ИПМ им.М.В.Келдыша РАН

Абрау-2017 • Труды конференции

Н.С. Живчикова, Ю.В. Шевчук

**Масштабируемая распределенная
система архивирования сенсорных
данных**

Рекомендуемая форма библиографической ссылки

Живчикова Н.С., Шевчук Ю.В. Масштабируемая распределенная система архивирования сенсорных данных // Научный сервис в сети Интернет: труды XIX Всероссийской научной конференции (18-23 сентября 2017 г., г. Новороссийск). — М.: ИПМ им. М.В.Келдыша, 2017. — С. 157-169. — URL: <http://keldysh.ru/abrau/2017/35.pdf> doi:[10.20948/abrau-2017-35](https://doi.org/10.20948/abrau-2017-35)

Размещена также [презентация к докладу](#)

Масштабируемая распределенная система архивирования сенсорных данных

Н.С. Живчикова¹, Ю.В. Шевчук

1 ИПС им. А.К. Айламазяна РАН, ИЦМС

Аннотация. В статье описывается проект реализации системы хранения сенсорных данных на базе распределенного хранилища типа ключ-значение Riak KV.

Ключевые слова: сенсорные данные, распределённые системы, RiakKV

1. Введение

Сенсорные данные — поток информации, поступающей небольшими порциями от большого числа датчиков; каждая порция содержит отметку времени и значение, длина которого у разных датчиков может быть разная. Это может быть число, представленное несколькими байтами, может быть текстовая строка, может быть бинарный образ.

В статье рассматривается проект распределенной, горизонтально масштабируемой системы хранения сенсорных данных с использованием распределенного хранилища типа «ключ-значение» Riak [1]. Задача системы хранения — сохранение данных, поступающих от большого числа (порядка 10^6) датчиков так, чтобы было возможно эффективное извлечение массива смежных по времени данных по имени датчика и интервалу дат — для анализа или построения графика.

В качестве примера практического применения системы хранения сенсорных данных приведём систему мониторинга телекоммуникационного оборудования. Цель мониторинга — обеспечение надёжной работы сети, контроль нагрузки, анализ нештатных ситуаций. Сеть небольшого регионального Интернет-провайдера содержит более 1000 сетевых коммутаторов, в каждом от 19 портов, для каждого порта коммутатор поддерживает 30 счётчиков статистики RMON. Перемножая, получаем 570000 датчиков. Данные от каждого датчика поступают с интервалом 60 секунд — 9500 операций добавления в секунду. Ещё драматичнее выглядит ситуация в бурно развивающейся области Интернета вещей, где очень многочисленные, очень дешёвые устройства собираются производить невиданные ранее объёмы данных [2].

Рассматриваемая система ориентирована в первую очередь на первичные данные, поступающие от устройств и сохраняемые без предварительной

обработки, с исходными временными метками, не пересчитанными в канонические сетки. Основной акцент делается на производительность операций записи: цель — сохранить все поступающие данные, даже если большая часть из них никогда не будет востребована. На момент сохранения может быть совершенно непонятно, когда и как эти данные будут использоваться, и любая предварительная обработка может уничтожить важные детали, присутствующие в первичных данных.

Из операций чтения на данный момент рассматривается только одна: извлечение заданной части временного ряда выбранного датчика. Хранение должно быть организовано так, чтобы такие запросы можно было эффективно обрабатывать. Хотя это не исключает реализацию в дальнейшем более сложных запросов с использованием параллельной обработки способом map-reduce; они строятся на базе тех же базовых операций чтения.

Использование распределенной архитектуры в задаче хранения сенсорных данных привлекательно не только с точки зрения повышения производительности и объема системы хранения, но и с точки зрения повышения доступности системы и устойчивости к отказам аппаратуры.

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

- Riak позволяет работать в режиме AP в терминах CAP-теоремы, то есть в условиях нарушения связности кластера или отказа части узлов он позволяет продолжать принимать поступающие данные и, возможно, даже отвечать на запросы — в зависимости от масштаба повреждения и коэффициента репликации, заданного администратором;
- Riak это децентрализованная система, все узлы в ней равноправны, нет узлов со специальными функциями, отказ которых повлечет отказ всей системы;
- для сохранения данных на диске Riak использует журнальное хранилище Bitcask[4], в котором одновременная запись производится в небольшое число файлов, а такой режим обещает хорошую производительность записи[2];
- Riak это система с открытым исходным кодом.

