НОВЫЕ ТЕХНОЛОГИИ РАСПРЕДЕЛЕННОГО ХРАНЕНИЯ И ОБРАБОТКИ БОЛЬШИХ МАССИВОВ ДАННЫХ

О.В. Сухорослов

Институт системного анализа РАН
117312, г. Москва, пр-т 60-летия Октября, д. 9

Аннотация. В статье рассматриваются новые технологии, позволяющие организовать распределенное хранение и параллельную обработку больших объемов данных в крупномасштабных кластерных системах. Речь идет о петабайтах данных, для хранения и обработки которых необходимы значительные вычислительные ресурсы. В качестве таких ресурсов рассматриваются кластерные системы, состоящие из тысяч серверов. В подобных распределенных системах остро стоят вопросы обеспечения отказоустойчивости и бесперебойного функционирования сервисов хранения и обработки данных. Другой важной проблемой является создание высокоуровневой модели программирования процессов обработки данных на подобных системах, скрывающей от пользователя детали распределения данных и планирования вычислений в ненадежной распределенной среде. В статье приводится описание оригинальных технологий, нацеленных на решение указанных проблем и уже применяемых в крупнейших информационных системах. Поскольку большинство подобных технологий являются закрытыми коммерческими разработками, особое внимание уделено описанию создаваемых в настоящее время открытых (open source) аналогов данных технологий.

Annotation. The article describes emerging technologies for distributed storage and processing of large data sets on cluster systems. These technologies enable parallel processing of petabytes of data on thousands of machines. Such large-scale distributed computing systems pose some challenges concerning enabling automatic fault-tolerance and high availability of services. Another important issue is a development of high-level programming

models and data processing tools which hide from a user all low-level details of data distribution, scheduling and process synchronization in a distributed environment. The article presents several emerging technologies aimed on solving the above-mentioned problems. Some of these, mostly proprietary, technologies are already used in largest commercial information systems, while others provide open source analogues of closed proprietary solutions.

Введение

В современном мире все большую роль играют технологии, обеспечивающие эффективное хранение и обработку данных. Связано это с наблюдаемым с конца прошлого века лавинообразным ростом информации. Современные задачи и приложения, связанные с анализом данных, предъявляют особые требования к вычислительным ресурсам, значительно превышающие возможности отдельных компьютеров.

Описываемые в статье технологии берут свое начало внутри компании Google, крупнейшей поисковой системы Web. В настоящее время Web насчитывает десятки миллиардов страниц. Роботы поисковой системы круглосуточно загружают петабайты данных с содержанием новых и измененных Web-страниц. Загруженные данные подвергаются различным процедурам обработки, связанным с построением индекса Web, вычислением индексов цитирования отдельных страниц и т.д. [1]. Полученные в результате обработки данные, измеряемые также петабайтами, размещаются в долговременном хранилище и используются для генерации результатов поисковых запросов пользователей. В настоящее время появляется все больше Web-приложений, накапливающих и фильтрующих большие объемы информации, таких как социальные сети и различные сервисы агрегации. Вычислительная инфраструктура Google насчитывает сотни тысяч серверов, размещенных в нескольких центрах обработки данных по всему миру. По опубликованной статистике [2] в сентябре 2007 г. в инфраструктуре Google было запущено в общей сложности более 2 млн. заданий со средним количеством задействованных машин около 400. Суммарный объем входных данных заданий составил около 400 петабайт.

Другой областью, где распространены приложения, связанные с хранением и обработкой больших объемов данных, являются научные вычисления. Чаще всего это выражается в необходимости проведения трудоемкого анализа собранных массивов экспериментальных данных для получения новых научных результатов. Ярким примером является крупномасштабный международный проект в области физики высоких энергий по созданию Большого адронного коллайдера (БАК). Предполагается, что во время проведения экспериментов БАК будет генерировать

порядка 15 петабайтов первичных экспериментальных данных в год. Для хранения и обработки этих данных планируется использовать специально созданную Grid-инфраструктуру WLCG/EGEE, насчитывающую более 200 ресурсных центров в десятках стран с общей численностью в тысячи машин. Подобного масштаба задачи встречаются и в других областях исследований, таких как биоинформатика, астрофизика, науки о Земле и т.д.

Для достижения приемлемого времени работы подобным приложениям необходимы ресурсы уровня суперкомпьютеров или кластеров с сотнями и тысячами узлов. Несмотря на значительный рост емкости носителей данных, временные характеристики доступа к данным изменились слабо, что приводит к необходимости кластеризации носителей для обеспечения приемлемой пропускной способности. Для обеспечения надежности хранения данных также необходима избыточная репликация данных.

Постоянный рост объемов обрабатываемых данных требует соответствующего наращивания вычислительных ресурсов, в связи с чем используемая вычислительная среда должна обладать высокой масштабируемостью. В настоящее время в области массированной обработки данных наблюдается переход от специализированных суперкомпьютерных архитектур к более экономичным и масштабируемым, но менее надежным кластерным системам из недорогих серверов массового производства.

Реализация процедуры обработки данных в подобной распределенной среде сопряжена с решением таких непростых задач, как разбиение и распределение данных между процессорами, планирование и балансировка нагрузки, обработка отказов отдельных узлов, сбор и агрегация промежуточных результатов. Необходимость непосредственной реализации данных механизмов при программировании процедуры обработки данных является серьезным препятствием на пути широкого внедрения подобных систем. Поэтому необходимо, чтобы соответствующие технологии уже содержали в себе реализации данных механизмов и предоставляли пользователю высокоуровневые модели программирования, скрывающие от него детали реализации вычислений в ненадежной распределенной среде.

1. Распределенные системы хранения данных

В главе рассматриваются новые системы, ориентированные на хранение больших массивов данных в распределенных кластерных системах. Условно их можно разделить на два класса: распределенные файловые системы (Google File System, Hadoop Distributed System) и распределенные хранилища структурированных данных (Google BigTable, HBase). Рассматриваемые системы имеют принципиальные отличия от традиционных файловых систем и реляционных баз данных.

1.1. Google File System

Распределенная файловая система Google File System (GFS) является закрытой разработкой компании Google, используемой для хранения больших массивов данных. Внутри Google функционирует более 200 GFS-кластеров, крупнейшие из которых насчитывают более 5 тысяч машин, хранящих около 5 петабайт данных и обслуживающих порядка 10 тысяч клиентов [3]. Описание GFS было опубликовано в работе [4].

Как и любая распределенная файловая система, GFS ориентирована на обеспечение высокой производительности, масштабируемости, надежности и доступности. Отличия архитектуры GFS от других распределенных файловых систем, таких как AFS, xFS, Lustre, обусловлены спецификой приложений и вычислительной инфраструктуры Google. Данная инфраструктура построена из большого количества недорогих серверов массового производства, что приводит к постоянным отказам оборудования. Поэтому внутренние технологии Google в обязательном порядке предусматривают механизмы обнаружения и автоматического восстановления после отказов отдельных машин. Специфика приложений Google такова, что хранимые файлы обычно имеют большой размер (многие гигабайты) по сравнению с обычными файловыми системами. Как правило, содержимое файлов изменяется только за счет записи новых данных в конец файла. Запись может вестись одновременно несколькими клиентами, поэтому требуются гарантии атомарности отдельных операций. После окончания записи, файлы в основном только считываются, причем – последовательно.

Операции чтения больших порций данных могут происходить в потоковом режиме. Стабильно высокая пропускная способность более важна для приложений, чем низкое время отклика при выполнении отдельных операций.

Файлы в GFS организованы в иерархическую структуру директорий и идентифицируются при помощи полных путей. GFS предоставляет интерфейс с типичными для файловой системы операциями *create*, *delete*, *open*, *close*, *read* и *write*. Дополнительно поддерживаются операции *snapshot* и *record append*. Первая операция создает копию файла или директории. Вторая операция реализует атомарную запись в конец файла. В отличие от многих файловых систем, в целях упрощения реализации и повышения эффективности GFS не реализует стандартный POSIX-интерфейс.

GFS – распределенная система, организованная по схеме “главный-подчиненный” (master-slave). В системе есть один главный сервер (master) и несколько chunk-серверов. Файлы разбиваются на блоки (chunk) фиксированного размера (обычно 64 Мб), размещаемые на chunk-серверах. Для обеспечения надежности и распределения нагрузки каждый блок реплицируется на нескольких (по умолчанию, трех) серверах, размещенных в различных серверных стойках. Масштабируемость системы достигается за счет возможности “горячего” подключения новых chunk-серверов, а также описываемых ниже стратегий, позволяющих разгрузить главный сервер и, тем самым, избежать возникновения в системе узкого места.

