Arenadata™ Streaming

Версия - v1.4-RUS

Архитектурные особенности

Оглавление

В документации приведены архитектурные особенности платформы Arenadata Streaming.

Инструкция может быть полезна администраторам, программистам, разработчикам и сотрудникам подразделений информационных технологий, осуществляющих внедрение и сопровождение системы.

Important: Контактная информация службы поддержки – e-mail: info@arenadata.io

Глава 1

Обзор

Разработанная компанией **Arenadata** платформа **ADS** имеет возможность выступления в качестве единой платформы для обработки всех потоков данных в реальном времени, которые может иметь крупная компания. С этой целью продуман достаточно широкий набор вариантов использования:

- **ADS** должна иметь высокую пропускную способность для поддержки потоков событий большого объема, таких как агрегация журнала в режиме реального времени.
- **ADS** должна грамотно справляться с большими объемами данных, чтобы иметь возможность поддержки периодических загрузок данных из автономных систем.
- **ADS** должна обрабатывать передачу данных с низкой задержкой с целью обработки большого количества традиционных случаев использования сообщений.

Партицирование и потребительская модель были мотивированы желанием **Arenadata** поддерживать секционированную распределенную обработку в реальном времени для создания новых унаследованных лент.

Наконец, в тех случаях, когда поток подается в сторонние системы данных для обслуживания, система должна гарантировать отказоустойчивость при наличии сбоев в работе машины.

Поддержка перечисленных критериев использования привела **Arenadata** к разработке платформы **ADS** с несколькими уникальными элементами, более похожими на журнал базы данных, чем на традиционную систему обмена сообщениями.

Глава 2

Персистентность

ADS в значительной степени опирается на файловую систему для хранения и кэширования сообщений. Существует общее представление о том, что “диски медленные”, и это заставляет скептически относиться к тому, что персистентная структура может предложить конкурентоспособную производительность. На самом деле диски могут быть и намного медленнее, и намного быстрее в зависимости от того, как они используются. И правильно разработанная структура диска часто может быть такой же быстрой, как и сеть.

Ключевым фактом о производительности диска является то, что пропускная способность жестких дисков отличается от латентности диска в течение последнего десятилетия. В результате производительность линейных записей в конфигурации **JBOD** с шестью массивами **SATA RAID-5** размером *7200 об/мин* составляет около *600 МБ/с*, но производительность операции случайной записи составляет всего около *100 к/сек* – разница более *6000X*. Эти линейные операции чтения и записи являются наиболее предсказуемыми для всех моделей использования и сильно оптимизированы операционной системой. Современная ОС предоставляет технологии опережающего чтения и записи, которые обеспечивают предварительную выборку данных в больших кратных блоках и группируют меньшие логические записи в большие физические записи. Дальнейшее обсуждение этой проблемы можно найти в статье ACM Queue, где показывается, что последовательный доступ к диску может в некоторых случаях быть быстрее, чем случайный доступ к памяти.

Для компенсации расхождения в производительности современные операционные системы становятся все более агрессивными в использовании основной памяти для кэширования диска. Современная ОС может перенаправить всю свободную память на кэширование диска с небольшим снижением производительности при восстановлении памяти. Все операции чтения и записи на диск будут проходить через этот единый кэш. Но данная функция не может быть легко отключена без использования прямого ввода-вывода, поэтому, даже если процесс поддерживает встроенный кэш данных, эти данные, вероятно, будут дублироваться в pagecache операционной системы, сохраняя все дважды.

Платформа **ADS** строится поверх **JVM**, и всем, кто сталкивался с использованием памяти **Java**, известно:

- Накладные расходы памяти на объектах очень высоки, что часто удваивает размер хранимых данных;
- Сбор мусора Java становится все более неудобным и медленным по мере увеличения объема данных in-heap.

В результате этих факторов, используя файловую систему и полагаясь на pagecache, лучше поддерживать кэш in-memory или другую структуру – при наличии свободной памяти появляется возможность использовать ее в качестве второго кэша. Это приводит к кэшу до *28-30 ГБ* на машине *32 Гб* без ограничений сборщиков мусора. Кроме того, кэш остается теплым, даже при перезагрузке сервиса. В то время как кэш in-process необходимо перестраивать в памяти (что для кэша *10 ГБ* может занять до *10 минут*), иначе ему придется стартовать с полностью холодного кэша (что приводит к снижению производительности). К тому же значительно упрощается код, поскольку вся логика поддержания согласованности между кэшем и файловой системой теперь находится в ОС, которая имеет тенденцию делать это более эффективно и правильно, чем однократные попытки in-process. Если использование диска способствует линейному чтению, то опережающее чтение эффективно заполняет этот кэш полезными данными на каждом диске.

В **ADS** предлагается другая простая структура: вместо того, чтобы содержать все данные in-memory, и когда заканчивается пространство полностью очищать файловую систему, инвертировать данный процесс и записывать все данные сразу же в постоянный журнал файловой системы без необходимости сброса на диск. По сути это просто означает, что данные переносятся в pagecache ядра.

Такой ориентированный на pagecache стиль описывается в статье Notes from the Architect.

Персистентная структура данных, используемая в системах обмена сообщениями, часто представляет собой очередь для каждого потребителя со связанным BTree или другими структурами данных произвольного доступа общего назначения для поддержки метаданных о сообщениях. BTrees являются наиболее универсальной структурой данных и позволяют поддерживать широкий спектр транзакционной и нетранзакционной семантики в системе обмена сообщениями. Они имеют довольно высокую стоимость, однако: Btree операции – $O\ (log\ N)$. Обычно $O\ (log\ N)$ считается эквивалентным постоянному времени, но это не относится к дисковым операциям. Диск выполняет поиск за *10 мс*, и каждый диск может делать только один поиск за раз, поэтому параллелизм ограничен. Следовательно, даже небольшое количество запросов на диск приводит к очень высоким накладным расходам. Поскольку системы хранения данных сочетают очень быстрые кэшированные операции с очень медленными физическими дисковыми операциями, наблюдаемая производительность древовидных структур часто суперлинейна по мере увеличения данных с фиксированным кэшем, то есть дублирование данных – это намного хуже по сравнению с вдвое медленной скоростью.