Мы провели ряд экспериментов, чтобы выяснить возможности Riak в задаче хранения сенсорных данных, немного доработали Riak под эту задачу и выяснили, в каких режимах от него можно получить наилучшую производительность. Результаты экспериментов опубликованы в отдельной статье [5], а в настоящей статье мы рассматриваем архитектуру системы хранения сенсорных данных, разработанную с учётом полученных в экспериментах знаний о возможностях Riak.

Отметим, что существует ещё одна реализация системы хранения сенсорных данных на основе Riak: Riak TS [3], разработанная фирмой Basho. В

Riak TS акцент сделан на хранение предобработанных данных и операции с ними, система позиционируется как база данных временных рядов. Можно рассматривать Riak TS и нашу систему не как конкурирующие, а как взаимодополняющие решения.

2. Организация хранения

Прямолинейный подход к сохранению сенсорных данных — сохранять каждую порцию данных в Riak немедленно по поступлении порции в систему. При таком подходе каждой новой порции данных от датчика будет соответствовать отдельный объект в Riak (ключ-значение).

Первая проблема с этим подходом — слишком мелкие объекты, слишком много ключей. В Bitcask все ключи хранятся в хэш-таблице в оперативной памяти. Это обеспечивает эффективность обращения к данным по ключу, но с другой стороны, число ключей в хранилище оказывается ограничено размером доступной оперативной памяти. [7]

В документации Riak есть рекомендации о выборе размера объекта: рекомендуют около 100 КБайт [9]. Наши тесты производительности записи в Riak [5] тоже показывают, что размер сохраняемого объекта сильно влияет на производительность записи (рис. 1). Сохранять объекты маленького размера неэффективно.

Рис.1. Зависимость производительности записи от размера объекта RiakB

Вторая проблема — непонятно, что использовать в качестве ключа объекта. Казалось бы, естественно использовать в качестве ключа временную

метку порции данных, но это не обеспечит эффективного поиска объектов по интервалу ключей невозможна, ключ должен быть задан точно.

Очевидное решение обеих проблем — *timeboxing*: деление оси времени на промежутки фиксированного размера (например, сутки) и запись всех порций данных одного датчика, относящихся к одним суткам, в один объект. Размер объекта таким образом увеличивается, и по интервалу дат легко вычислить ключи всех объектов, в которых находятся удовлетворяющие запросу порции данных. Например, для интервала длиной в неделю потребуется запросить в Riak 7 объектов, за каждые принадлежащие этой неделе сутки.

Однако *timeboxing* хорошо работает только если есть возможность выбрать ширину временного интервала так, чтобы получить оптимальный размер объекта. Если темп поступления порций и размер данных в них переменный, такой возможности нет, и проблема размера объектов Riak остаётся нерешённой.

Поэтому мы выбрали другой подход: исходить из оптимального размера объекта. Данные каждого датчика накапливаются в отдельном буфере в оперативной памяти, пока не накопится достаточно для создания объекта оптимального размера B (рис. 2).

При этом интервал дат порций, попадающих в объект, оказывается произвольным, заранее непредсказуемым. В качестве ключей объектов используются «серийные номера» — ряд натуральных чисел, не связанных напрямую с интервалом дат в объекте. Для каждого датчика система поддерживает в оперативной памяти следующие метаданные:

- серийный номер и интервал дат последнего записанного объекта;
- серийный номер самого старого объекта;
- средняя скорость поступления информации от данного датчика, которая вычисляется как

$$v = ewma(S_i / (D2_i - D1_i))$$

и обновляется при записи каждого очередного объекта. Здесь S_i — размер объекта с серийным номером i , $D1_i$ и $D2_i$ — нижняя и верхняя границы интервала дат, представленных в объекте, функция *ewma* — экспоненциально взвешенное скользящее среднее.

Используя скорость, даты и серийный номер последнего объекта можно точно вычислить серийные номера объектов, относящихся к интервалу дат из запроса, если скорость постоянная (постоянный темп поступления порций и размер данных в них). Если же скорость переменная, полученный по вычисленному ключу объект может оказаться выше или ниже желаемого интервала. Тогда система повторяет вычисление с учетом дат, обнаруженных в полученном объекте, и запрашивает новый объект. Процесс повторяется, пока не будет получен объект, принадлежащий искомому интервалу дат, а все остальные принадлежащие интервалу объекты легко находятся под соседними номерами. Это поиск в отсортированной последовательности данных, похожий

на бинарный, только вместо слепого деления пополам делаются попытки вычислить номер нужного объекта, чтобы уменьшить количество шагов поиска.