Главный сервер хранит в памяти все метаданные файловой системы, включая пространства имен, права доступа, отображение файлов в блоки и текущие местоположения всех реплик блоков. Главный сервер также контролирует общесистемные процессы, такие как размещение и репликация блоков, назначение главной реплики, удаление неиспользуемых блоков и миграция блоков между серверами. Chunk-сервера периодически отправляют главному серверу сообщения с информацией о своем состоянии. Главный сервер использует ответы на эти сообщения для передачи инструкций chunk-серверам.

GFS-клиент реализует интерфейс (API) файловой системы и взаимодействует с главным и chunk-серверами от имени приложения. Важно отметить, что клиент обращается к главному серверу только за метаданными, а все операции по чтению и

записи данных осуществляются напрямую с chunk-серверами. Это позволяет уменьшить нагрузку на главный сервер.

Рассмотрим схему взаимодействия клиента с GFS на примере чтения файла. Сначала клиент вычисляет по указанному байтовому отступу в файле номер соответствующего блока, после чего отправляет главному серверу запрос с именем файла и номером блока. В ответ главный сервер передает клиенту уникальный идентификатор данного блока и адреса его реплик. Клиент кэширует эту информацию локально, после чего отправляет запрос одному (например, ближайшему) из chunk-серверов с репликой блока. В запросе клиент указывает идентификатор блока и байтовый отступ внутри блока. Дальнейшие операции чтения данных из этого блока осуществляются клиентом автономно, без участия главного сервера, до тех пор пока информация в кэше не будет удалена по истечении времени жизни или при повторном открытии файла. В целях дополнительно оптимизации нагрузки, клиент обычно запрашивает у главного сервера информацию сразу о нескольких блоках.

GFS реализует ослабленную модель целостности данных, достаточную для работы целевых приложений и, в то же время, относительно простую и эффективную в реализации. Операции изменения метаданных, такие как создание файлов, являются атомарными и выполняются главным сервером. Операции изменения данных разделяются на два типа: запись с определенным отступом (write) и присоединение (record append). В первом случае не гарантируется атомарность выполнения операции, то есть записанные одним клиентом данные могут быть “перемешаны” с данными другого клиента с несоблюдением указанных отступов. Но при этом гарантируется, что все клиенты будут видеть одни и те же данные, независимо от используемой реплики. Во втором случае гарантируется, что данные будут записаны атомарно (непрерывным фрагментом) как минимум один раз. При этом внутри данного региона файла могут появляться неполные фрагменты и дубликаты записываемых данных, связанные с возникновением ошибок при записи данных на уровне реплик. В этом случае GFS не гарантирует, что все реплики будут абсолютно идентичны, а лишь - что присоединяемые данные будут записаны во все реплики атомарно. Описанная модель целостности накладывает обязательства на клиентов GFS по проверке считываемых данных на наличие неполных фрагментов и дубликатов. Подобную проверку можно

реализовать с помощью вставки в записываемые данные контрольных сумм и идентификаторов записей.

В целях снижения нагрузки на главный сервер, операции записи данных в блок координируются одним из chunk-серверов, хранящих реплику блока. Данная реплика называется главной (primary). Центральный сервер управляет назначением центральных реплик путем выдачи реплике временной аренды (lease) с возможностью ее продления. Главная реплика управляет порядком внесения изменений при одновременной записи данных в ее блок, координируя действия остальных, подчиненных (secondary) реплик блока. При записи данных клиент получает от главного сервера адрес главной реплики блока, после чего взаимодействует напрямую с chunk-серверами. Клиент отправляет данные на все реплики блока и, после получения подтверждений от реплик, обращается в главной реплике с запросом на запись данных. Главная реплика присваивает каждому поступающему запросу номер, применяет изменения к локальным данным в соответствии с номером запроса и передает запрос остальным репликам с указанием его номера. После получения уведомлений о выполнении запроса от подчиненных реплик, главная реплика возвращает ответ клиенту. В случае возникновения ошибок при записи данных, клиент пытается повторить описанную процедуру повторно.

Для эффективного использования сетевых ресурсов, записываемые данные передаются репликам путем их пересылки по цепочке от одного chunk-сервера к другому. При получении первой порции данных сервер тут же начинает их передачу в конвейерном режиме ближайшему (в терминах топологии систем) серверу. Это позволяет полностью задействовать пропускную способность каждой машины, избежать возникновения узких мест и минимизировать задержку при передаче данных.

GFS поддерживает операцию shapshot, позволяющую быстро создавать копии файлов или целых директорий. Эта операция используется приложениями для ветвления наборов данных или сохранения копии данных перед их изменением, с возможностью последующего отката к старой версии. Для реализации snapshot используется стандартная стратегия отложенного копирования (copy-on-write) – копии данных создаются во время первой последующей записи в копируемые данные. При

выполнении операции главный сервер отзывает все аренды у главных реплик блоков, входящих в копируемые данные, что позволяет ему контролировать операции записи.

Главный сервер GFS управляет размещением реплик в системе. При создании нового блока он назначает chunk-сервера для хранения реплик блока исходя из такой информации о серверах, как объем используемого дискового пространства, число недавно размещенных на сервере реплик, положение в сетевой топологии. Главный сервер постоянно контролирует состояние chunk-серверов и хранимых на них реплик с помощью сообщений, получаемых от серверов. В случае выхода из строя сервера или повреждения реплики, главный сервер автоматически выполняет размещение новых реплик до тех пор, пока не будет восстановлен заданный уровень репликации данных. Аналогичная процедура проводится в случае, если пользователь увеличивает уровень репликации хранимых данных. Кроме того, главный сервер периодически производит балансировку системы, перераспределяя реплики между chunk-серверами. Наконец, главный сервер производит периодическую “сборку мусора”, заключающуюся в физическом удалении формально удаленных или поврежденных данных.

Как уже упоминалась, вычислительная инфраструктура Google подвержена постоянным и неизбежным отказам оборудования. GFS учитывает это обстоятельство путем реализации механизма автоматического восстановления после отказов. Главный и chunk-сервера реализованы таким образом, чтобы максимально быстро восстанавливать свое состояние после аварийного перезапуска.

Главный сервер сохраняет все операции и изменения своего состояния в log-файле, который автоматически реплицируется на нескольких машинах. При превышении максимального размера log-файла создается резервная копия состояния главного сервера, а log-файл очищается. В случае аварийной остановки главного сервера, он автоматически перезапускается и восстанавливает свое состояние по последней резервной копии и log-файлу. Также некоторое время уходит на получение главным сервером информации о местоположении реплик от chunk-серверов, поскольку данная информация не сохраняется главным сервером. В случае отказа самой машины, главный сервер автоматически запускается на другой машине кластера и восстанавливает состояние с помощью реплицированных файлов. Клиенты автоматически устанавливают соединение с новым сервером по истечении таймаута,

поскольку главный сервер адресуется с помощью DNS-имени. В системе также присутствует несколько резервных главных серверов (shadow master), которые предоставляют доступ к системе в режиме чтения, даже если главный сервер недоступен.

В случае аварийной остановки главного сервера, он автоматически перезапускается и восстанавливает свое состояние по хранимым на диске данным. В случае отказа самой машины с chunk-сервером, главный сервер автоматически удаляет ссылки на хранимые сервером реплики и начинает процесс их восстановления с помощью реплик на других серверах. Напомним, что каждый блок по умолчанию хранится на трех серверах. На chunk-серверах также могут случаться сбои жестких дисков, приводящие к частичному повреждению хранимых реплик. Поэтому каждый chunk-сервер выполняет проверку целостности хранимых им данных с помощью контрольных сумм. В случае если контрольная сумма не совпадает, то chunk-сервер сообщает об этом главному серверу, который создает новую реплику блока.

В заключении отметим главные особенности распределенной файловой системы GFS:

- Высокая отказоустойчивость системы, автоматическое восстановление после отказов и поддержка кластеров из массовых серверов;
- Ориентация на относительно небольшое число крупных файлов, которые записываются однократно;
- Оптимизация под операции записи в конец файла, выполняемые одновременно многими клиентами;
- Нестандартный интерфейс файловой системы и ослабленная модель целостности данных;
- Эффективное использование сетевых ресурсов и оптимизация под высокую агрегированную пропускную способность.

1.2. Hadoop File System

Распределенная файловая система Hadoop File System (HDFS) [5] входит в состав свободно распространяемой платформы распределенных вычислений Hadoop

(см. раздел 2.2). При создании HDFS было использовано много ключевых идей из системы GFS, описанной в предыдущем разделе. Поэтому, по сути, HDFS является общедоступной альтернативой закрытой технологии GFS. Приведенное ниже описание HDFS соответствует текущей на момент написания статьи версии Hadoop 0.17.0.