Интуитивно персистентная очередь может быть построена на простых операциях чтения и добавления к файлам, как это бывает с решениями по ведению журнала. Такая структура имеет преимущество в том, что все операции – $O\ (1)$, и операции чтения не блокируют операции записи или друг друга. Это имеет очевидные преимущества в производительности, так как она полностью отделена от размера данных. Один сервер теперь может в полной мере использовать ряд дешевых и низкоскоростных дисков *1+TB SATA*. Хотя у них низкая производительность поиска, она приемлема для многочисленных операций чтения и записи и достигает *1/3* цены и *3x* емкости.

Имея доступ к практически неограниченному дисковому пространству без какого-либо снижения производительности **ADS** может предоставить некоторые функции, которые обычно не встречаются в системах обмена сообщениями. Например, в **ADS** вместо того, чтобы удалять сообщения сразу после их считывания, можно сохранять их в течение относительно длительного периода, что приводит к большой гибкости для потребителей.

Глава 3

Производительность

Специалисты **Arenadata** приложили значительные усилия для повышения производительности платформы. Одним из основных вариантов использования является обработка данных веб-активности, которая очень велика: каждый просмотр страницы может генерировать десятки записей. Кроме того, каждое опубликованное сообщение читается как минимум одним потребителем (а часто – многими), поэтому есть стремление сделать потребление как можно более дешевым.

По опыту разработки и эксплуатации ряда аналогичных систем выяснилось, что производительность является ключом к эффективным многопользовательским операциям. Если нижестоящий сервис инфраструктуры может легко стать узким местом из-за незначительного увеличения по использованию приложения, то такие небольшие изменения часто создают проблемы. В результате приложение выйдет из строя под нагрузкой перед инфраструктурой. Это особенно важно при попытке запуска централизованного сервиса, поддерживающего десятки или сотни приложений в централизованном кластере, поскольку изменения в шаблонах использования происходят почти ежедневно.

После устранения неудачных шаблонов доступа к диску остается две общие причины неэффективности в подобном типе системы: слишком много мелких операций ввода-вывода и избыточное копирование байтов.

Проблема множества мелких операций ввода-вывода происходит как между клиентом и сервером, так и в собственных персистентных операциях сервера. Во избежание этого протокол **Arenadata** построен вокруг абстракции “набор сообщений”, группирующей сообщения. Это позволяет сетевым запросам объединять сообщения вместе, амортизировав накладные расходы в сети, а не отправлять каждый раз по одному сообщению. Сервер единоразово добавляет фрагменты сообщений в свой журнал, а потребитель извлекает большие линейные фрагменты за раз.

Такая простая оптимизация на порядок увеличивает скорость работы. Пакетирование приводит к увеличению сетевых пакетов, последовательных операций с дисками и смежными блоками памяти и т.д., что позволяет платформе **ADS** превращать поток случайных сообщений в линейные записи, которые поступают потребителям.

Другая непродуктивность заключается в копировании байтов. При низких скоростях передачи сообщений это не проблема, но под нагрузкой влияние значительно. Во избежание этого **ADS** использует стандартный бинарный формат сообщений, который совместно применяется поставщиком, брокером и потребителем (таким образом, блоки данных могут передаваться без изменений).

Журнал сообщений, поддерживаемый брокером, сам по себе является просто каталогом файлов, каждый из которых заполняется рядом наборов сообщений, которые были записаны на диск в том же формате, который используется поставщиком и потребителем. Поддержка общего формата позволяет оптимизировать наиболее важную операцию – сетевую передачу персистентных блоков журнала. Современные операционные системы **unix** предлагают высоко оптимизированный путь кода для передачи данных из pagecache в сокет; в **Linux** это делается с помощью системного вызова *sendfile*.

Путь передачи данных из файла в сокет заключается в следующем:

1. Операционная система считывает данные с диска в pagecache в пространстве ядра.
2. Приложение считывает данные из пространства ядра в буфер пространства пользователя.
3. Приложение записывает данные в пространство ядра в буфер сокета.
4. Операционная система копирует данные из буфера сокета в буфер сетевого адаптера по сети.

Такой метод явно неэффективен – он предполагает четыре операции копирования и два системных вызова. А при использовании *sendfile* повторное копирование исключается (zero-copy), позволяя ОС напрямую отправлять данные из pagecache в сеть. Таким образом, из оптимизированного пути требуется только последняя копия в буфер сетевого адаптера.

Как правило, общий вариант использования состоит из нескольких потребителей топика. При приведенной выше оптимизации данные копируются в pagecache только один раз и при каждом потреблении используются повторно, а не хранятся в памяти и копируются в пространство пользователя при каждом чтении. Это позволяет считывать сообщения со скоростью, приближенной к пределу сетевого подключения.

Такая комбинация использования pagecache и *sendfile* означает, что в кластере **ADS** нет активности чтения на дисках, поскольку данные полностью обслуживаются из кэша.

Дополнительные сведения о поддержке *sendfile* и zero-copy в **Java** приведены в статье.

