ISSN 2079-3316 ПРОГРАММНЫЕ СИСТЕМЫ: ТЕОРИЯ И ПРИЛОЖЕНИЯ № 3(17), 2013, с.??—??

УДК (Код УДК!)

А.А. Малышевский

Использование экосистемы Hadoop при обработке данных дистанционного зондирования Земли

Аннотация. В докладе описаны технические аспекты применения фреймворка для распределенных вычислений Hadoop и смежных технологий в задаче обработки данных дистанционного зондирования Земли. Специфика предметной области такова, что подавляющее большинство программных библиотек, применяемых для обработки данных дистанционного зондирования Земли (ДДЗ), реализованы на C/C++; разработчики, работающие в этой предметной области, также привыкли программировать на C++. Hadoop же разработан на Java и для Java. Однако у Hadoop есть и инструменты, позволяющие работать с любыми исполняемыми файлами: Hadoop Stream, Hadoop Pipes. Набор их функционала заметно ограничен по сравнению с привычными для Hadoop инструментами, но для решения конкретных прикладных задач этого стандартного набора может быть достаточно. При решении задачи обработки ДДЗ использовался Hadoop Pipes.

Кроме самой системы Hadoop рассказано о практическом использовании обменного формата хранения данных Avro, используемого для распределения сцен съемки на узлы обработки, вариантах его сериализации и десериализации, проблемах конкретного применения.

Приведены результаты тестов производительности разных стадий обработки данных в распределенном режиме.

Доклад может представлять интерес для разработчиков кластерных распределенных систем обработки больших данных, особенно с учетом рассматриваемого практического опыта применения в гетерогенной среде (java + наследованный код C++) и выбора эффективного формата хранения и обмена данных в кластере.

Hadoop Pipes, avro, ДЗЗ, распределенные вычисления

Введение

Данные с космических аппаратов серии «Метеор-М» при приеме сохраняются в файлы битового потока размером до нескольких десятков гигабайт. Представляют они собой набор битовых инфор-

мационных строк одинакового состава, расшифровка которых описана в соответствующей документации. Данные размещены в потоке плотно для экономии радиоканала сброса информации, при этом никаких алгоритмов сжатия к ним не применяется. Битовому потоку предшествует человекочитаемая xml-подобная структура с набором параметров, описывающих сеанс записи данных и связи с наземным пунктом приема. Таким образом, можно сказать, что структура файла смешанная и очень специфическая.

. Традиционный способ обработки таких файлов связан с существенными задержками во времени при многократном копировании файлов по сети и при загрузке данных в оперативную память, размер которой ограничивает размер решаемой задачи

Задача обработки ДДЗ хорошо укладывается в концепцию big data: “to big to move”, “to big to fit in memory”. Такие задачи обработки решаются делением решаемого поля задачи на сегменты, которые могут быть обработаны относительно независимо друг от друга. При выборе принципа деления на сегменты важным фактором является обеспечение локальности обрабатывающих мощностей по отношению к месту хранения сегментов (data locality). Эти сегменты распределяются между доступными и свободными от нагрузки узлами (серверами обработки), каждый из которых хранит и обрабатывает свой сегмент. Масштабирование достигается путем добавления новых узлов в кластер, при этом увеличивается не только вычислительный ресурс, но и ресурс хранения данных.

Однако широко распространенные в среде разработки C++ реализации MPI не имеют эффективной встроенной поддержки распределенной файловой системы.

Удобным и эффективным решением распределенной обработки является экосистема Hadoop, включающая Hadoop Distributed FileSystem для распределенного хранения данных. Однако экосистема Hadoop разработана на Java и для Java, в то время как значительная часть библиотек функций расчета движения космического аппарата и перепроецирования изображений использует

C/C++. Поэтому в качестве связующей Hadoop и C/C++-код технологии был использован механизм Hadoop Pipes.

Специфика задачи обуславливает требование подавать данные на вход C++-приложению порциями, соответствующими условным сценам съемки. Таких сцен в исходном файле может быть от нескольких штук до сотни. Размер памяти, занимаемой каждой из таких сцен, также колеблется от нескольких десятков мегабайтов, до нескольких сотен в оригинальном виде. Это также должно быть учтено при решении задачи.