Перечислим основные тезисы и предположения, повлиявшие на архитектуру и реализацию HDFS. Во многом эти предположения пересекаются с GFS. HDFS спроектирована для запуска на кластерах из массовых комплектующих, обладает высокой отказоустойчивостью и реализует автоматическое восстановление после отказов. HDFS нацелена на поддержку приложений, связанных с обработкой больших объемов данных. Поэтому акцент делается на обеспечении высокой пропускной способности при доступе к данным в потоковом режиме, а не на низкой задержке (латентности). Большие объемы данных подразумевают большие размеры хранимых файлов, измеряемые в гигабайтах и терабайтах, поэтому HDFS оптимизирована для хранения файлов подобного размера. Отказ от следования стандарту POSIX, накладывающему ряд жестких ограничений, которые не требуются для целевых приложений, позволяет повысить производительность системы. HDFS жестко следует модели однократной записи файла с последующим многократным чтением его (write-once-read-many). Данная модель соответствует многим приложениям и упрощает обеспечение целостности данных. Текущая версия HDFS поддерживает только однократную запись в файл одним клиентом. В планах разработчиков предусмотрена реализация операции записи данных в конец файла (append), аналогичной функциональности GFS.

Пространство имен HDFS имеет иерархическую структуру с возможностью создания вложенных директорий. Каждый файл хранится в виде последовательности блоков фиксированного размера, составляющего по умолчанию 64 Мб. Копии блоков (реплики) хранятся на нескольких серверах, по умолчанию - трех. Размер блока и число реплик (т.н. фактор репликации) может задаваться индивидуально для каждого файла.

HDFS имеет очень похожую на GFS архитектуру типа “master-slave”. Главный сервер называется namenode, а подчиненные сервера – datanode. Для каждого из серверов в кластере рекомендуется использовать выделенную машину, хотя при отладке они могут быть запущены в рамках одной машины. Namenode-сервер

управляет пространством имен файловой системы, репликацией и доступом клиентов к данным. Функции datanode-сервера аналогичны функциям chunk-сервера GFS: хранение блоков данных на локальном диске, обслуживание запросов клиентов на чтение и запись данных, а также выполнение команд главного сервера по созданию, удалению и репликации блоков. Передача данных осуществляется только между клиентом и datanode-сервером, минуя namenode-сервер. Каждый datanode-сервер периодически отправляет namenode-серверу сообщения с информацией о своем состоянии и хранимых блоках, т.н. heartbeat-сообщения.

Производительность системы, состоящей из большого числа серверов в нескольких стойках (rack), существенно зависит от выбранной стратегии размещения реплик блока. В большинстве случаев, доступная пропускная способность сети между серверами в одной стойке больше, чем между серверами, находящимися в разных стойках. При размещении реплик блока в различных стойках достигается равномерное распределение реплик по кластеру и максимальная пропускная способность для чтения данных. Однако данная стратегия приводит к снижению производительности операций записи, поскольку данные требуется передать сразу в несколько стоек. В качестве альтернативы в HDFS предлагается стратегия, согласно которой первая реплика размещается на локальной машине, вторая – на другой машине в этой же стойки, а оставшиеся реплики равномерно распределяются по машинам за пределами данной стойки. При чтении данных клиент выбирает ближайшую к нему реплику, например, находящуюся с ним в одной стойке.

Namenode-сервер фиксирует все транзакции, связанные с изменением метаданных файловой системы, в log-файле, называемом EditLog. Примерами подобных транзакций могут служить создание нового файла или изменение фактора репликации у существующего файла. Все метаданные файловой системы, включая отображения файлов в блоки и атрибуты файлов, хранятся в файле FsImage. Файлы EditLog и FsImage хранятся на локальном диске namenode-сервера. Для хранения метаданных файловой системы используется компактное представление, позволяющее загружать их целиком в оперативную память namenode-сервера. При запуске namenode-сервер считывает файлы EditLog и FsImage в оперативную память и применяет все транзакции из log-файла к образу файловой системы, после чего сохраняет новую

версию FsImage на диск и очищает EditLog. Подобная операция проводится пока только при запуске сервера. Во время работы сервера размер log-файла может стать очень большим, поэтому в системе предусмотрен специальный компонент (т.н. secondary namenode), который контролирует размер log-файла и периодически обновляет файл FsImage.

Datanode-сервер хранит каждый блок данных в виде файла на локальном жестком диске. При этом серверу ничего не известно о HDFS-файлах. При запуске datanode-сервер сканирует локальную файловую систему, генерирует список хранимых HDFS-блоков и отправляет эту информацию namenode-серверу.

Клиенты и серверы HDFS взаимодействуют друг с другом по протоколу TCP с использованием механизма Remote Procedure Call (RPC). Важной особенностью, характерной также и для GFS, является то, что namenode-сервер никогда не инициирует RPC-вызовы, а только отвечает на вызовы клиентов и datanode-серверов. Все нужные инструкции datanode-серверам namenode-сервер отправляет, используя ответы на приходящие heartbeat-вызовы. Подобная техника, называемая piggybacking, позволяет уменьшить зависимость главного сервера от состояний подчиненных серверов.

Namenode-сервер контролирует состояние datanode-серверов с помощью heartbeat-сообщений. При прекращении поступления сообщений от datanode-сервера, вследствие отказа самого сервера или нарушения связности сети, данный сервер помечается как отказавший и не учитывается при обработке запросов клиентов. Все данные, хранимые на отказавшем сервере, в результате чего уровень репликации некоторых блоков падает ниже установленного значения. Namenode-сервер автоматически инициирует репликацию данных блоков. Аналогичные действия проводятся в случае сбоя жесткого диска на datanode-сервере, повреждения отдельных реплик или увеличения фактора репликации файла.

Со временем размещение данных в HDFS-кластере может оказаться несбалансированным, снижая производительность системы. Связано это может быть, например, с добавлением новых datanode-серверов или заполнением дисков на части серверов. В HDFS реализован механизм балансировки кластера, осуществляющий перераспределение данных между серверами. Процесс балансировки кластера запускается вручную администратором.

Целостность хранимых данных контролируется с помощью контрольных сумм. При создании файла, HDFS-клиент вычисляет контрольную сумму каждого блока и сохраняет ее в специальном скрытом HDFS-файле в той же директории, что и файл. При чтении данных клиент проверяет их на целостность с помощью соответствующего checksum-файла. В случае если проверка не проходит, клиент может обратиться за данными к другому datanode-серверу.

Данные, хранимые в файлах EditLog и FsImage на namenode-сервере, являются критичными для функционирования HDSF. Повреждение этих может привести к разрушению файловой системы. Поэтому namenode-сервер поддерживает работу с несколькими копиями данных файлов, которые могут быть размещены на других машинах. В этом случае изменения записываются синхронно в каждую из копий файла. Данный механизм снижает производительность namenode-сервера, но позволяет увеличить его отказоустойчивость.

В текущей реализации HDFS главный сервер является “слабым местом” системы. При выходе из строя namenode-сервера система требуется ручное вмешательство, до которого система становится неработоспособной. Автоматический перезапуск namenode-сервера и его миграция на другую машину пока не реализованы.

При записи данных в HDFS используется подход, позволяющий достигнуть высокой пропускной способности. Приложение ведет запись в потоковом режиме, при этом HDFS-клиент кэширует записываемые данные во временном локальном файле. Когда в файле накапливаются данные на один HDFS-блок, клиент обращается к namenode-серверу, который регистрирует новый файл, выделяет блок и возвращает клиенту список datanode-серверов для хранения реплик блока. Клиент начинает передачу данных блока из временного файла первому datanode-серверу из списка. Datanode-сервер получает данные небольшими порциями, сохраняет их на диске и пересылает следующему datanode-серверу в списке. Таким образом, данные передаются в конвейерном режиме и реплицируются на требуемом количестве серверов. По окончании записи, клиент уведомляет namenode-сервер, который фиксирует транзакцию создания файла, после чего он становится доступным в системе.

Механизм удаления файлов в HDFS реализован аналогично GFS. Файл не удаляется из системы мгновенно, а перемещается в специальную директорию /trash. По

истечении настраиваемого периода времени, namenode-сервер удаляет файл из пространства имен HDFS и освобождает связанные с файлом блоки. По умолчанию удаленные файлы хранятся в системе в течение 6 часов, чем объясняется задержка между формальным удалением файла и освобождением дискового пространства.

HDFS реализована на языке Java, что обеспечивает высокую переносимость системы. В то же время, HDFS тесно “привязана” к распространенной на серверах платформе GNU/Linux, что затрудняет запуск системы на ОС Windows (требуется эмулятор shell, например Cygwin).

Для доступа к HDFS из приложений программисты могут использовать прикладной интерфейс программирования (API) на языках Java и C. Также планируется реализовать доступ к HDFS через протокол WebDAV. Пользователям HDFS доступен интерфейс командной строки DFSShell. Для администрирования системы используется набор команд DFSAdmin. HDFS также предоставляет Web-интерфейс, позволяющий пользователям просматривать информацию о системе, структуру файловой системы и содержимое файлов через браузер.