Глава 4

Поставщики данных

4.1 Балансировка нагрузки

Поставщик отправляет данные непосредственно брокеру, являющемуся лидером партиции, без какого-либо промежуточного уровня маршрутизации. При этом все узлы **ADS** могут ответить на запрос метаданных, какие серверы активны, и где лидеры партиций топика находятся в любой момент времени, чтобы позволить поставщику соответствующим образом направлять свои заявки.

Клиент контролирует, в какую партицию публикуются сообщения. Публикация данных может осуществляться произвольным образом, реализуя некую случайную балансировку нагрузки, либо с помощью некоторой функции семантического разбиения, предоставленной интерфейсом **ADS**, которая позволяет пользователю указывать ключ для секционирования, и использует этот хэш в партиции (при необходимости функцию можно переопределить). Например, если выбранный ключ является id пользователя, то все данные по конкретному пользователю публикуются в данную партицию. Потребителям, в свою очередь, это позволяет делать выводы относительно их потребления. Данный стиль партицирования явно разработан таким образом, чтобы разрешить локально-чувствительную обработку.

4.2 Асинхронная передача

Пакетирование является одним из главных факторов эффективности, и для ее обеспечения поставщик накапливает данные в памяти и затем отправляет их большими партиями в одном запросе. Пакетная обработка может быть сконфигурирована таким образом, чтобы накапливание не превышало фиксированного количества сообщений, и/или время ожидания было не больше указанного (например, *64k* или *10 мс*). Это позволяет накапливать больше байтов для передачи и большее количество операций ввода-вывода на серверах. Буферизация настраиваема и дает возможность обмена небольшой дополнительной задержки по времени на лучшую пропускную способность.

Глава 5

Потребители данных

Потребитель платформы **ADS** передает выборки запросов брокерам-лидерам необходимых ему партиций и задает свое смещение в журнале. Таким образом, потребитель имеет контроль над своей позицией в журнале и при необходимости может повторно считывать данные.

Изначально вопрос заключался в том, должны ли потребители получать данные от брокеров или брокеры должны передавать данные потребителю. В этом отношении **ADS** следует более традиционному способу, разделяемому большинством систем обмена сообщениями, где данные отправляются в брокер поставщиком и получаются из брокера потребителем. Некоторые системы, ориентированные на ведение журнала, такие как **Scribe** и **Apache Flume**, следуют совершенно другому принципу – модели “push”. У обоих подходов есть плюсы и минусы. Система push сталкивается с трудностями при работе с несколькими потребителями, поскольку брокер контролирует скорость передачи данных. А цель, как правило, заключается в том, чтобы потребитель считывал данные с максимально возможной скоростью; к сожалению, в системе push потребитель имеет тенденцию к перегрузке, когда его уровень потребления падает ниже уровня поставки данных. Система “pull” имеет лучшие свойства, при которых потребитель просто “отстает и догоняет”. При этом можно использовать протокол, в котором потребитель указывает свою перегрузку, но получение повышенной скорости передачи данных для потребителя сложнее, чем кажется. Предыдущие попытки построения систем привели **ADS** к использованию традиционной модели pull.

Другим преимуществом системы pull является то, что она поддается агрессивной пакетной обработке данных, отправляемых потребителю. В то время как система push должна либо немедленно отправить запрос, либо накопить больше данных, а затем отправить их, не зная, сможет ли нижестоящий потребитель немедленно их обработать. А при настроенной низкой задержке оправка сообщений осуществляется по одному за раз и в любом случае буферизуется, что является расточительным. В модели pull потребитель всегда получает все доступные ему данные в зависимости от его текущего положения в журнале (или до установленного максимального размера). Таким образом, можно получить оптимальное дозирование без лишней задержки.

Недостаток системы pull заключается в том, что если у брокера нет данных, потребитель может выполнять запрос в жестком цикле, фактически находясь в режиме ожидания поступления данных. Во избежание этого в **ADS** есть параметры запроса pull, которые позволяют заявке потребителя блокировать “длинный опрос” (“long poll”) до поступления данных (и при необходимости до тех пор, пока не будет достигнуто заданное количество байтов).

5.1 Позиция потребителя

Отслеживание считываемых данных, на удивление, является одной из ключевых точек производительности системы обмена сообщениями.

Большинство систем обмена сообщениями содержат метаданные о том, какие сообщения были считаны в брокер. То есть, при передаче сообщения потребителю брокер либо немедленно записывает его локально, либо

ожидает подтверждения от потребителя. Это довольно интуитивный выбор, и действительно, серверу неясно, куда еще может потребоваться заявка. Поскольку структуры данных, используемые для хранения во многих системах обмена сообщениями, плохо масштабируются, это прагматичный выбор – поскольку брокер знает, какие данные считаны, и он может тут же удалить их, сохраняя размер данных небольшим.

Что, возможно, не очевидно, так это то, что заставить брокера и потребителя прийти к соглашению о считанных данных не является тривиальной задачей. Если брокер каждый раз немедленно записывает сообщение как считанное при его передаче по сети, то если потребитель не обрабатывает сообщение (например, по причине сбоя или времени ожидания запроса), сообщение теряется. Для решения этой проблемы многие системы обмена сообщениями добавляют функцию подтверждения, которая помечает сообщения как отправленные, но не считанные, пока брокер не дождется подтверждения от потребителя, чтобы записать сообщение как считанное. Такая стратегия устраняет проблему потери данных, но создает новые проблемы. Прежде всего, если потребитель обрабатывает сообщение, но не может отправить подтверждение, сообщение считывается дважды. Вторая проблема связана с производительностью – теперь брокер должен хранить несколько состояний о каждом сообщении (сначала заблокировать, чтобы не было повторного считывания, затем отметить как явно считанное, чтобы можно было удалить). Подобные замысловатые проблемы необходимо решать, как, например, что делать с сообщениями, которые отправляются, но никогда не подтверждаются.