1. Используемые технологии

1.1. Экосистема Hadoop (CDH)

Hadoop — проект фонда Apache Software Foundation, свободно распространяемый набор утилит, библиотек и фреймворк для разработки и выполнения распределённых программ, работающих на кластерах из сотен и тысяч узлов. Используется, например, для реализации поисковых и контекстных механизмов Yahoo! и Facebook. Разработан на Java в рамках вычислительной парадигмы MapReduce, согласно которой приложение разделяется на большое количество одинаковых элементарных заданий, выполняемых на тех узлах кластера, на которых размещены соответствующие «порции» данных и естественным образом сводимых в конечный результат.

Считается одной из основополагающих технологий «больших данных». Вокруг Hadoop образовалась целая экосистема из связанных проектов и технологий, многие из которых развивались изначально в рамках проекта, а впоследствии стали самостоятельными.

Одной из основных целей Hadoop изначально было обеспечение горизонтальной масштабируемости кластера посредством добавления недорогих узлов, без прибегания к мощным серверам и дорогим сетям хранения данных.

1.2. Распределенная файловая система Hadoop Distributed File System (HDFS)

HDFS (*Hadoop Distributed File System*) — файловая система, предназначенная для хранения файлов больших размеров, поблочно распределённых между узлами вычислительного кластера. Все блоки в HDFS (кроме последнего блока файла) имеют одинаковый размер, и каждый блок может быть размещён на нескольких узлах, размер блока и коэффициент репликации (количество узлов, на которых должен быть размещён каждый блок) определяются в настройках на уровне файла. Благодаря репликации обеспечивается устойчивость распределённой системы к отказам отдельных узлов. Организация файлов в пространстве имён — традиционная иерархическая: есть корневой каталог, поддерживается вложение каталогов, в одном каталоге могут располагаться и файлы, и другие каталоги.

При хранении данных в HDFS и их использовании в map/reduce задачах рекомендуется группировать данные в большие файлы, которые при этом поддерживают возможность обработки их частей независимо друг от друга. В таком случае Hadoop обеспечивает наилучшую производительность. Для рассматриваемого случая файлы в HDFS соответствуют файлам битового потока с внутренним делением для распределенной обработки на наборы информационных строк единой структуры, соответствующим описанным выше сценам.

1.3. Технология интеграции Hadoop Pipes

Технология Hadoop pipes позволяет запускать реализованные в коде C++ map/reduce задачи в рамках стандартной инфраструктуры Hadoop. Код должен быть структурирован так, чтобы он мог запускаться в рамках одного процесса и не иметь никаких зависимостей от результатов его параллельного выполнения в рамках

других процессов в кластере. Это код при работе в системе «оборачивается» специальной C++-библиотекой управления, которая берет на себя функции связи этого C++ кода с инфраструктурой Hadoop. Есть возможность так же использовать Python и другие скриптовые языки программирования.

1.4. Avro

Выбор обменного формата для распределенной обработки данных ДЗЗ играет важную роль в построении архитектуры комплекса процессов обработки этих данных. Для оптимального результата, особенно при использовании вкупе с Hadoop, формат должен соответствовать ряду требований:

- Формат должен позволять делить файлы на блоки, которые можно будет обрабатывать одновременно независимо друг от друга, в терминологии Hadoop это называется “splittable”.
- Формат должен быть хорошо сжимаемым, чтобы большое количество данных занимало на диске как можно меньше физической памяти. Причем желательно, чтобы разные блоки одного файла могли сжиматься независимо друг от друга, только так удастся удовлетворить первые два требования одновременно.
- Скорость записи и чтения сжатых данных должна быть достаточно быстрой.
- В нашем случае подготовка файла для распределенной обработки выполняется Java-приложением, в то время как обработка выполняется C++-приложением. Таким образом, формат должен подразумевать наличие механизма, который удовлетворит этой особенности. И чем проще этот механизм реализовать, тем лучше.
- Дополнительным плюсом является поддержка версионности структуры данных, т.е. возможности менять схему данных так, чтобы оставалась возможность обрабатывать данные, сохраненными ранее в старой схеме.
- Формат должен быть достаточно распространенным. Это обеспечивает определенную надежность, снимает часть рисков.