В заключение отметим основные отличия текущей версии HDFS от ее прототипа - GFS:

- Отсутствие автоматического восстановления после отказа главного сервера;
- Отсутствие поддержки записи в файл после его создания одним или несколькими клиентами одновременно, а также операций append и snapshot;
- Реализация на Java, в то время как GFS реализована на C++.

1.3. Google BigTable

Система BigTable является закрытой разработкой компании Google, используемой для хранения структурированных данных многочисленными сервисами и проектами Google. Внутри Google функционирует более 500 экземпляров BigTable (т.н. cells), крупнейший из которых насчитывает более 3 тысяч машин, хранящих свыше 6 петабайт данных [3]. Наиболее загруженные экземпляры BigTable обслуживают в круглосуточном режиме более 500 тысяч запросов в секунду. Описание BigTable было опубликовано в работе [6].

BigTable - распределенная система хранения структурированных данных, позволяющая хранить петабайты данных на тысячах серверов. При создании системы акцент делался на следующих характеристиках: универсальность, масштабируемость, высокая производительность и надежность. Во многом BigTable напоминает базу данных и использует многие стратегии реализации, применяемые в высокопроизводительных СУБД. Однако, как будет ясно из описания, существует ряд принципиальных отличий BigTable от традиционных систем.

В отличие от реляционной модели данных, в BigTable применяется более простая модель разреженной, многомерной, сортированной хэш-таблицы. Каждое значение, хранимое в хэш-таблице, индексируется с помощью ключа строки, ключа столбца и времени: (row:string, column:string, time:int64) → string. Само хранимое значение является байтовым массивом, никак не интерпретируемым системой. Подобную хэш-таблицу можно представить в виде таблицы, каждая строка и столбец которой имеют уникальные ключи. В ячейках таблицы могут содержаться значения, причем у значения может быть несколько версий, с каждой из которых связана временная метка (timestamp). Иными словами, у таблицы есть несколько временных слоев. Выбранная модель данных обусловлена спецификой приложений Google. Например, Web-страницы могут храниться в таблице, ключами строк в которой являются URL страниц, а в ячейках находятся несколько версий содержимого страницы, загруженных роботом в разные моменты времени. Другой особенностью хранимых данных является то, что в разных строках таблиц могут быть заполнены различные группы столбцов. Зачастую таблицы являются разреженными, что повлияло на выбор модели хранения данных, ориентированной на столбцы (column-oriented).

BigTable-кластер хранит несколько таблиц, созданных пользователями. Строки в таблицах хранятся в лексикографическом порядке значений их ключей. Диапазон значений ключей динамически разбивается на несколько частей, называемых таблетами (tablet). Разбиение на таблеты используется для распределения данных по машинам кластера. При создании таблица состоит из одного таблета. С ростом хранимых в таблице данных, она автоматически разбивается на несколько таблетов таким образом, чтобы каждый таблет был размером около 100-200 Мб.

Ключи столбцов также организованы в группы, называемые семействами столбцов (column family). Ключ столбца имеет вид family:qualifier, где family – имя семейства, а qualifier – уникальное имя столбца в рамках семейства. Перед записью данных в таблице должны быть определены используемые семейства, после чего допускается использовать любые имена для столбцов в рамках семейства. Предполагается, что в таблице содержится небольшое число семейств, и их состав редко меняется. В то же время в таблице может быть неограниченное число столбцов. Права доступа к данным определяются на уровне семейств столбцов. Приложениям могут быть даны права на чтение, запись новых данных и изменение семейств.

В каждой ячейке таблицы может храниться несколько версий данных с различными временными метками. Данные метки могут явно назначаться клиентом или автоматически генерироваться BigTable. Версии значения ячейки хранятся в порядке убывания временной метки, то есть самые поздние данные считываются в первую очередь. BigTable поддерживает автоматическое удаление устаревших версий данных с помощью двух настроек, определяемых на уровне семейства: число хранимых последних версий и срок давности данных.

В отличие от реляционных СУБД, BigTable не поддерживает язык SQL. Доступ к BigTable из приложений обеспечивается с помощью клиентской библиотеки, реализующей довольно простой интерфейс прикладного программирования (API). BigTable API содержит операции как для управления таблицами и семействами столбцов, так и для чтения и записи данных. Каждая операция чтения и записи данных в рамках одной строки является атомарной, вне зависимости от того, какие столбцы задействованы. Это позволяет клиентам проще отслеживать поведение системы в присутствии одновременных обновлений одной строки. Также клиенты могут группировать несколько операций в пределах одной строки в атомарную транзакцию. В состав API входит интерфейс для итерации по хранимым значениям с указанием различных фильтров и ограничений.

BigTable-кластер состоит из главного сервера (master) и множества таблет-серверов (tablet server), которые могут динамически добавляться или удаляться из кластера. В обязанности главного сервера входит распределение таблетов по таблет-серверам, отслеживание состояния таблет-серверов, балансировка нагрузки между

среверами и удаление неиспользуемых файлов. Кроме того, главный сервер управляет созданием и изменением таблиц и семейств столбцов. Каждый таблет-сервер предоставляет доступ к выделенному набору таблетов. Сервер обрабатывает запросы на чтение и запись в обслуживаемые таблеты, а также разбивает таблеты, размер которых стал слишком большим. Как и в предыдущих описанных системах, данные передаются между клиентом и таблет-сервером напрямую, минуя главный сервер. В отличие от GFS, клиенты не используют главный сервер для определения местоположения таблетов. Поэтому главный сервер обычно загружен слабо.

Для хранения информации о местоположении таблетов используется трехуровневая иерархическая структура. Первый уровень состоит из файла, содержащего адрес корневого таблета (root tablet). Данный файл размещается на высоконадежном сервисе Chubby, описанном в разделе 3.1. Отметим, что в Chubby также хранятся схемы таблиц, права доступа и другие служебные данные BigTable. В корневом таблете хранятся местоположения всех таблетов в системной таблице METADATA. В свою очередь, в таблетах METADATA хранятся адреса всех таблетов из пользовательских таблиц. Клиенты BigTable кэшируют информацию о местоположении таблетов, а также считывают сразу информацию о нескольких таблетах, что позволяет снизить число обращений к таблице METADATA.

В любой момент времени каждый таблет обслуживается одним таблет-сервером. Главный сервер отслеживает статус таблет-серверов и текущее расположение таблетов. В случае если какой-то из таблетов не назначен ни одному серверу, главный сервер находит таблет-сервер с достаточным свободным пространством и отправляет ему запрос на загрузку данного таблета.

Сами таблеты хранятся в виде файлов в распределенной файловой системе GFS (см. раздел 1.1), что обеспечивает надежное хранение данных. Список GFS-файлов с данными таблета хранится в таблице METADATA. Для хранения данных BigTable используется специальный формат SSTable, позволяющий реализовать эффективный поиск значений по их ключам, а также итерации по значениям, связанным с заданным диапазоном ключей. SSTable состоит из несколько блоков и индекса блоков, хранимого в конце файла. Для быстрого поиска блоков таблет-сервер загружает индекс в оперативную память.

При обслуживании запросов на запись, таблет-сервер заносит все изменения таблетов в log-файл, также хранимый в GFS. При этом последние изменения сохраняются в оперативной памяти в отсортированном буфере, называемом memtable. Операции чтения обслуживаются путем объединения содержимого SSTable-файлов таблета и memcache. При превышении порогового размера, содержимое memtable конвертируется в формат SSTable и сохраняется в GFS. Эта процедура называется малым сжатием (minor compaction). Большое количество файлов SSTable снижает производительность и увеличивает объем хранимых данных. Поэтому периодически проводится процедура сжатия слиянием (merging compaction), когда из нескольких SSTable-файлов и содержимого memtable формируется один SSTable-файл. Случай, когда в сжатии слиянием участвуют все SSTable-файлы таблета, называется большим сжатием (mahjor compaction). После большого сжатия в SSTable не остается удаленных записей. Во время загрузки таблета, таблет-сервер считывает из GFS в оперативную память индексы SSTable и log-файл с последними изменениями. Далее сервер восстанавливает содержимое буфера memcache путем применения всех изменений со времени последнего малого сжатия.