ADS справляется с этим по-другому. Топик разделяется на набор полностью упорядоченных партиций, каждая из которых потребляется в любой момент времени ровно одним потребителем из подписавшейся группы потребителей. Это означает, что позиция потребителя в каждой партиции, то есть смещение от следующего сообщения – это всего лишь целое число. Это говорит о том, что состояние о считывании данных очень малоемкое – всего лишь одно число для каждой партиции. Текущее состояние можно проверять. Все это делает подобный эквивалент функции подтверждения о считывании сообщений очень дешевым.

К тому же данное решение имеет побочное преимущество – потребитель имеет возможность намеренно вернуть назад свое смещение и повторно считать данные. Это нарушает общий порядок очереди, но при этом является существенным плюсом для многих потребителей.

5.2 Автономная загрузка данных

Масштабируемая персистентность позволяет потребителям, которые только периодически считывают пакетные данные, время от времени загружать данные в автономную систему, такую как, например, **Hadoop** или реляционное хранилище данных.

В случае **Hadoop** нагрузка данных распараллеливается на отдельные задачи, по одной для каждой комбинации узел/топик/партиция. **Hadoop** обеспечивает управление задачами, а задачи, потерпевшие неудачу, могут перезапускаться без риска дублирования данных – они просто возобновляются с исходной позиции.

5.3 Отслеживание смещений потребителя

Потребитель высокого уровня отслеживает свое максимальное смещение при считывании данных в каждой партиции и периодически фиксирует вектор смещения для возможности возобновления работы с необходимых позиций. **ADS** предоставляет возможность хранения всех смещений группы потребителей в определенном брокере (для каждой группы) – в менеджере смещений, то есть любой инстанс потребителя группы должен передавать свои смещения и запрашивать выборки через менеджер смещения (брокер). У потребителей высокого уровня эта операция выполняется автоматически, в то время как простой потребитель управляет своими смещениями вручную, так как в настоящее время автоматическая передача смещений простых потребителей не поддерживается **Java**, им доступно только ручное фиксирование или извлечение смещений в **ZooKeeper**. При использовании **Scala** простой потребитель может явно зафиксировать или получить смещения в менеджере.

Поиск менеджера смещения осуществляется по запросу *GroupCoordinatorRequest* в любой брокер **ADS** с последующим ответом *GroupCoordinatorResponse*, в котором содержится менеджер смещения. После чего

потребитель может перейти к фиксации или извлечению смещений. В случае перемещения менеджера смещений потребителю необходимо повторно выполнить его поиск.

При получении менеджером смещения запроса на фиксацию *OffsetCommitRequest*, он добавляет смещение в специальный сжатый топик **ADS** с именем *__consumer_offsets*. Менеджер смещения отправляет успешный ответ фиксации смещения потребителю только после получения смещений всеми репликами топика. В случае если смещения не могут реплицироваться в пределах настренного времени ожидания, фиксация смещения завершается неудачей, и потребитель может повторить ее после отмены действия (у потребителей высокого уровня процедура выполняется автоматически). Брокеры периодически сжимают топик смещения, так как ему требуется поддерживать только последнее смещение на партицию. Менеджер смещения также упорядоченно кэширует все смещения в таблице in-memory для возможности их быстрой обработки.

Когда менеджер смещения получает запрос на извлечение смещения, он просто возвращает последний зафиксированный вектор смещения из кэша. В случае если менеджер был только что запущен или стал менеджером смещения для нового набора групп потребителей (став лидером для партиции топика смещения), может потребоваться загрузка партиции топика смещений в кэш. В это время операции по извлечению смещения завершаются ошибкой *OffsetsLoadInProgress*, и потребитель может повторить запрос *OffsetFetchRequest* после окончания копирования (у потребителей высокого уровня процедура выполняется автоматически).

5.4 Миграция смещений из ZooKeeper в ADS

Для миграции потребителей и смещений из **ZooKeeper** в **ADS** необходимо выполнить следующие действия:

1. Установить *offsets.storage=ads* и *dual.commit.enabled=true* в настройках потребителя.
2. Выполнить резкий переход к потребителям и убедиться в их исправности.
3. Установить *dual.commit.enabled=false* в настройках потребителя.
4. Выполнить резкий переход к потребителям и убедиться в их исправности.

Обратный переход (с **ADS** в **ZooKeeper**) также может выполняться с помощью вышеуказанных шагов при установке *offsets.storage=zookeeper*.

Глава 6

Механизм доставки сообщений

После введения в принципы работы поставщиков и потребителей данных следует изучить семантический анализ гарантий между ними, которые обеспечивает **ADS**. Существует несколько возможных гарантий по доставке сообщений:

- *At most once* – сообщения могут быть потеряны, но никогда не будут повторно отправлены.
- *At least once* – сообщения никогда не теряются, но могут быть повторно отправлены.
- *Exactly once* – наиболее востребованное – каждое сообщение доставляется один и только один раз.

Все сводится к двум проблемам: гарантии долговечности при публикации данных и гарантии при их считывании.

Многие системы утверждают, что обеспечивают гарантию “Exactly once”, но при этом большинство подобных утверждений ошибочно (т.к. ими не предусмотрены случаи, когда потребители или поставщики могут потерпеть неудачу; при многопотребительских процессах; когда данные, записанные на диск, могут быть потеряны).