Вышеприведенным требованиям удовлетворяет система сериализации Avro. Avro Apache — это программная библиотека функ-

ций, позволяющих сериализовать данные в формате со встроенной схемой. Вместо использования библиотек сгенерированных проксиобъектов и строгой типизации формат Avro интенсивно задействует схемы, которые сохраняются вместе с сериализованными данными. Сопровождение данных в формате Avro сопутствующей схемой данных позволяет любому приложению десериализовать данные. Таким образом, можно сказать, что формат данных является самоописываемым.

В нашем случае при разработке использование формата Avro происходит следующим образом:

(1) первой создается/редактируется/дополняется схема данных в формате JSON
(2) для Java-приложения, выполняющего подготовку к распределенной обработке (ingest), с помощью среды сборки Maven по указанной схеме генерируется набор исходных кодов Java-классов; для записи данных в файл остается создать и заполнить сгенерированные объекты и вызвать библиотечную функцию сохранения, данные будут сохранены в Avro-файл
(3) на стороне обработчика, т.е. на стороне C++, с помощью консольной утилиты avrogencpp так же по схеме генерируется набор исходных кодов структур уже для языка C++; для чтения остается вызвать библиотечную функцию чтения и соответствующие структуры будут автоматически заполнены значениями

Таким образом, использование инструментария Avro позволяет значительно автоматизировать работу разработчиков в части создания классов, разбирающих обменный формат файлов, что особенно важно в области ДДЗ.

Серьезной проблемой является то, что Hadoop Pipes может подавать на вход в C++-код только лишь строки (std::string). При этом передать необходимо одну или несколько сцен, а файлы в HDFS удобнее держать с большим количеством сцен, чем необходимо.

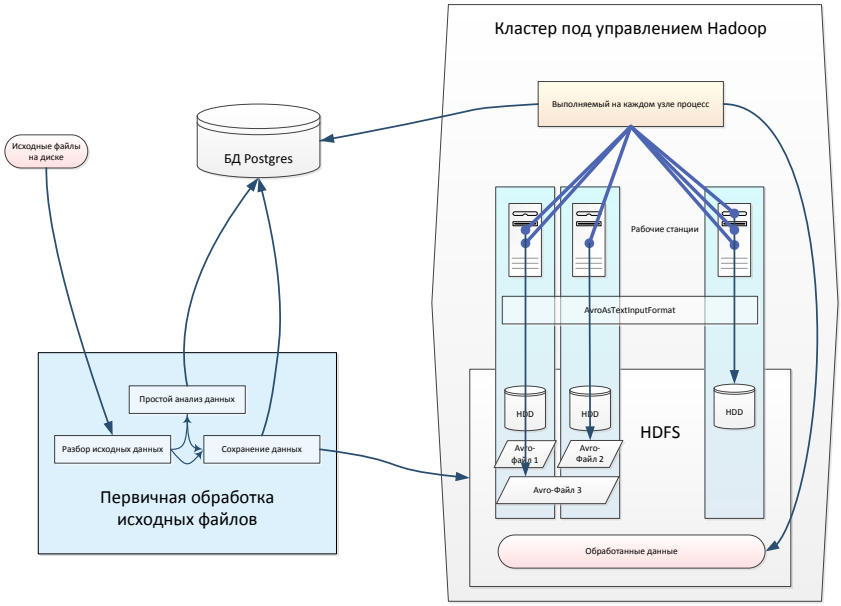
Выходом из ситуации стал специальный формат чтения, InputFormat в терминах Hadoop, разработанный специально для ис-

пользования технологий Hadoop Stream или Hadoop Pipes – AvroAsTextInputFormat. Суть его заключается в том, что при чтении фреймворк считывает данные в доменные объекты, затем сериализует их в JSON и отправляет этот JSON через Hadoop Pipes на вход C/C++-, или любому другому, поддерживающему SWIG языку, приложению. На стороне приложения уже самими разработчиками производится процедура десериализации JSON в доменные объекты и с ними можно работать.

У подхода есть очевидный минус – приходится тратить время на сериализацию и десериализацию данных, но современные методы и библиотеки сериализации/десериализации в/из JSON развиты достаточно, чтобы ресурсные потери на эту процедуру не оказали существенного влияния на общее время обработки.

2. Детали использования экосистемы для решения поставленной задачи

2.1. Путь движения данных до конечного пользователя



2.2. Проблемы использования формата Avro

При всех плюсах формата Avro именно при использовании этой технологии мы встретили наибольшее число проблем. Большинство из них обусловлено выбором механизма передачи сцен с помощью AvroAsTextInputFormat.