Для обнаружения и отслеживания статуса таблет-серверов в BigTable используется сервис Chubby (см. раздел 3.1). Главный сервер и таблет-сервера поддерживают периодически продлеваемые Chubby-сессии. Каждый таблет-сервер при запуске создает в выделенной директории Chubby новый файл и получает блокировку на него. Главный сервер определяет появление новых таблет-серверов по созданию новых файлов в данной директории. Когда таблет-сервер теряет блокировку на созданный им файл, он прекращает обслуживание клиентов и пытается заново получить блокировку. В случае если файл больше не существует, таблет-сервер прекращает свою работу. Главный сервер периодически запрашивает у таблет-сервера статус его блокировки. Если сервер теряет блокировку или становится недоступным, то главный сервер пытается получить блокировку на файл сервера. Если главный сервер получает блокировку, то он удаляет файл сервера, тем самым, гарантируя, что сервер больше не будет обслуживать клиентов. После чего главный сервер переносит таблеты, обслуживавшиеся сервером, в список неназначенных таблетов. В случае потери

Chubby-сессии, главный сервер прекращает свою работу, поскольку надежное функционирование системы без доступа к Chubby невозможно.

Используемая Google система управления кластером производит мониторинг и автоматический перезапуск главного сервера. При запуске главный сервер производит следующие действия. Сервер получает блокировку на файл в Chubby, выделенный для главного сервера BigTable. Таким образом, гарантируется, что в системе только один главный сервер. Далее сервер сканирует директорию с файлами таблет-серверов в Chubby и связывается с каждым из серверов, чтобы определить список обслуживаемых ими таблетов. Затем главный сервер сканирует таблицу METADATA для получения списка таблетов. В случае если обнаружен таблет, не обслуживаемый ни одним таблет-сервером, он заносится в список неназначенных таблетов.

Набор хранимых в системе таблетов может изменяться в результате операций создания и удаления таблиц, слияния двух таблетов или разбиения таблета на части. Главный сервер управляет всеми перечисленными операциями, за исключением последней. Разбиение таблета инициируется таблет-сервером и фиксируется путем записи информации в таблицу METADATA, после чего таблет-сервер уведомляет об изменениях главный сервер.

В реализации BigTable используется ряд дополнительных механизмов, нацеленных на достижение высокой производительности, надежности и доступности системы.

Клиенты могут объединять несколько семейств столбцов в так называемые группы локализации (locality group). Для каждой группы в каждом из таблетов создается отдельная структура SSTable. Размещение семейств столбцов, обычно не считываемых совместно, в различных группам позволяет повысить эффективность операций чтения. Кроме того, на уровне групп локализации может задаваться ряд дополнительных параметров. Например, группа может быть объявлена как размещаемая в памяти. В этом случае данные, относящиеся к группе, кэшируются в оперативной памяти таблет-сервера.

BigTable поддерживает сжатие хранимых данных, параметры которого также определяются на уровне группы локализации. Указанный пользователем формат сжатия применяется к каждому блоку SSTable. Поскольку хранимые в группе данные

обычно относятся к одному типу, и ключ строки может быть выбран таким образом, чтобы похожие данные находились рядом, то на практике часто достигается высокая степень сжатия данных.

Для повышения производительности операций чтения в таблет-серверах применяется двухуровневая схема кэширования: на верхнем уровне кэшируются значения из ячеек таблицы, а на нижнем – блоки SSTable.

При чтении данных требуется обращение ко всем SSTable-файлам таблета. Для уменьшения числа дорогостоящих операций чтения с диска, BigTable позволяет задать на уровне группы локализации Bloom-фильтры, создаваемые для SSTable-файлов. Применение Bloom-фильтров позволяет определять, есть ли искомые данные в SSTable, не обращаясь к диску.

В заключение перечислим основные отличительные особенности BigTable в сравнении с реляционными СУБД:

- Ориентация на хранение больших объемов слабоструктурированных, разреженных данных;
- Высокая производительность, масштабируемость и устойчивость;
- Модель хранения, ориентированная на столбцы, аналогичная column-oriented базам данных (C-Store, Vertica, Sybase IQ, SenSage, KDB+, MonetDB/X100);
- Нет поддержки реляционной модели, типов данных у столбцов, языка SQL, транзакций и т.д.

1.4. HBase

Система HBase [7] входит в состав свободно распространяемой платформы распределенных вычислений Hadoop (см. раздел 2.2). HBase является общедоступным аналогом описанной выше системы BigTable, работающим поверх распределенной файловой системы HDFS. Приведенное ниже описание HBase соответствует текущей на момент написания статьи версии HBase 0.1.2.

Поскольку модель данных, архитектура и реализация HBase очень близка к BigTable, остановимся только на характерных особенностях и отличиях HBase от BigTable.

HBase использует несколько иную терминологию, чем BigTable. Так, аналог таблетов называется регионом (region), а обслуживающие регионы серверы называются регион-сервером (RegionServer). Внутри регионов данные разбиты на вертикальные семейства столбцов. Каждое семейство внутри региона хранится в отдельной структуре данных, называемой Store и аналогичной SSTable в BigTable. Содержимое Store хранится в нескольких файлах в HDFS, называемых StoreFile. Регион-сервер кэширует последние изменения в оперативной памяти (MemCache) и периодически сохраняет их в новые Store-файлы. Аналогично процедурам сжатия в BigTable, регион-сервер периодически объединяет Store-файлы. В отличие от BigTable, HBase не поддерживает определение прав доступа для семейств столбцов.

Главный сервер HBase (Master) управляет назначением регионов серверам и отслеживает их состояние. В отличие от BigTable, HBase не использует подобный Chubby высоконадежный сервис для координации серверов. В случае, если регион-сервер теряет связь с главным сервером, то он автоматически завершает работу и перезапускается. При этом главный сервер перераспределяет обслуживавшиеся данным сервером регионы между другими регион-серверами. В отличие от этого, таблет-сервер BigTable может продолжать обслуживать клиентов после потери соединения с главным сервером. При перезапуске главного сервера, регион-сервера подключаются к нему и передают список обслуживаемых ими регионов.

Метаданные и местоположения регионов хранятся в таблице META. В свою очередь, местоположения всех регионов таблицы META хранятся в таблице ROOT, занимающей всегда один регион. Клиенты запрашивают местоположение ROOT-региона у главного сервера, после чего просматривают таблицу META путем взаимодействия с регион-серверами. Местоположения регионов кэшируются клиентом.

Как остальные компоненты, входящие в состав платформы Hadoop, HBase реализована на языке Java. Система имеет несколько интерфейсов для клиентских приложений: Java API, REST-интерфейс и доступ по протоколу Thrift. Пользователи могут взаимодействовать с системой через командную оболочку HBase Shell, которая поддерживает SQL-подобный язык HQL. HBase также предоставляет Web-интерфейс, позволяющий пользователям просматривать информацию о системе и хранимых таблицах.

2. Модели программирования и технологии распределенной обработки данных

В главе рассматриваются модели программирования и технологии, ориентированные на обработку больших массивов данных на распределенных кластерных системах. Центральное место в главе занимает описание модели программирования MapReduce, разработанной в компании Google, и ее открытой реализации Apache Hadoop. В качестве другого подхода к описанию и реализации процессов обработки данных рассматривается технология Microsoft Dryad.

2.1. Модель программирования MapReduce

MapReduce [2] – модель программирования и платформа для пакетной обработки больших объемов данных, разработанная и используемая внутри компании Google для широкого круга приложений. Модель MapeReduce отличается простотой и удобством использования, скрывая от пользователя детали организации вычислений в ненадежной распределенной среде. Пользователю достаточно описать процедуру обработки данных в виде двух функций – map и reduce, после чего система автоматически распределяет вычисления по кластеру из большого количества машин, обрабатывает отказы машин, балансирует нагрузку и координирует взаимодействия между машинами для эффективного использования сетевых и дисковых ресурсов. Впервые описание MapReduce было опубликовано в работе [8]. За последние четыре года внутри Google было разработано более 10 тысяч программ для MapReduce. В среднем, каждый день на кластерах Google выполняется около тысячи MapReduce-заданий, обрабатывающих вместе более 20 петабайтов данных [2]. Используемая в Google реализация MapReduce является закрытой технологией, однако существует общедоступная реализация Apache Hadoop (см. раздел 2.2).

В рамках MapReduce вычисления принимают на вход и производят на выходе данные, состоящие из множества пар “ключ-значение”. Обозначим входные данные как list(k1,v1), а выходные – как list(k2,v2). Пользователь описывает вычисления в виде

двух функций - map и reduce, близких по смыслу одноименным функциям языка Lisp. Функция ***map(k1,v1)→list(k2,v2)*** применяется к каждой паре входных данных и возвращает набор промежуточных пар. Далее реализация MapReduce группирует промежуточные значения v2, связанные с одним ключом k2, и передает эти значения функции reduce. Функция ***reduce(k2,list(v2))→list(v2)*** преобразует промежуточные значения в окончательный набор значений для данного ключа. Как правило, это одно агрегированное значение, например, сумма.