Семантика **ADS** прямолинейна. При публикации сообщения, оно фиксируется в журнале. Как только опубликованное сообщение зафиксировано, оно не может быть потеряно, пока брокер, реплицирующий партицию, на которую было написано это сообщение, остается “живым”. Определения зафиксированного сообщения, живой партиции, а также описание типов сбоев приведено подробно в следующем разделе. Сейчас рассмотрим идеальный брокер без потерь и попытаемся понять гарантии поставщика и потребителя. Если поставщик пытается опубликовать сообщение и возникает сетевая ошибка, он не может быть уверен в том, когда произошла ошибка – до или после фиксации сообщения. Аналогично семантике операции вставки в таблицу базы данных с автогенерацией ключа.

Поставщик **ADS** поддерживает опцию идемпотентной доставки, гарантирующую, что повторная отправка не приводит к дублированию записей в журнале. Для достижения этого брокер назначает каждому поставщику идентификатор и дедуплицирует данные, используя порядковый номер, который отправляется поставщиком вместе с каждым сообщением. Так же поставщику предоставляется возможность отправки сообщений в несколько партиций топика, используя подобную транзакционной семантику: то есть либо успешно записаны все сообщения, либо не записано ни одно из них. Основным вариантом использования этого метода является однократная обработка между топиками **ADS** (описание приведено ниже).

Не все варианты использования требуют таких строгих гарантий. Для чувствительных к задержкам случаев у поставщика есть возможность определить желаемый уровень устойчивости (время ожидания фиксации сообщения может занять порядка *10 мс*). Однако поставщик может также указать, что он хочет выполнить запись данных полностью асинхронно, или что он хочет ждать только до тех пор, пока лидер (но не обязательно последователи) получит сообщение.

Теперь рассмотрим семантику с точки зрения потребителя. Все реплики имеют один и тот же журнал

с одинаковыми смещениями. Потребитель контролирует свое положение в этом журнале. Если у потребителя не было сбоев, то он просто хранит эту позицию в памяти. Но в случае если потребитель терпит неудачу и необходимо, чтобы партиция этого топика была перехвачена другим процессом, новому процессу нужно выбрать соответствующую позицию, с которой следует продолжить обработку. К примеру, потребитель считывает данные – он имеет два варианта обработки данных и обновления своей позиции:

1. Потребитель может считывать сообщения, затем сохранять свою позицию в журнале и потом выполнять обработку. В этом случае существует вероятность того, что процесс выйдет из строя после сохранения позиции потребителя и до сохранения результатов обработки данных. Тогда другой процесс, перехвативший на себя обработку, начинается с сохраненной позиции, даже если какие-то сообщения до этой позиции не были обработаны. Это соответствует семантике “At most once”, так как в случае сбоя потребителя данные могут быть не обработаны.

2. Потребитель может считывать сообщения, затем обрабатывать их и потом сохранять свою позицию. В этом случае существует вероятность того, что процесс выйдет из строя после обработки данных, но до сохранения позиции потребителя. Тогда другой процесс, перехвативший на себя обработку, обрабатывает уже обработанные данные. В случае сбоя потребителя это соответствует семантике “At least once”. При этом во многих случаях сообщения имеют первичный ключ, поэтому обновления являются идемпотентными (повторное получение одного и того же сообщения просто перезаписывает его другой копией самого себя).

Что по поводу семантики “Exactly once” (то, что действительно желаемо)? При потреблении данных из топика **ADS** и их записи в другой топик, можно использовать транзакционную семантику поставщика, которая была упомянута выше. Позиция потребителя сохраняется как сообщение в топике, поэтому можно записать смещение в **ADS** в той же транзакции, что и топики смещения, получающие обработанные данные. Если транзакция прерывается, позиция потребителя возвращается к своему старому значению, и записанные в топик смещения данные доступны другим потребителям в зависимости от их “уровня изоляции”. На установленном по умолчанию уровне изоляции “read_uncommitted” все сообщения видны для потребителей, даже если они были частью прерванной транзакции, а на уровне “read_committed” потребитель возвращает только зафиксированные транзакцией сообщения (и любые данные, которые не являлись частью транзакции).

При записи данных во внешнюю систему существует оговорка, которая заключается в необходимости координировать позицию потребителя с тем, что фактически хранится в качестве выходных данных. Классический способ достижения этого – ведение двухфазной фиксации между хранением потребительской позиции и хранением потребительских данных. Но также этого можно добиться и проще – позволяя потребителю хранить смещение в том же месте, что и выходные данные. Например, коннектор **ADS Connect** записывает данные в **HDFS** вместе со смещениями считанных данных для гарантии их обновлений. Аналогичный шаблон используется для многих других систем данных, требующих более сильную семантику, и у которых сообщения не имеют первичного ключа для дедупликации. Такой метод лучше, потому что многие выходные системы, в которые потребитель может записывать данные, не поддерживают двухфазную фиксацию.

Так что **ADS** эффективно поддерживает гарантию “Exactly once”, и транзакции поставщик/потребитель могут использоваться для ее обеспечения при передаче и обработке данных между топиками. “Exactly once” гарантия в других системах обычно требует взаимодействия с такими же системами, а в **ADS** обеспечивается смещение, которое это реализует. В иных случаях **ADS** по умолчанию гарантирует как минимум доставку “At least once” и дает возможность пользователю реализовывать доставку по принципу “At most once” путем отключения повторных попыток записи у поставщика и фиксации смещений у потребителя перед обработкой пакета данных.

Глава 7

Репликация

Платформа **ADS** реплицирует журнал для каждой партиции топика через настраиваемое число серверов (коэффициент репликации можно задать по принципу “topic-by-topic”). Это реализует постоянный доступ к данным, выполняя автоматический переход к репликам при сбое сервера в кластере.