Серийный номер самого старого объекта используется для чистки архива, как описано в разделе 7.

Рис.2. Механизм обработки поступающих данных ответственным узлом кластера Riak

3. Лог

Буферизация данных в оперативной памяти имеет один недостаток. Если на принимающем узле случится сбой (перезагрузка, отключение питания), все данные в частично заполненных буферах будут потеряны.

Эту проблему мы решаем при помощи лога. Все поступающие на некоторый узел данные, без разделения по датчикам накапливаются в отдельном буфере. Этот буфер быстро достигает размера B и записывается в Riak (рис. 2). В качестве ключей объектов лога также используются числа натурального ряда с идентификатором узла в качестве префикса. Узел хранит в оперативной памяти и публикует в Riak в специальном разделе «метаданные» (отдельный раздел это «ведро» – bucket в терминологии Riak), номер и даты самого старого объекта лога; номер и даты самого нового объекта лога хранятся только в памяти узла.

При нормальной работе системы лог только записывается и никогда не читается; самые старые объекты лога удаляются, когда все находящиеся в них данные окажутся записаны из буферов в памяти в объекты Riak.

Чтение лога используется в двух случаях: для восстановления несброшенных буферов в памяти при старте системы после сбоя или нормальной остановки, и при переконфигурации системы в ситуации разделения, описанной ниже в разделе 5.

4. Распределение датчиков по узлам

Буферизация данных в оперативной памяти диктует необходимость привязки каждого датчика к узлу, на котором находится буфер с данными данного датчика — узел, ответственный за данный датчик (рис. 3).

Когда клиент-источник данных обращается к системе по DNS-имени, с которым ассоциировано несколько IP-адресов, соединение устанавливается с произвольным узлом. Если клиент пытается передать в систему данные датчика S_i , для которого ответственным узлом является не тот узел, с которым установилось соединение, у системы есть два варианта: или принять данные и незаметно для клиента передать их ответственному узлу, или выдать клиенту указание установить соединение с ответственным узлом напрямую: redirect S_i IPADDR (рис. 4).

Ответственность узла за датчик не означает, что объекты Riak с данными датчика хранятся на этом узле. Объекты распределяются по узлам в соответствии со значением хэш-функции Riak [10], и большей частью хранятся на других узлах.

Рис. 3. Распределение клиентов-источников по ответственным узлам

Рис. 4. Перенаправление запроса клиента-источника ответственному узлу

Также ответственность узла за датчик не означает, что узел обязательно участвует в обработке запросов на чтение данных датчика S_i . Действительно, если в интервал запроса входят данные, которые пока находятся в буфере, запрос должен быть перенаправлен ответственному узлу. Но если запрос относится к более старым данным, которые уже сохранены в виде объектов Riak, любой узел может считать из Riak опубликованные метаданные датчика (раздел 4), провести вычисление ключа как описано выше, считать объекты с данными и ответить клиенту. Это положительное свойство, позволяющее распределить обработку запросов на чтение по кластеру. Также оно обеспечивает обработку запросов на чтение в ситуации разделения.

5. Метаданные датчиков

Информация об ответственности узлов за датчики должна быть доступна всем узлам. Это легко достигается публикацией этих данных в Riak. Для этого в Riak создаётся отдельный раздел, в качестве ключей объектов в этом разделе используются идентификаторы датчиков S_i . Кроме идентификатора ответственного узла, в метаданных датчика присутствуют метаданные, описанные в разделе 1; эта информация нужна для чтения данных любого датчика любым узлом, как описано в разделе 3.

Объекты в разделе метаданных имеют размер, соответствующий объёму метаданных одного датчика. Он существенно меньше оптимального размера B . Чтобы при обновлении метаданных не страдала производительность системы, эти обновления не должны случаться часто. В то же время, строго говоря,

обновление метаданных датчика нужно каждый раз при записи очередного объекта с данными датчика.