В терминах математических отношений встроенная в библиотеку Avro операция десериализации из JSON в доменный объект не является обратным отношением к операции сериализации в JSON с помощью AvroAsTextInputFormat. Другими словами, если сериализовать объект в JSON с помощью AvroAsTextInputFormat, а затем десериализовать его обратно, нет гарантии, что мы получим тот же

объект, что и сериализовали. Изучение исходного кода этого класса, а также смежных с ним, показало, что в этом случае почему-то используется не стандартный механизм сериализации avro-объекта в JSON, а другой, не имеющий парного механизма десериализации. В системе bug-tracking Jira для Avro заведен об этом «баг», но он уже закрыт с формулировкой «Not A Problem»; объясняется это самим Дугом Каттингом, одним из основных разработчиков Hadoop и Avro, тем, что десериализация в таких случая проводится там, где ничего не знают об Avro, поэтому проводится она в полуручном режиме и свойство обратности здесь не нужно: https://issues.apache.org/jira/browse/AVRO-1456. На мой взгляд, сомнительная причина.

Эта проблема приводит в частности к невозможности использовать такой тип данных Avro как union. Тип данных union в схеме формата avro-файла говорит о том, что поле может одного из нескольких типов данных, например int или double. Чаще всего union используются для полей, значений для которых может не быть; в таких случаях с помощью union указывают, что поле может быть типа, например, string или типа null, т.е. отсутствовать. У нас таких полей много. И даже наличие хотя бы одного такого поля приводит к невозможности воспользоваться предложенной схемой – данные не удается считать на стороне C++-приложения. Из-за этого нам пришлось вместо union использовать тип array, и в случае отсутствия значения массив оставался пустым. Это достаточно неудобно как со стороны, сохраняющей данные, так и со стороны, использующей объекты с прочитанными данными, в которых вместо поля определенного типа приходится иметь дело с массивом из 0 или 1 элемента.

То же касается и использования типа bytes. По задумке этот тип представляет собой просто последовательный набор байт. Используется он для передачи основной информации со спутника – целевой информации, в результате обработки формирующее растровые данные. В предложенной схеме данные сохраненные в этом типе так же не удается прочитать. Для решения этой проблемы нам пришлось вместо типа bytes использовать массив (array) с це-

лыми числами (int) внутри. Это привело к почти четырехкратному росту потребления оперативной памяти при сохранении данных в файл, при чтении данных из файла и сериализации их в JSON, при десериализации их из JSON и пользовании получившимися объектами (если их не клонировать в аналогичные объекты, но с массивом байтов, а не целых чисел int).

Процессы, занимающиеся чтением сцен из Avro-файлов и сериализующие их в JSON, требуют аномально много оперативной памяти даже без учета проблем с типом bytes. Ввиду наличия большого запаса по памяти природу этого эффекта выяснять не стали

И самой главной проблемой стала скорость сериализации в JSON и особенно десериализации получившейся строки в JSON на стороне C++. Объект, занимающий в исходном файле около 70 мб десериализуется из JSON на узле стандартной рабочей станции нашего кластера около 20 секунд, объект объёмом 300-400мб - около 140 секунд. Это дольше времени, необходимого на саму обработку соответствующей сцены, почти теряется смысл распределенной обработки при таких накладных расходах. Исследование показало, что проблема именно во встроенном в библиотеки Avro механизме сериализации-десериализации в/из JSON, на достаточно больших объектах этот механизм работает на порядок медленнее аналогов. Это достаточно удивительно, учитывая, что для этих целей Avro использует быструю надежную библиотеку Jackson, которая сама по себе работает значительно быстрее. Видимо, дело в нюансах формата Avro, из-за которых теряется более 90% производительности оригинальной библиотеки Jackson.

2.3. Решение проблем передачи данных в C++-код

После получения результатов тестирования в условиях, близких к реальным, было принято решение отказаться от передачи данных в C++-код через сериализацию данных в JSON и последующую десериализацию через AvroAsTextInputFormat в пользу механизма с передачей в C++-код имен файлов, в которых хранятся соответствующие схемы.