Другие системы обмена сообщениями предоставляют некоторые связанные с репликацией функции, но, по нашему (полностью предвзятому) мнению, они не сильно востребованы и при этом имеют большие минусы: неактивность подчиненных процессов, сильное отрицательное влияние на производительность, ручная настройка fiddly и т.д. Платформа **ADS** подразумевает репликацию по умолчанию – фактически нереплицируемые топики реализуются как реплицированные с коэффициентом репликации – один.

Единицей репликации является партиция топика. В условиях отсутствия сбоев каждая партиция в **ADS** имеет одного лидера и ноль или более последователей. Коэффициент репликации составляет общее количество реплик, включая лидера, которому направляются все операции чтения и записи. Как правило, существует гораздо больше партиций, чем брокеров, и лидеры равномерно распределяются между брокерами. Журналы подписчиков идентичны журналу лидера – все имеют одинаковые смещения и данные, расположенные в одном и том же порядке (хотя, конечно, в любой момент времени лидер может иметь несколько пока еще нереплицированных сообщений в конце своего журнала).

Последователи считывают данные от лидера, как обычный потребитель, и применяют их к собственному журналу. Наличие у лидера последователей имеет хорошее свойство, позволяющее последователям пакетировать записи используемых ими журналов в своем журнале.

Как и в большинстве распределенных систем для автоматической обработки сбоев требуется точное определение термина “живой” узел. Для такого узла у **ADS** есть два условия:

1. Узел должен иметь возможность поддержки сессии с **ZooKeeper** (через механизм **Heartbeat ZooKeeper**).
2. Если узел является подчиненным, он должен реплицировать записи, происходящие на лидере, и не отставать “слишком далеко”.

В **ADS** узлы, удовлетворяющие этим двум условиям, называются “синхронизированными” во избежание неопределенности “живых” или “неисправных” узлов. Лидер отслеживает набор синхронизированных узлов и, если последователь терпит неудачу, зависает или отстает, лидер удаляет его из списка синхронизирующих реплик. Выявление зависающих и отстающих реплик контролируется конфигурацией *replica.lag.time.max.ms*.

В терминологии распределенных систем **ADS** старается обрабатывать модель “сбой/восстановление”, когда узлы внезапно прекращают работу, а затем восстанавливаются (возможно, не зная, что они потерпели неудачу). **ADS** не обрабатывает так называемые “византийские” сбои, когда узлы выдают произвольные или вредоносные ответы.

Теперь можно более точно определить, что сообщение считается зафиксированным, когда все

синхронизированные реплики для этой партиции применены к своему журналу. Потребителем могут быть считаны только зафиксированные данные. С другой стороны, у поставщиков есть возможность либо дожидаться фиксации сообщения, либо нет – в зависимости от предпочтения в отношении компромисса между задержкой и устойчивостью. Это предпочтение управляется настройкой *acks*, используемой поставщиком. Важно так же обратить внимание, что в топиках есть параметр минимального количества (*minimum number*) синхронизированных реплик, которое проверяется при запросе поставщиком подтверждения, что сообщение записано в полный набор синхронизированных реплик. Если у поставщика менее строгий запрос подтверждения, то сообщение может быть зафиксировано и считано, даже если количество синхронизированных реплик ниже минимального (например, оно может быть таким же низким, как у лидера).

Important: Гарантия, которую оказывает ADS, заключается в том, что зафиксированное сообщение не может быть потеряно, если хотя бы одна из синхронизированных реплик остается “живой”

Платформа **ADS** остается доступной при наличии сбоев узла после короткого периода отказа, но может быть недоступной при сетевых разделениях.

Глава 8

Сжатие журналов

Сжатие журналов гарантирует, что платформа **ADS** всегда сохраняет в журнале партиции топика по крайней мере последнее известное значение для каждого ключа сообщения. При этом учитываются варианты использования и сценарии, такие как перезагрузка кэшей после перезапуска приложения во время эксплуатационного обслуживания, восстановление состояния после сбоя приложения или системы.

До настоящего момента был описан только наиболее простой подход к хранению данных – когда старые данные удаляются из журнала через установленный период времени, или при достижении журналом определенного размера. Это хорошо работает для временных данных таких событий, как ведение журнала, где каждая запись фиксируется отдельно. Однако важным классом потоков данных является журнал изменений ключей и переменных (например, изменения в таблице базы данных).

Рассмотрим конкретный пример. Имеется топик, содержащий адреса электронной почты пользователей. Каждый раз, когда пользователь обновляет адрес своей электронной почты, в топик отправляется сообщение с использованием идентификатора пользователя в качестве первичного ключа. Теперь допустим, что за определенный период времени отправляются следующие сообщения для пользователя с id *123*, каждое сообщение соответствует изменению адреса электронной почты (сообщения для других идентификаторов опущены):

```
1  123 => bill@microsoft.com
2          .
3          .
4          .
5  123 => bill@gatesfoundation.org
6          .
7          .
8          .
9  123 => bill@gmail.com
```

Сжатие журнала дает более гранулированный механизм хранения, поэтому **ADS** гарантирует сохранение по крайней мере последнего обновления для каждого первичного ключа (например, *bill@gmail.com*). Так же гарантируется, что журнал содержит полный снапшот конечного значения для каждого ключа, а не только недавно измененные. Это означает, что нижестоящие потребители могут восстановить свое собственное состояние с этого топика без необходимости ведения полного журнала всех изменений.

Далее приведены несколько вариантов использования:

1. *Подписка на изменение базы данных.* Часто необходимо иметь набор данных в нескольких системах, и при этом одна из этих систем представляет собой некоторую базу данных (либо РСУБД, либо, возможно, новомодное хранилище ключ-значение). Например, может быть база данных, кэш, кластер поиска и кластер **Hadoop**. Каждое изменение БД должно отражаться в кэше, кластере поиска и, в конечном итоге, в **Hadoop**. В случае если идет обработка обновлений только в реальном времени, то нужен последний

актуальный журнал. Но в случае необходимости перезагрузки кэша или восстановления отказавшего узла поиска, может понадобиться полный набор данных.

