достаточно для поддержания глиссирования.

4.3. Зависимость производительности от заполнения хранилища V при оптимальном R

Данная серия тестов проведена, чтобы найти баланс между производительностью и эффективностью использования дискового пространства. В отличие от тестов раздела 4.1, в этой серии порог ротации файлов Bitcask R выбирался оптимальным для каждого теста согласно (4), чтобы система всегда работала в режиме глиссирования. Результаты тестов показаны на рис. 11. Снижение производительности при $V > 0.6$ можно объяснить тем, что с ростом V растёт уровень заполнения файловой системы U и вместе с ним растёт фрагментация файлов Bitcask, что, в свою очередь, снижает производительность файловой системы.

При выборе конфигурации системы нужно также учитывать изменения V, связанные с работой в состоянии разделения. Например, в кластере из пяти узлов с $V = 0.6$ при выходе из строя одного узла

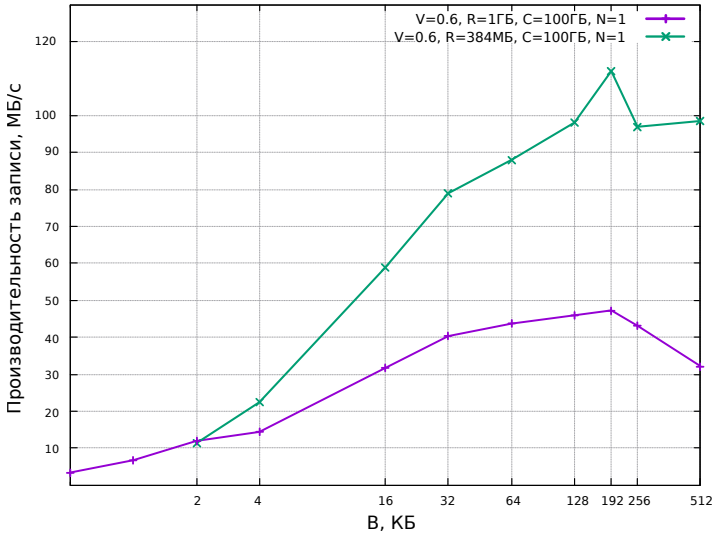


Рис. 12: Зависимость производительности от размера объектов Riak B

относительный объём данных возрастёт до $V = 0.75$, и это усугубит снижение производительности.

4.4. Зависимость производительности от размера объектов Riak

Цель эксперимента — выяснить зависимость производительности записи от размера данных (*value*) в объектах Riak. В документации Riak [**15**] есть рекомендация ограничить размер объекта сверху значением от 100КБ до 2МБ, но не сказано, как влияет на скорость использование небольших размеров, таких как 1 . . . 4КБ.

Поскольку на производительность записи сильно влияет объем данных в хранилище V (раздел 4.1), в данной серии экспериментов мы будем поддерживать V постоянным на уровне 0.6, чтобы изменение V не маскировало интересующую нас зависимость. Для этого при изменении B от эксперимента к эксперименту мы одновременно изменяем S, поддерживая объем живых данных V постоянным. То есть с уменьшением B мы уменьшаем размер объектов в Riak, но увеличиваем их количество. Результаты тестов представлены на рис. 12.

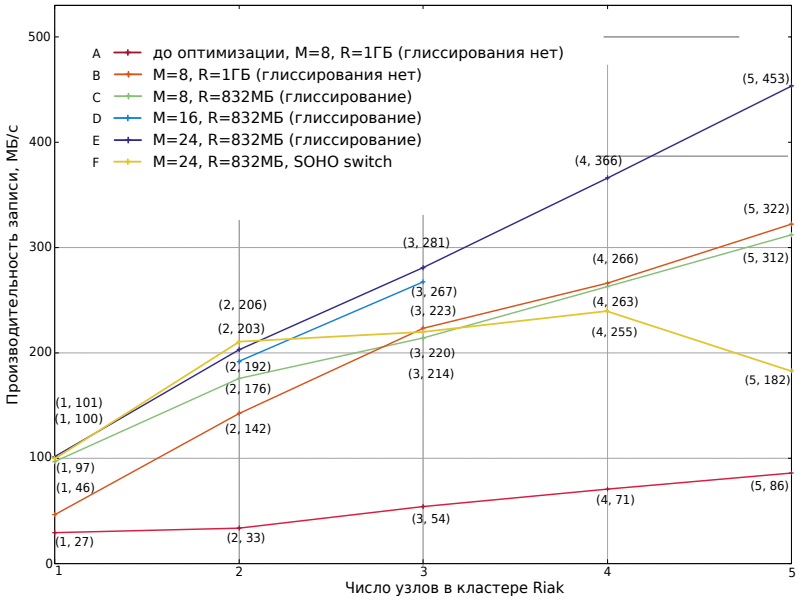


Рис. 13: Зависимость производительности от количества узлов Riak

4.5. Зависимость производительности от количества узлов Riak

Тесты проводились для числа узлов $N \in \{1, 2, 3, 4, 5\}$ при размере объектов Riak B=128K. Результаты представлены на рис. 13.