В этом вопросе возможен следующий компромисс. При изменении ответственного узла метаданные необходимо записывать немедленно, но при записи очередного объекта данных, когда требуется изменить в метаданных номер и даты последнего объекта, можно пропустить запись. Худшее, что может случиться из-за устаревших метаданных — это что запрос на чтение, который мог бы быть обработан любым узлом, будет направлен ответственному узлу. Получив такой запрос, ответственный узел может записать метаданные, чтобы повторные чтения того же датчика могли обрабатываться без участия ответственного узла. А для датчиков, которые не фигурируют в запросах чтения (а таких большинство), запись метаданных можно откладывать сколь угодно долго без существенных негативных эффектов.

6. Разделение системы

Ситуация разделения возникает при нарушении связи между узлами в кластере или при выходе из строя части узлов. Для узла Riak ситуация разделения всегда выглядит как уменьшение числа узлов в кластере. Покинувшие кластер узлы были ответственны за какие-то датчики, теперь эти ответственные недоступны, но выбор новых ответственных немедленно после обнаружения разделения не имеет смысла.

Если приходит запрос на чтение такого «покинутого» датчика и согласно метаданным датчика запрошенный интервал дат уже находится в объектах Riak, делается попытка прочитать данные. Попытка закончится успехом, если хотя бы один из узлов с репликами нужных объектов находится в строю. Если же таких узлов нет (все остались в другой половине сети), или запрошенный интервал дат недавний и для чтения требуется ответственный, клиенту возвращается ошибка «данные временно недоступны». В любом случае запросы на чтение не приводят к выбору нового ответственного.

Если приходит запрос на запись данных датчика, чей ответственный недоступен, приходится выбирать нового ответственного из числа находящихся в строю узлов (рис. 5). При этом есть возможность прочитать лог старого ответственного, выбрать оттуда данные этого датчика и поместить их во вновь выделенный буфер. Далее новый ответственный работает с датчиком обычным порядком: принимает порции данных, накапливает в буфере, записывает наполненный буфер в объект Riak, обновляет метаданные датчика.

Итак, при разделении смена ответственных происходит только для тех датчиков, чьи источники оказались на той стороне разделения, где нет их ответственных.

Рис. 5. Обработка запроса клиента-источника в ситуации разделения системы

7. Восстановление после разделения

Для узла Riak восстановление выглядит как увеличение числа узлов в системе. Для датчиков, которые за время разделения сменили ответственного, происходит возвращение к старому ответственному. Передача ответственности при восстановлении проходит легче, чем при разделении: между действующим и новым ответственным есть связь, накопленный буфер передаётся новому ответственному в сообщении, необходимости в чтении и фильтрации лога нет. В метаданных датчика на некоторое время сохраняется информация, что за этот датчик недавно был ответственным другой узел и находящиеся в буфере данные могут находиться в его логе, а не в логе текущего ответственного — на случай перезагрузки системы до того, как буфер будет сброшен.

Аналогично обрабатывается передача ответственности при добавлении новых узлов в систему для увеличения её объёма и/или производительности, с той разницей что ответственность не возвращается на старое место, а передаётся от наиболее нагруженного узла вновь добавленному.

8. Удаление старых данных

Система реализует принцип циклического хранилища: поддерживает заполнение хранилища на заданном уровне, удаляя самые старые объекты, чтобы освободить место для вновь поступающих данных.

Отметим, что в Riak есть системный механизм удаления старых данных: `bitcask.expiry` [8]. Он удобен и эффективен, но обеспечивает фиксированное

время хранения, а мы хотим получить максимальное время хранения в пределах доступного дискового пространства в условиях, когда количество датчиков, размер порции данных и темп поступления информации заранее непредсказуемы.

Эффективная реализация принципа циклического хранилища в распределённой системе оказывается не совсем простой. Прежде всего нужно определить, когда пора начинать чистку. Каждый ответственный узел может отслеживать объём хранилища, занятый его датчиками. Обозначим за V_c общий объём данных в системе или кластере,

$$V_c = \sum_{i=1}^N V_i$$

где V_i — суммарный объём объектов, относящихся к датчикам, за которые отвечает узел i , N — число узлов. Для принятия решения о необходимости чистки нужно проверить условие $V_c \leq V$, где V заданное администратором ограничение на общий объём данных в системе.

Мы не можем рассчитывать на равномерное распределение датчиков по ответственным узлам: датчики могут быть распределены неравномерно в результате разделений или добавления узлов в систему, и темп поступления и размер порций у датчиков могут быть очень разные. Поэтому каждый узел должен периодически публиковать своё текущее значение V_i и вычислять V_c с использованием опубликованных другими узлами значений.