В качестве примера рассмотрим задачу подсчета числа вхождения каждого слова в большую коллекцию документов. Входные данные можно представить в виде пар <имя документа, содержимое>. Тогда функция map для каждого слова, найденного в содержимом документа, должна возвратить пару <слово, 1>, а функция reduce должна просуммировать все значения для каждого слова, возвратив пары <слово, число вхождений>. Ниже приведен пример реализации данных функций на псевдокоде:

```
map(String key, String value):
   // key: имя документа
   // value: содержимое документа
   for each word w in value:
      EmitIntermediate(w, “1”);

reduce(String key, Iterator values):
   // key: слово
   // values: список отметок о вхождении слова
   int result = 0;
   for each v in values:
      result += ParseInt(v);
   Emit(AsString(result));
```

Обратим внимание на использование итератора в приведенном коде функции reduce. Обусловлено это тем, что обрабатываемые функцией наборы данных могут быть слишком большими для размещения в памяти.

Помимо описания функций map и reduce, в спецификацию MapReduce-задания входят пути к входным и выходным файлам, размещаемым в GFS, а также дополнительные конфигурационные параметры. Пользователь формирует объект со спецификацией задания при помощи интерфейса прикладного программирования на языке C++, предоставляемого библиотекой MapReduce. Запуск задания производится с помощью вызова специальной функции *MapReduce*, которой передается спецификация задания.

Реализация MapReduce в Google ориентирована на вычислительную инфраструктуру, состоящую из большого числа недорогих серверов из массовых комплектующих. Как правило, это двухпроцессорные x86-машины с 4-8 Гб оперативной памяти, работающие под управлением Linux. Для соединения машин в кластере используется коммутируемый Gigabit Ethernet. Кластеры состоят из тысяч машин, поэтому постоянно возникают отказы отдельных узлов. Для хранения данных используются недорогие IDE-диски, подключенные к каждой из машин. Данные хранятся под управлением распределенной файловой системы GFS (см. раздел 1.1). Напомним, что GFS использует репликацию для надежного хранения данных. Запуском MapReduce-заданий на кластере управляет планировщик, который отслеживает состояние машин и подбирает группу машин для выполнения задания.

Вызовы функции map распределяются между несколькими машинами путем автоматического разбиения входных данных на M частей, размер каждой из которых составляет обычно 16-64 Мб. Полученные порции данных могут обрабатываться параллельно различными машинами. Вызовы reduce распределяются путем разбиения пространства промежуточных ключей на R частей, определяемых с помощью функции разбиения (partitioning function). По умолчанию используется функция *hash(k2)* **mod** *R*. Число частей R и функция разбиения указываются пользователем в спецификации задания.

Рассмотрим детально процесс выполнения задания в MapReduce.

Запуск задания инициируется блокирующим вызовом внутри кода пользователя библиотеки MapReduce. После чего система разбивает входные файлы на M частей, размер которых контролируется с помощью конфигурационного параметра. Далее MapReduce осуществляет запуск копий программы на машинах кластера. Одна из

копий программы играет роль управляющего (master), а остальные являются рабочими (worker). Управляющий процесс осуществляет распределение задач (M map-задач и R reduce-задач) между рабочими. Каждому свободному рабочему процессу назначается map- или reduce-задача. Для каждой из задач управляющий процесс хранит ее состояние (ожидает выполнения, выполняется, завершена) и идентификатор рабочего процесса (для выполняющихся и выполненных заданий).

Рабочий процесс, выполняющий map-задачу, считывает содержимое соответствующего фрагмента входных данных из GFS, выделяет из данных пары ключ-значение и передает каждую из пар заданной пользователем функции map. Полученные промежуточные пары буферизуются в памяти и периодически записываются на локальный диск. При записи данные разбиваются на R частей, в соответствии с функцией разбиения. По завершении задачи, местоположения файлов с промежуточными данными передаются управляющему процессу. Для каждой из map-задач управляющий процесс хранит местоположения и размеры R фрагментов с промежуточными данными, полученными задачей. Данная информация периодически обновляется по мере выполнения map-заданий и передается рабочим процессам, выполняющим reduce-задачи.

При получении от управляющего процесса информации о готовых промежуточных данных, reduce-процесс считывает данные с локального диска map-процесса при помощи RPC-вызовов. Когда reduce-процесс получил все промежуточные данные для его порции выходных ключей, он производит сортировку значений по ключам. В случае если объем промежуточных данных слишком велик для размещения в памяти, используется внешняя сортировка. После сортировки, reduce-процесс последовательно сканирует промежуточные данные и для каждого встреченного уникального ключа вызывает заданную пользователем функцию reduce, передавая ей ключ и список найденных значений. Результаты вызова reduce записываются в выходной файл в GFS. Данные внутри выходного файла отсортированы по значению ключа. Для каждой reduce-задачи создается отдельный выходной файл.

Когда все map- и reduce-задачи выполнены, управляющий процесс завершает выполнение вызова MapReduce в программе пользователя и передает управление следующему за ней коду. Результаты выполнения задания доступны в виде R файлов.

Обычно пользователь не объединяет полученные данные в один файл, а передает на вход следующему MappReduce-заданию или другому приложению, которое поддерживает работу с несколькими входными файлами.

Рассмотрим обеспечение отказоустойчивости при выполнении задач MapReduce.

Управляющий процесс периодически опрашивает рабочие процессы. В случае если в течение определенного времени ответ от рабочего процесса не поступил, управляющий процесс помечает рабочий процесс как отказавший. Все map-задачи, выполненные данным процессом переводятся в состояние “ожидает выполнения” и запускаются повторно на других машинах. Аналогично, все выполняемые отказавшим процессом map- и reduce-задачи переводятся в состояние “ожидает выполнения” и запускаются повторно. Перезапуск map-задач необходим, потому что их результаты хранятся на локальном диске отказавшей машины и, следовательно, становятся недоступны. Перезапуск reduce-задач не требуется, поскольку их результаты сохраняются в GFS. В случае повторного запуска уже выполненного map-задания, все reduce-процессы уведомляются об этом. Те процессы, которые не успели считать результаты предыдущего запуска задачи, будут считывать их у нового map-процесса.

В случае если заданные пользователем функции map и reduce являются детерминированными функциями своих входных значений, распределенная реализация MapReduce гарантирует получение такого же результата, что и при последовательном выполнении программы без ошибок. Для этого используется атомарная фиксация результатов map- и reduce-задач. По выполнении map-задачи, рабочий процесс отправляет управляющему процессу сообщение, в котором указывает список R промежуточных файлов. В случае, если управляющий процесс получает сообщение о завершении уже выполненной задачи, он игнорирует данное сообщение. По завершении reduce-задачи, рабочий процесс атомарно переименовывает средствами GFS временный выходной файл в окончательный выходной файл. В случае если reduce-задача выполняется на нескольких машинах, то только результат одной из них будет в итоге записан в окончательный выходной файл.

В случае если функции map и/или reduce являются недетерминированными, система предоставляет более слабые гарантии. Утверждается, что результат определенной reduce-задачи R1 будет эквивалентен результату R1, полученному при

некотором запуске последовательной недетерминированной программы. При этом результат другой reduce-задачи R2 может соответствовать результату R2, полученному при другом запуске последовательной недетерминированной программы. Подобная ситуация может возникать из-за того, что задачи R1 и R2 использовали результаты двух различных запусков некоторой map-задачи.

Количество map- и reduce-задач обычно подбирается таким образом, что оно было гораздо больше числа рабочих машин. В этом случае достигается лучшая балансировка нагрузки между машинами. Кроме того, это позволяет эффективнее обрабатывать отказ map-процесса, поскольку большое число выполненных им задач может быть распределено по всем рабочим машинам. На количество map-заданий также влияет размер получаемых фрагментов входных данных. Идеальным является размер 16-64 Мб, не превышающий размер блока GFS и позволяющий локализовать вычисления рядом с данными. Количество reduce-заданий обычно ограничивается пользователями допустимым числом выходных файлов.

Рассмотрим некоторые оптимизации, примененные при реализации MapReduce.

При распределении map-задач по машинам в кластере управляющий процесс учитывает то, каким образом входные данные размещены в GFS (см. раздел 1.1). Поскольку данные хранятся в виде нескольких реплик на тех же машинах кластера, то управляющий сервер пытается отправить map-задачу на машину, хранящую соответствующий фрагмент входных данных, или же на машину, наиболее близкую к данным в смысле сетевой топологии. Подобная стратегия позволяет существенно снизить объем данных, передаваемых по сети во время запуска задания, и, тем самым, уменьшить время выполнения задания.

Для уменьшения промежуточных данных передаваемых по сети от map-процессов к reduce-процессам, внутри map-задачи может производиться локальная редукция промежуточных данных с одним значением ключа. Для этого, пользователь должен указать в спецификации задания так называемую combiner-функцию, которая имеет такую же семантику, что функция reduce. Часто в качестве combiner-функции указывается reduce.