Линия A получена до оптимизации слияния (раздел 3), поэтому эти результаты сравнительно низкие. Линия B получена с оптимизацией слияния, но при неудачно выбранном пороге ротации файлов Bitcask R, не обеспечивающим глиссирования. Линия C получена с оптимизацией слияния и при R, обеспечивающим глиссирование.

Результаты на линии C при $N > 1$ оказались ниже ожидаемых. Их удалось улучшить за счёт увеличения числа одновременных запросов к Riak (параметр M – число процессов Erlang, генерирующих запросы `put` на каждом узле Riak). Результаты тестов с увеличенным M представлены на линиях D и E; это лучшие достигнутые результаты на сегодняшний день.

Возможное объяснение влияния M на производительность такое. Объекты распределяются по узлам Riak в соответствии со значением хэш-функции от ключа объекта независимо от того, какой узел обрабатывает запрос `put`. С ростом числа узлов доля объектов, которые сохраняются не на том узле, который обрабатывал запрос `put`, растёт как $1 - \frac{1}{N}$. Соответственно растёт среднее время обработки запросов, так что для поддержания скорости приходится увеличивать число параллельно выполняющихся запросов.

Линия F получена при тех же параметрах, что и линия E, но в этом тесте использовался коммутатор Gigabit Ethernet класса SOHO с небольшим объёмом буферной памяти (1 Мбит). Было замечено, что в ходе тестов с $N > 2$ коммутатор передает пакеты «пауза» (IEEE 802.3x Pause Frame), ограничивая трафик через коммутатор. В результате производительность системы значительно упала, вплоть до отрицательного роста при увеличении числа узлов с 4 до 5.[2]

По опыту с линиями C и F можно заключить, что при масштабировании системы всегда желательно тестирование полученного прироста производительности и, если она ниже ожидаемой, требуется поиск и устранение узких мест в системе.

5. Дальнейшие возможности оптимизации слияния

Кроме принятых в разделе 3 мер есть ещё возможности повышения эффективности слияния, пока не реализованные.

5.1. Выборочное чтение

Данная оптимизация относится к работе системы в области B (рис. 7). Даже если система настроена на работу в области A в штатном режиме, временный переход в область B возможен при разделении системы.

[2]Эта проблема известна как «TCP incast»[**16**]. При первых запусках она вызывала аварийное завершение теста до выхода в установившийся режим. Удалось смягчить эффект за счёт отключения режима TSO (TCP segment offload) на сетевом интерфейсе для тестов линии F с 3,4 и 5 узлами. Отключение TSO уменьшает длину серий пакетов, адресованных одному и тому же узлу; такие серии создают повышенную нагрузку на буферную память коммутатора и являются первопричиной проблемы

Как описано выше в разделе 2.1, процесс слияния в Bitcask прочитывает сливаемые файлы полностью, отбрасывает мёртвые данные, а живые данные переписывает в новый файл — результат слияния. Относится ли очередная прочитанная запись к мёртвым или живым данным, выясняется при помощи индекса (*keydir*): если для ключа рассматриваемой записи ссылка в *keydir* указывает не на ту позицию в файле, где обнаружил запись процесс слияния, значит данная запись устарела (есть более свежее значение) и относится к мёртвым.

Чем меньше живых данных в файле, там меньше оправдано прочтение всех записей в файле подряд. Если в файле мало живых данных, эффективнее будет выбрать из него живые данные так, как это делает запрос `get` — руководствуясь индексом, а затем удалить файл, не читая все остальные (мёртвые) записи.

Чтобы реализовать этот подход, процесс слияния должен иметь возможность эффективно получить список всех живых ключей в конкретном файле. Реализация потребует доработки имеющегося индекса *keydir* или введения дополнительного индекса. Окупятся ли эти затраты — открытый вопрос.