Когда решение о чистке принято, перед узлом встаёт вопрос, какие объекты удалять. По метаданным каждый ответственный узел может найти самый старый объект среди своих датчиков и приблизительно определить его верхнюю дату $D2_i$, но этого недостаточно. Нужна координация между узлами, чтобы все узлы удаляли объекты, относящиеся к одним и тем же датам, никто значительно не вырывался вперёд. Для этого предлагается следующий механизм.

Вместе с V_i все узлы публикуют свои предложения о возрастном пороге удаления E_i и вычисляют действующий порог

$$E = M + \mu(t - M)$$

где

$$M = \min(E_1, \dots, E_N),$$

t — текущая отметка времени,

$\mu = 0.01$ — коэффициент упреждающего удаления.

Каждый узел вычисляет свой целевой уровень заполнения

$$V_g = V_i - (V_c - V) \frac{V_i}{V_c}$$

и двигается по списку метаданных датчиков, удаляя все объекты с $D_2 < E$ и запоминая $D2$ самого старого неудалённого объекта в качестве новой кандидатуры E_i . Движение приостанавливается, когда V_i , уменьшающийся в

результате чистки и увеличивающийся при сохранении новых данных в хранилище, достигнет целевого уровня: $V_i \leq V_g$.

Текущее состояние V_i и E_i публикуется после завершения прохода по списку метаданных, затем делается задержка, чтобы все узлы успели опубликовать свои данные, и начинается следующая итерация процесса удаления.

Коэффициент упреждающего удаления μ повышает результативность прохода по метаданным. При $\mu = 0$ в результате прохода будут удалены только объекты с $D_2 = E_i$, и с большой вероятностью такой объект найдётся только один. При $\mu = 0.01$ в результате прохода будут удалены все объекты, занимающие нижний 1% в распределении объектов по оси времени.

9. Заключение

На основании [5] мы надеемся, что система, реализованная на описанных принципах, будет иметь высокую производительность записи и масштабируемость. Что касается характеристик в режиме чтения, поддержания заполнения хранилища на заданном уровне V , поведения системы при разделении и восстановлении после разделения, их предстоит проверить экспериментально после реализации системы.

Литература

1. Basho Technologies. — URL: <http://basho.com/products/#riak>.
2. Pavel Hardak. Using Spark and Riak for IoT Apps—Patterns and Anti-Patterns/ Basho Technologies, 2017. — URL: <https://www.slideshare.net/SparkSummit/usingspark-and-riak-for-iot-appspatterns-and-antipatterns-spark-summiteast-talk-pavel-hardak>.
3. Riak TS, Basho Technologies. — URL: <http://basho.com/products/riak-ts/>.
4. Zhivchikova N.S., Shevchuk Yu.V. Experiments with sensor data storage performance // Program Systems: Theory and Applications. — Online Scientific Journal, PSI RAS, 2017. — URL: <http://psta.psir.ru>, (in print).
5. Zhivchikova N.S., Shevchuk Yu.V. Riak performance in sensor data storage application // Program Systems: Theory and Applications. — Online Scientific Journal, PSI RAS, 2017. — URL: http://psta.psir.ru/read/psta2017_3_61-85.pdf.
6. Bitcask: A Log-Structured Hash Table for Fast Key/Value Data // Basho Technologies. — URL: <http://basho.com/wp-content/uploads/2015/05/bitcask-intro.pdf>.
7. Justin Sheehy, David Smith Riak KV // Basho Technologies. — URL: <http://docs.basho.com/riak/kv/2.2.3/setup/planning/backend/bitcask/>.
8. Justin Sheehy, David Smith. Configurable parameters for Riak's Bitcask storage backend/ Basho Technologies. — URL: <http://docs.basho.com/riak/kv/2.2.3/configuring/backend/#bitcask/>.

9. Riak KV. FAQ // Basho Technologies. — URL: <https://docs.basho.com/riak/kv/2.2.3/developing/faq/> .
10. Consistent Hashing// Basho Technologies. — URL: <http://docs.basho.com/riak/1.4.8/theory/why-riak/>
11. 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.
12. E. A. Brewer. “Towards robust distributed systems”, Nineteenth annual ACM symposium on Principles of Distributed Computing (Portland, Oregon, United States, July 16–19, 2000), URL: <https://people.eecs.berkeley.edu/~brewer/cs262b-2004/PODCkeynote.pdf>