Одной из частых причин увеличения времени выполнения MapReduce-задания является появление “отстающего” процесса, который слишком долго выполняет одну

из последних map- или reduce-задач. Подобное поведение может быть обусловлено многими причинами, например, неисправностью жесткого диска или запуском на машине других вычислений. Для устранения подобной проблемы в MapReduce используется следующая стратегия. В конце вычислений, управляющий процесс запускает выполняющиеся задачи на дополнительных машинах. При выполнении задачи на одной из машин, ее выполнение на другой машине приостанавливается. Подобная стратегия значительно уменьшает время выполнения больших заданий.

Для отладки MapReduce-программ предусмотрен последовательный режим выполнения задания на локальной машине. Во время выполнения задания на кластере, управляющий процесс предоставляет Web-интерфейс с подробной информацией о статусе выполнения задания, доступом к log-файлам и т.д.

В заключение, рассмотрим отличия MapReduce от существующих моделей и систем параллельных вычислений. Модели параллельных вычислений с элементами функционального программирования, позволяющие пользователю формировать программу из примитивов типа map, reduce, scan, sort и т.д., предлагались в академической среде задолго до появления MapReduce (см., например [9]). С этой точки зрения MapReduce можно рассматривать как упрощенную квинтэссенцию данных моделей, ориентированную на решение определенного круга задач по обработке больших массивов данных. Пожалуй, главная заслуга создателей MapReduce заключается в отказоустойчивой реализации вычислений на большом количестве ненадежных машин. В отличие от MapReduce, большинство систем параллельной обработки данных были реализованы на кластерах меньшего масштаба и часто требуют от программиста ручной обработки возникающих отказов. Модель MapReduce накладывает ряд ограничений на программу для того, чтобы автоматизировать распараллеливание, запуск и управление вычислениями на кластере. С одной стороны, это значительно упрощает задачу программиста и практически не требует от него специальной квалификации. С другой стороны, накладываемые системой ограничения не позволяют реализовать в ней решение произвольных задач. Например, в рамках описанной модели нельзя простым образом реализовать операции типа JOIN и SPLIT или организовать взаимодействие между параллельными процессами так, как это делается в технологии MPI.

2.2. Платформа Apache Hadoop

Платформа распределенных вычислений Hadoop [10] разрабатывается на принципах open source в рамках организации The Apache Software Foundation. Платформа ориентирована на поддержку обработки больших объемов данных и заимствует многие идеи у закрытых технологий Google, таких как MapReduce, GFS и BigTable (см. разделы 2.1, 1.1, 1.3). Hadoop состоит из двух частей: Hadoop Core и HBase. В состав Hadoop Core входят распределенная файловая система HDFS (см. раздел 1.2) и реализация модели MapReduce. HBase содержит реализацию распределенной системы хранения структурированный данных (см. раздел 1.4). Данный раздел посвящен особенностям реализации модели MapReduce в Hadoop. Приведенное описание соответствует текущей на момент написания статьи версии Hadoop 0.17.0.

Для реализации вычислений в Hadoop используется архитектура “master-worker”. В отличие от Google MapReduce, в системе есть один выделенный управляющий процесс (т.н. JobTracker) и множество рабочих процессов (т.н. TaskTracker), которые осуществляют выполнение всех заданий пользователей. JobTracker принимает задания от приложений, разбивает их на map- и reduce-задачи, распределяет задачи по рабочим процессам, отслеживает выполнение и осуществляет перезапуск задач. TaskTracker запрашивает задачи у управляющего процесса, загружает код и выполняет запуск задачи, уведомляет управляющий процесс о состоянии выполнения задачи и предоставляет доступ к промежуточным данным map-задач. Процессы взаимодействуют с помощью RPC-вызовов, причем все вызовы идут в направлении от рабочего к управляющему процессу, что позволяет уменьшить его зависимость от состояния рабочих процессов.

Система реализована на языке Java. Для создания приложений используется прикладной интерфейс программирования на Java. Функции map и reduce описываются в виде классов, реализующих стандартные интерфейсы Mapper и Reducer. Спецификация задания оформляется в виде объекта типа JobConf, содержащего методы

для указания классов с map и reduce, форматов входных и выходных данных, путей к входных и выходным данным в HDFS и других параметров задания.

Отметим, что, в отличие от Google MapReduce, реализация функции reduce может возвращать пары с произвольными ключами, не совпадающими с переданным на вход функции промежуточным ключом. Аналогично Google MapReduce, пользователь может указать функции разбиения входных данных (Partitioner) и комбинирования промежуточных данных (Combiner), оформленные в виде Java-классов. В состав Hadoop входят часто используемые на практике реализации функций map и reduce, а также функции разбиения.

Запуск задания осуществляется с помощью вызова функции runJob или submitJob, каждой из которых передается объект JobConf. В первом случае приложение блокируется до завершения заданий, а во втором случае вызов сразу возвращает управление коду приложения. При запуске задания его спецификация и jar-файл с кодом автоматически размещаются в HDFS, после чего задание направляется JobTracker-процессу. В описании задания пользователь может указать набор дополнительных файлов, копируемых на рабочие узлы перед запуском вычислений.

Количество map-задач определяется системой автоматически, исходя из указанного пользователем желаемого числа задач, а также максимального (размер HDFS-блока) и минимального размеров фрагмента входных данных. В общем случае, количество map-задач может не совпадать с указанным пользователем. Количество reduce-задач управляется с помощью параметра задания. По умолчанию используется значение 1. Пользователь может установить число reduce-задач равным 0. В этом случае фаза reduce не проводится, и промежуточные результаты map-задач записываются в выходные файлы в HDFS. Данная возможность полезна в тех случаях, когда не требуется агрегация или сортировка результатов фазы map.

Hadoop поддерживает различные форматы входных и выходных данных, включая текстовый файл, двоичный формат со сжатием и таблицы HBase. Пользователь может использовать другие форматы данных путем создания специальных Java-классов для чтения и записи данных.

Для отладки и отслеживания статуса выполнения map- и reduce-задач Hadoop позволяет обновлять внутри функций map и reduce строку статуса задачи и значения

счетчиков, определенных пользователем. Статус задачи и значения счетчиков отображаются в Web-интерфейсе Hadoop.

В заключение отметим, что помимо запуска Java-программ, Hadoop позволяет указать в качестве реализаций map и reduce произвольные программы. Данные программы должны считывать входные данные из потока ввода и записывать результаты в поток вывода.

2.3. Технология Microsoft Dryad

Универсальная технология распределенной обработки данных Dryad [11] разрабатывается компанией Microsoft. В настоящее время эта технология является закрытой и применяется только внутри компании, например, в поисковой системе MSN Live Search. Описание технологии было опубликовано в работе [12].

Модель программирования Dryad основана на представлении приложения в виде ориентированного ациклического графа. Вершинами графа являются процессы. Ребра графа определяют потоки данных между процессами в виде односторонних каналов. У процесса может быть несколько входных и выходных каналов. Вершины графа могут быть сгруппированы в стадии (stage). Важно отметить, что модель программирования Dryad содержит в себе в качестве частных случаев реляционную алгебру и MapReduce.

Система организует выполнение приложения на имеющихся вычислительных ресурсах, будь то многоядерная машина или кластер из большого числа машин. При этом система автоматически осуществляет планирование вычислений, распределение вершин между машинами, обработку отказов и динамическую оптимизацию структуры графа.

В качестве вершин графа могут выступать программы на C++ или другом языке. Каналы между вершинами реализуются несколькими способами: файлы в распределенной и сетевой файловой системе, TCP-каналы, FIFO-очереди в оперативной памяти. Выбор того или иного варианта реализации каждого канала в графе может производиться системой автоматически, исходя из текущей конфигурации системы и размещения процессов по машинам.

Пользователь описывает граф приложения с помощь интерфейса прикладного программирования на языке C++. При запуске задания на кластере создается менеджер задания (job manager), которые управляет выполнением графа задания. Менеджер задания получает информацию о ресурсах кластера, генерирует граф задания, инициализирует его вершины, пересылает их код на узлы кластера и контролирует выполнение вершин. На узлах кластера запущены процессы, запускающие код вершин графа и позволяющие менеджеру задания отслеживать состояние выполнения вершин. При размещении вершин на узлах кластера система учитывает потоки данных между вершинами и старается разместить взаимодействующие вершины на одной машине или рядом друг с другом.

Наиболее интересной особенностью Dryad является поддержка динамической модификации структуры графа задания во время его выполнения. Приведем несколько примеров. Система обнаруживает вершины, выполняемые медленнее других вершин данной стадии, и автоматически создает дублирующие вершины (аналогично backup-заданиям в MapReduce). Вершина, выполняющая агрегацию данных из множества вершин, может быть снабжена вспомогательными вершинами, которые производят локальную агрегацию данных из вершин в пределах серверной стойки и т.п. (аналогично combine-функции в MapReduce). Вершина может быть заменена на несколько вершин путем разбиения обрабатываемых вершиной данных. Для равномерного разбиения входных данных по значениям их ключей система создает вспомогательные вершины, которые определяют распределение данных по ключам и разбивают данные на равные части. Пользователь также может реализовывать собственные стратегии динамической модификации графа.