5.2. Буферизация записи

Наблюдение за работой Bitcask на уровне системных вызовов ОС Linux показывает, что размер буфера при записи в файлы (как в текущий файл, так и в файлы–результаты слияния) определяется длиной ключа и данных, переданных пользователем Bitcask в запросе `put`. То есть запись производится блоками, не кратными 4КБ и без выравнивания на границу 4КБ. Как мы знаем [5], эффективность записи в этом режиме снижается, хотя при небольшом числе одновременных потоков записи и отсутствии операций чтения это снижение не выражено ярко.

Увеличение размера блока записи прямым увеличением размера буфера в буферном слое B повлечет пропорциональное увеличение объема памяти, необходимой для реализации буферного слоя. Но можно повысить эффективность записи и без увеличения B, если реализовать в Bitcask дополнительную буферизацию данных перед записью в файлы, подобно тому как это делает библиотека stdio языка Си, чтобы запись происходила с оптимальным для используемой

программно-аппаратной платформы размером буфера и с выравниванием на границу блока файловой системы. Дополнительный расход памяти при этом ограничивается одним буфером на каждый открытый для записи файл (текущий файл или результат слияния), которых в Bitcask совсем немного (два файла на каждый vnode).

6. Архитектура системы хранения

По результатам тестов можно заключить, что Riak обеспечивает существенно лучшую производительность записи при большом числе датчиков, чем реализация обобщенной модели на базе файловой системы [5]. Кроме того, Riak обеспечивает возможность дальнейшего наращивания объёма и производительности за счёт увеличения числа узлов распределённой системы. Даже при небольшом числе узлов (два), производительность записи в тестах превышает пропускную способность сети Gigabit Ethernet (120МБ/с). Таким образом, первый вариант тестовой установки с одним фронтальным компьютером (рис. 3) масштабируется плохо, и то же самое можно сказать про системы хранения сенсорных данных с выделенным фронтальным узлом, принимающим данные от датчиков. Итак, чтобы обеспечить горизонтальную масштабируемость, система обязательно должна иметь множественные точки приема сенсорных данных.

Чтобы обеспечить максимальную доступность системы, представляется разумным открыть порт приема сенсорных данных на каждом узле Riak. Приложение, обслуживающее порт приема, будет буферизовать данные в соответствии с обобщённой моделью сохранения и передавать данные в Riak на том же узле. Riak будет передавать данные на один или несколько узлов в соответствии с вычисленным значением хэш-функции Riak и политикой репликации данных. Приложение будет поддерживать протокол обмена с источниками данных, который позволит балансировать нагрузку на порты приёма сенсорных данных. Экземпляры приложения, работающие на узлах, будут поддерживать связь между собой, обнаруживать ситуации разделения системы (нарушения связи между узлами или выход узлов из строя), адаптироваться к работе в ситуации разделения и возвращаться в нормальный режим, когда разделение закончится. Детали реализации такого приложения являются темой отдельной статьи.

7. Заключение

Мы рассмотрели реализацию обобщенной модели сохранения сенсорных данных [5], в которой в качестве вторичной памяти используется распределённая NoSQL система типа ключ-значение Riak KV.

Задача сохранения сенсорных данных оказалась нетипичной нагрузкой для Riak. Для начала потребовалось внести несколько изменений в Riak, чтобы тест не завершался аварийно через полчаса после старта. Затем мы провели анализ производительности системы и выяснили, что производительность страдает из-за неэффективной работы механизма слияния (сборки мусора) в Bitcask. Мы реализовали улучшенный алгоритм управления слиянием в Riak и получили прирост производительности записи в 5 раз для нашего типа нагрузки. Мы протестировали систему при увеличении числа узлов в кластере Riak от 1 до 5 и получили линейный рост производительности записи около 85 МБайт/с на узел. Мы протестировали производительность записи при разных уровнях заполнения системы данными, разных размерах объектов Riak и порогах ротации файлов Bitcask и построили зависимости, полезные для дальнейшей работы.

При использовании результатов тестов для построения систем хранения сенсорных данных с заданными параметрами целесообразно повторить тесты для программно-аппаратной платформы, на которой предполагается строить систему хранения, поскольку количественные результаты от платформы к платформе могут отличаться очень сильно, и «узкие места» часто обнаруживаются там, где их не ждали.

В заключение хочется отметить, что система Riak и язык Erlang оказались благодарными предметами для приложения сил, поблагодарить авторов языка Perl [17] и утилиты redbug [18], которые использовались для изучения работы системы, авторов системы gnuplot [19], которая использовалась при подготовке представленных в статье графиков, а также всех, благодаря кому существует операционная система Debian GNU/Linux [20].