2. *Поиск событий.* Это стиль разработки приложений, который совмещает обработку запросов с дизайном приложения и использует журнал изменений в качестве основного хранилища для приложения.

3. *Ведение журнала для обеспечения высокой работоспособности.* Процесс, выполняющий локальные вычисления, можно сделать отказоустойчивым, фиксируя вносимые в локальном состоянии изменения, с целью, чтобы другой процесс мог перезагрузить эти изменения и продолжить работу в случае сбоя первого. Конкретным примером является обработка счетчиков, агрегатов и других операций типа “group by” в потоковой системе запросов. **Samza**, платформа потоковой обработки данных в реальном времени, использует эту функцию именно для данной цели.

В каждом из перечисленных случаев в первую очередь необходимо обрабатывать поток изменений в реальном времени, но иногда, когда машина выходит из строя или данные должны быть повторно загружены или переработаны, необходимо выполнить их полную загрузку. Сжатие журнала позволяет использовать в этих целях один и тот же топик резервного копирования.

Общая идея довольно проста. Имея бесконечную сохраняемость журнала, каждое изменение в вышеуказанных случаях регестрировалось бы в него, и тогда состояние системы фиксировалось бы каждый раз с момента ее первого запуска. Используя этот полный журнал, была бы возможность восстановить систему в любой момент времени, воспроизведя первые N записей журнала. Но этот гипотетический полный журнал не очень практичен для систем, многократно обновляющих одну запись, поскольку журнал имел бы безграничный рост даже для стабильного набора данных. Простой же механизм ведения журнала, отбрасывающий устаревшие обновления, ограничен в пространстве, но и не является способом восстановления текущего состояния – восстановление системы с момента запуска не осуществляется посредством журнала, поскольку старые обновления могут быть не зафиксированы.

Сжатие журнала – это механизм, обеспечивающий гранулированное хранение каждой записи, а не крупнозернистое хранение во времени. Идея заключается в выборочном удалении записей, имеющих более новое обновление с тем же самым первичным ключом. Таким образом, журнал гарантированно имеет по крайней мере последнее состояние для каждого ключа.

Такая политика хранения может быть настроена для каждого топика, поэтому один кластер может иметь несколько топиков, где сохранение осуществляется по заданному объему данных или по времени ожидания, и другие топики, где сохранение обеспечивает сжатие.

Эта функциональность вдохновлена одним из старейших и наиболее успешных элементов инфраструктуры **LinkedIn** – службой кэширования базы данных, называемой Databus. В отличие от большинства лог-структурированных систем хранения платформа **ADS** построена для подписчиков и организует данные для быстрого линейного чтения и записи. **ADS** действует как хранилище достоверных данных, поэтому она полезна даже в ситуациях, когда вышестоящий источник данных не может быть воспроизведен.

8.1 Основы сжатия журналов

Далее приведен рисунок, отображающий логическую структуру ведения журнала **ADS** со смещением для каждого сообщения (Рис.8.1.).

Журнал **ADS** идентичен традиционному журналу. В нем плотно фиксируются последовательные смещения и сохраняются все данные. Сжатие журнала добавляет опцию для обработки его хвоста (на Рис.8.1. показан журнал с уплотненным хвостом). Важно обратить внимание, что данные в хвосте журнала сохраняют исходное смещение, назначенное при первой записи, которое никогда не изменяется. Также важно, что все смещения остаются действующими позициями в журнале, даже если сообщение с этим смещением сжато; в таком случае позиция неотличима от следующего наибольшего смещения, которое появляется в журнале. Например, на Рис.8.1. смещения *36*, *37* и *38* являются эквивалентными позициями, и начало чтения в любом из этих смещений выдает набор данных, начинающийся с *38*.

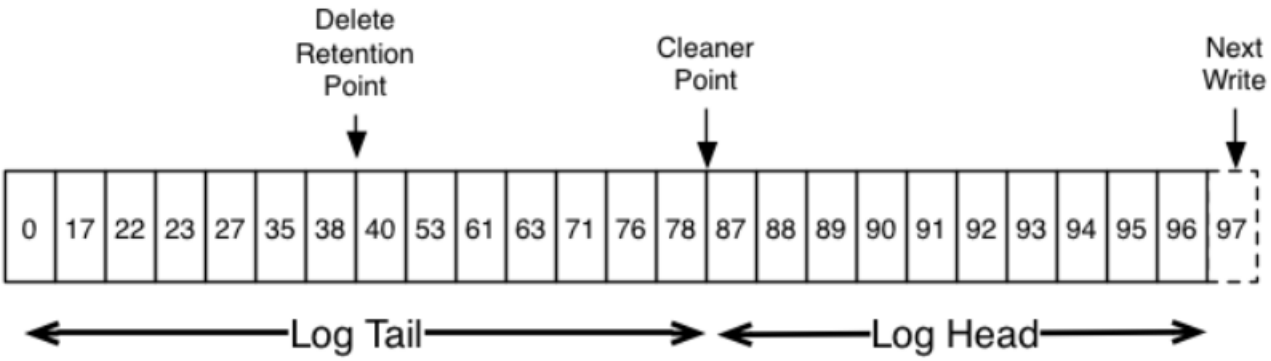

Рис.8.1.: Логическая структура журнала **ADS**

Сжатие также может удалять данные. Сообщение с ключом и нулевой полезной нагрузкой рассматривается на удаление из журнала и маркируется. Это приводит к удалению любых предыдущих сообщений с таким ключом (как и любые новые данные с таким же ключом), при этом маркированные на удаление данные являются особенными, поскольку они сами будут удалены из журнала через некоторое время. Момент времени, в который такие данные больше не сохраняются, на приведенном выше рисунке помечен как “Delete Retention Point”.