Такой подход заставил научиться читать файлы файловой системы HDFS из C++-кода и создать специальный InputFormat, т.е. класс для Hadoop, читающий входные данные, который обеспечил бы т.н. локальность данных. Под локальностью данных подразумевается высокая вероятность того, что каждому узлу будет передано имя такого файла, одна из копий которого присутствует именно на этом же узле.

Отказ от передачи данных через (Avro-)JSON избавляет от всех описанных выше проблем использования формата, с которыми мы столкнулись. Однако есть и минус – файлы приходится хранить по одному на сцену, а не по одному на 10-100 сцен. Это менее эффективно, но потеря производительности не должна быть даже близко столь ощутимой, каковы потери при сериализации/десериализации вышеописанным способом.

2.4. Проблемы использования HDFS в контексте классических задач обработки данных ДЗЗ

Проблемы использования HDFS в контексте классических задач обработки данных ДЗЗ сводятся к проблемам чтения и записи файлов из и в HDFS из C++-кода. Среда C++ является чужеродной для HDFS, а потому какие-либо сложности или недостатки при работе с файловой системой вполне предсказуемы.

Основной, самый очевидный, способ работать с файлами из C++ – это использование библиотеки libhdfs. Библиотека эта для взаимодействия с HDFS активно использует Java-код через JNI. Плохо это, во-первых, тем, что усложняет процесс, понижая, хоть и незначительно, надежность, скорость работы. Во-вторых, сильно усложняет отладку в сложных случаях. В-третьих, при использовании JNI, в частности в этом направлении при вызове Java-кода из C++, могут иметь действия различные нежелательные побочные эффекты, которые с учетом предыдущей причины может быть сложно исправить. Например, у нас сначала не заработало чтение avro-файла из HDFS, проблема была якобы в несовместимости сохраненных данных с форматом. Оказалось, что в момент вызова

JNI менялась локаль по умолчанию. Почему – не понятно. В результате разделителем целой и дробной части чисел с плавающей точкой начинала считаться не точка, а запятая, и число, сохраненное с точкой, не могло прочитаться. В-четвертых, это приводит к иногда нежелательным зависимостям от java- и других библиотек.

Другой способ работать с HDFS – использовать webhdfs, в том числе через библиотеку libwebhdfs. Webhdfs – это REST API для HDFS. Работа с файлами в таком случае ведется посредством HTTP-запросов. Такой метод не использует JNI, а, значит, лишен всех четырех описанных выше недостатков. Однако этот подход имеет один существенный недостаток, который, на мой взгляд, в случае, когда в работе с файловой системой HDFS задействовано небольшое количество различных операций и этот набор редко меняется, перевешивает все недостатки использования JNI. Производительность этого подхода ниже, чем при использовании libhdfs, особенно это заметно при интенсивном (в смысле нагрузок, а не разнообразии операций) использовании HDFS.

После того, как были решены все возникшие проблемы с libhdfs за приемлемый отрезок времени, именно подход с использованием JNI был выбран в качестве рабочего.

Заключение

Основным продуктом обработки ДДЗ в задаче является геопривязанный файл изображения GeoTIFF в проекции EPSG 4326.

В тестировании у нас было задействовано 5 рабочих станций со следующими характеристиками: 2x CPU Intel Xeon E5-2660 v2 (10 ядер, 25 МБ кэш, 2.2GHz), 128GB, 10x HDD 600GB/10K, InfiniBand HCA.

Таблица 1. Данные о производительности

Исходный файл	В один поток	В кластере. Первый вариант	В кластере. Второй вариант
Файл данных с прибора МСУ-МР, объемом 3,5 Гб + данные с прибора БСКВУ объемом 0,3 Гб, покрывающие растр в 230 тысяч строк шириной 1572 пикселя в трех каналах		160с.	
Файл данных с прибора КМСС, объемом 2 Гб, покрывающий растр в 80-100 тысяч строк шириной 7926 пикселей в трех каналах		285с.	

Об авторе:

Александр Александрович Малышевский

Главный программист ЗАО «СТТ груп», Москва.

e-mail: a.malyshevskiy@cttgroup.ru

A. A. Malyshevskiy. Hadoop ecosystem application in remote sensing data processing.

Abstract. (Перевод аннотации на английский язык!).

Hadoop Pipes, avro, remote sensing, distributed computing