Список литературы

[1] K. Grolinger, W. A. Higashino, A. Tiwari, M. A. M. Capretz. “Data management in cloud environments: NoSQL and NewSQL data stores”,

Journal of Cloud Computing: Advances, Systems and Applications, **2** (2013), 22. ↑ 32

[2] *Riak: a decentralized datastore*, Basho Technologies, URL: https://github.com/basho/riak ↑ 32

[3] E. A. Brewer. "Towards robust distributed systems", Nineteenth annual ACM symposium on Principles of Distributed Computing (Portland, Oregon, United States, July 16–19, 2000), pp. 7, URL: https://people.eecs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf ↑ 32

[4] J. Sheehy, D. Smith. *Bitcask: a log-structured hash table for fast key/value data*, Basho Technologies, 2010-04-27, URL: http://basho.com/wp-content/uploads/2015/05/bitcask-intro.pdf ↑ 32,38

[5] N. S. Zhivchikova, Yu. V. Shevchuk. "Experiments with sensor data storage performance", *Program Systems: Theory and Applications*, **8**:4(35) (2017) (to appear). ↑ 32,33,56,57,58

[6] A. Ghaffari, N. Chechina, P. Trinder, J. Meredith. "Scalable persistent storage for Erlang", *Twelfth ACM SIGPLAN Workshop on Erlang* (Boston, MA, USA, September 25–27, 2013), pp. 73–74, URL: http://eprints.gla.ac.uk/107445/1/107445.pdf ↑ 34

[7] *Basho Bench, a benchmarking tool*, Basho Technologies, URL: https://github.com/basho/basho_bench ↑ 34

[8] Z. Zatrochová. *Analysis and testing of distributed NoSQL datastore Riak*, Master thesis, Masaryk University, Brno, 2015, URL: https://is.muni.cz/th/374482/fi_m/thesis.pdf ↑ 34

[9] P. Nosek. *C client for Riak — NoSQL database*, URL: https://github.com/fenek/riak-c-driver ↑ 34

[10] *Riak Erlang Client*, Basho Technologies, URL: https://github.com/basho/riak-erlang-client ↑ 35

[11] *NIST/SEMATECH e-Handbook of Statistical Methods*, URL: http://www.itl.nist.gov/div898/handbook/pmc/section4/pmc431.htm ↑ 38

[12] Github Basho Bitcask, URL: https://github.com/basho/bitcask/issues/113 ↑ 40

[13] Github Basho Bitcask, URL: https://github.com/basho/bitcask/issues/114 ↑ 40

[14] Basho Technologies, URL: http://docs.basho.com/riak/kv/2.1.4/setup/planning/backend/bitcask/ ↑ 44,45

[15] Basho Technologies, URL: http://docs.basho.com/riak/kv/2.1.4/developing/data-modeling/#sensor-data ↑ 53

[16] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, S. Seshan. "Measurement and analysis of TCP throughput collapse in cluster-based storage systems", *Proceedings of the 6th USENIX Conference on File and Storage Technologies*,

FAST'08 (San Jose, CA, USA, February 26–29, 2008), 14 p., URL: http://www.pdl.cmu.edu/PDL-FTP/Storage/FASTIncast.pdf ↑ 55
[17] *The Perl Programming Language*, URL: https://www.perl.org ↑ 58
[18] *Erlang performance and debugging tools*, URL: https://github.com/massemanet/eper/blob/master/doc/redbug.txt ↑ 58
[19] gnuplot homepage, URL: http://www.gnuplot.info ↑ 58
[20] debian: the universal operating system, URL: http://www.debian.org/ ↑ 58

Рекомендовал к публикации *д.ф.-м.н. С.В. Знаменский*

Пример ссылки на эту публикацию:

Н. С. Живчикова, Ю. В. Шевчук. «Производительность Riak KV в задаче сохранения сенсорных данных», *Программные системы: теория и приложения*, 2017, **8**:3(34), с. 31–60.

URL: http://psta.psiras.ru/read/psta2017_3_31-60.pdf

Об авторах:

Надежда Сергеевна Живчикова

Инженер-исследователь исследовательского центра мультипроцессорных систем Института программных систем им. А. К. Айламазяна Российской академии наук.

e-mail: ming@pereslavl.ru

Юрий Владимирович Шевчук

Зав. лабораторией телекоммуникаций исследовательноско центра мультипроцессорных систем Института программных систем им. А.К.Айламазяна Российской академии наук.

e-mail: sizif@botik.ru

Это же по-английски: DOI 10.25209/2079-3316-2017-8-3-61-85