Сжатие выполняется в фоновом режиме с периодическим копированием сегментов журнала. Очистка не блокирует операции чтения и может регулироваться на настроенный объем пропускной способности ввода-вывода данных во избежание влияния на поставщиков и потребителей. Процесс сжатия сегмента журнала выглядит примерно как показано на Рис.8.2.

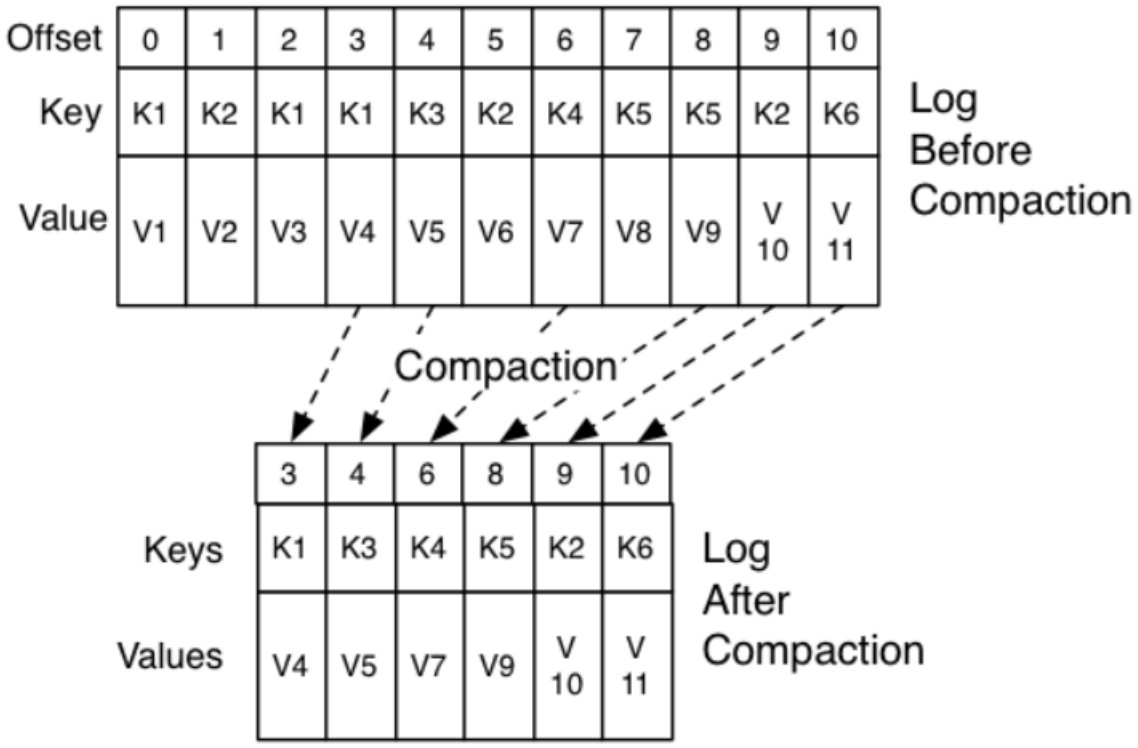

Рис.8.2.: Процесс сжатия сегмента журнала

8.2 Обеспечиваемые гарантии

Сжатие журнала обеспечивает следующие гарантии:

1. Любой потребитель, находящийся в голове журнала, видит каждое записанное сообщение; эти сообщения имеют последовательные смещения. Параметр топика *min.compaction.lag.ms* используется для гарантии минимального промежутка времени, затрачиваемого после записи сообщения прежде, чем оно будет сжато. То есть обеспечивается нижняя граница того, как долго каждое сообщение остается в неуплотненной голове журнала.
2. Всегда поддерживается упорядоченность данных. Сжатие никогда не нарушает порядок сообщений, а просто удаляет их.
3. Смещение для сообщения никогда не изменяется. Это постоянный идентификатор позиции в журнале.
4. Любому идущему от начала журнала потребителю видны, по крайней мере, окончательные состояния всех данных в порядке их записи. Кроме того, видны все маркированные под удаление данные при условии, что потребитель достигает головы журнала в течение меньшего периода времени, чем установлено в топике в параметре *delete.retention.ms* (по умолчанию *24 часа*). Другими словами, поскольку удаление маркированных данных происходит одновременно с операцией чтения, потребитель может пропустить эти данные при отставании более, чем на установленное в параметре *delete.retention.ms* время.

8.3 Детали сжатия журнала

Сжатие журнала выполняется посредством очистки log cleaner – объединением фоновых потоков, которые перезаписывают файлы сегментов журнала, и удалением данных, ключ которых отображается в голове журнала. Средство очистки работает следующим образом:

1. Выбирает журнал с самым высоким отношением между головой и хвостом.
2. Создает краткую сводку последнего смещения для каждого ключа в голове журнала.
3. Перезаписывает журнал от начала до конца, удаляя ключи, которые имеют более позднее появление в журнале. Новые, чистые сегменты немедленно подставляются в журнал, поэтому дополнительное пространство на диске требуется всего лишь для одного дополнительного сегмента журнала (а не для полной копии журнала).
4. Суммарно, голова журнала – это просто компактная хэш-таблица, использующая *24 байта* на запись. В результате с *8 ГБ* пространства под очистку одна итерация log cleaner может освободить около *366 ГБ* головы журнала (принимая данные равными *1 Кб*).