Возможности Dryad интегрированы в высокоуровневые языки и системы, такие как DryadLINQ и Microsoft SQL Server. В настоящее время не существует общедоступных аналогов Dryad, таких как Hadoop для MapReduce.

3. Инфраструктурные сервисы

В крупномасштабных распределенных системах, состоящих из ненадежных вычислительных машин, остро стоят вопросы обеспечения отказоустойчивости и бесперебойной работы функционирующих сервисов. Для решения подобных задач могут применяться высоконадежные реплицируемые сервисы координации распределенных процессов. В качестве примеров реализации подобных сервисов рассматриваются сервисы Chubby и Zookeeper.

3.1. Chubby

Из соображений отказоустойчивости и масштабируемости, описанные выше технологии Google спроектированы как распределенные системы, компоненты которых слабо связаны друг с другом (loosely-coupled). Это означает, что компоненты системы должны динамически обнаруживать и отслеживать состояние друг друга, автоматически выбирать в случае отказа новый главный сервер и гибким образом координировать свои действия. Возложение подобных функций на главный сервер делает его уязвимым местом системы. С другой стороны, реализация полностью децентрализованных механизмов в присутствии большого количества машин может оказаться сложной и неэффективной в сравнении с централизованными решениями.

Для решения этой проблемы в компании Google был создан отдельный высоконадежный сервис Chubby [13], используемый такими системами, как GFS, BigTable и MapReduce. Наличие готового сервиса координации упрощает создание сложных распределенных систем. Внутренние системы Google используют Chubby для обнаружения серверов, выбора главного сервера и хранения важных данных. По сути, GFS и BigTable используют Chubby в качестве корневого сервиса для своих распределенных структур данных.

С точки зрения клиентов Chubby выглядит как централизованный сервис. Высокая надежность сервиса обеспечивается за счет репликации его на пяти машинах и использования децентрализованного механизма выборов главной реплики. Сервис доступен до тех пор, пока большинство реплик функционируют и могут

взаимодействовать друг с другом. Выбор главного сервера среди множества узлов является частным случаем задачи о консенсусе в распределенной системе. Для выбора главной реплики в Chubby используется известный алгоритм Paxos [14].

Chubby предоставляет клиентам интерфейс, напоминающий файловую систему с иерархическим пространством имен. В отличие от обычных файловых систем, акцент делается на высокой доступности и надежности, а не на хранимом объеме данных и пропускной способности. Сервис ориентирован на хранение большого числа небольших файлов и поддержку большого числа клиентов. При этом большую часть запросов составляют операции чтения.

Обслуживаемые Chubby файлы и директории образуют узлы файловой системы. Клиенты могут создавать произвольные узлы и записывать в них данные. Кроме того, узлы могут использоваться в качестве блокировки (lock). Блокировка может быть исключающей (exclusive) или совместной (shared). Также существует специальный тип временных узлов, которые автоматически удаляются в том случае, если они не открыты ни одним клиентом (или пусты, в случае директорий).

Каждый клиент Chubby поддерживает периодически продлеваемую сессию. В случае если клиент не продлил свою сессию в течение определенного времени, Chubby автоматически снимает удерживаемые клиентом блокировки и удаляет связанные с клиентом временные файлы. Этот механизм может использоваться для отслеживания статусов клиентов. В свою очередь, Chubby периодически отправляет клиентам уведомления о событиях, связанных с открытыми клиентом узлами: создании, удалении и модификации узлов, изменении содержимого или статуса блокировки узла и т.п.

Приведем несколько примеров использования Chubby для координации компонентов распределенной системы. Для определения главного сервера в системе может использоваться исключительная блокировка выделенного файла в Chubby. Первый сервер, получивший блокировку файла, считается главным, после чего он записывает свой адрес в данный файл. Другие сервера отслеживают содержимое файла для определения текущего главного сервера и статус блокировки файла для обнаружения отказа главного сервера. Для получения главным сервером списка подчиненных серверов может использоваться выделенная директория, в который

каждый сервер создает временный файл со своим адресом. При отказе сервера его сессия в Chubby истекает, и файл автоматически удаляется. Подобные события могут отслеживаться главным сервером системы с помощью механизма уведомлений Chubby. Как было упомянуто в разделе 1.3, система BigTable хранит в Chubby различные метаданные и местоположение корневого таблета. Сервис также активно используется в качестве альтернативы сервису разрешения имен DNS.

В заключение, отметим что, как и другие описанные ранее технологии Google, сервис Chubby является закрытой разработкой, используемой только внутри компании.

3.2. ZooKeeper

Общедоступным аналогом описанной выше технологии Chubby является сервис ZooKeeper [15], разработка которого ведется сотрудниками компании Yahoo на принципах open source. Опустим описание архитектуры и принципы реализации ZooKeeper, поскольку они практически совпадают с Chubby. Отметим, что ZooKeeper может быть применен в сочетании с платформой Hadoop для повышения надежности системы и координации отдельных серверов. Например, с помощью ZooKeeper можно произвести динамическую конфигурацию Hadoop во время запуска системы по требованию на вычислительном кластере. В этом случае клиент и развернутые на кластере TaskTracker-процессы могут определить через ZooKeeper адрес развернутого JobTracker-процесса.

Заключение

В заключении остановимся на новизне описанных технологий, сферах их применения, а также дальнейших перспективах их развития.

С точки зрения алгоритмов и методов в описанных технологиях мало нового. Во многом они опираются на наработки предшественников, сочетая их и, в некоторых случаях, упрощая. Главной заслугой этих систем является выход на принципиально новые масштабы обрабатываемых данных и самих вычислительных инфраструктур. Для этого потребовалось радикально пересмотреть использовавшиеся ранее предположения об архитектуре и принципах функционирования подобных систем. Например, в описанных системах отказ является штатной ситуацией, которая обнаруживается и обрабатывается в автоматическом режиме.

В результате неизбежных компромиссов системы часто жертвуют привычной функциональностью, такой как строгие гарантии целостности данных, поддержка реляционной модели или реализация стандартных интерфейсов файловой системы. Модель программирования MapReduce жертвует гибкостью и универсальностью ради поддержки автоматического управления вычислениями в распределенной среде.

Безусловно, круг приложений MapReduce ограничен параллельными по данным задачами, где не требуется организовывать сложное взаимодействие между процессами. Но, как показывает практика, таких приложений очень много, и тот уровень удобства, на который MapReduce поднимает программирование подобных вычислений, заслуживает внимания. Следующим шагом в данном направлении является технология Microsoft Dryad, более универсальная и мощная, чем MapReduce.

Несмотря на то, что оригинальные реализации технологий являются закрытыми разработками, благодаря open source проектам активно развиваются их общедоступные аналоги. Хочется отметить, что данные технологии, пришедшие из индустрии, начинают активно использоваться в академической среде. Как упоминалось во введении, стоящие перед исследователями вычислительные задачи часто имеют такие же требования, что и задачи Google. Представляется, что подобные технологии в ближайшее время станут неотъемлемой частью современных информационных систем,

в которых все чаще возникает потребность в хранении и анализе больших объемов информации.

Литература

[1] Barroso, L. A., Dean, J., and Urs Hölzle, U. Web search for a planet: The Google cluster architecture. IEEE Micro 23, 2, pp. 22-28, 2003.

[2] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. Commun. ACM, vol. 51, no. 1, pp. 107-113, January 2008.

[3] J. Dean. Handling Large Datasets at Google: Current Systems and Future Directions. Data-Intensive Computing Symposium, March 2008. http://research.yahoo.com/files/6DeanGoogle.pdf

[4] Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google file system. In 19th Symposium on Operating Systems Principles, Lake George, NY, pp. 29-43, 2003.

[5] The Hadoop Distributed File System: Architecture and Design. http://hadoop.apache.org/core/docs/current/hdfs_design.html

[6] Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In OSDI'06: Seventh Symposium on Operating System Design and Implementation, Seattle, WA, USA, November 2006, pp. 205-218.

[7] HBase. http://hadoop.apache.org/hbase/

[8] Dean, J. and Ghemawat, S. MapReduce: Simplified data processing on large clusters. In Proceedings of Operating Systems Design and Implementation (OSDI). San Francisco, CA. 137-150, 2004.

[9] M. Cole. Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press & Pitman, ISBN 0-262-53086-4, 1989.

[10] Apache Hadoop. http://hadoop.apache.org/

[11] Microsoft Dryad. http://research.microsoft.com/research/sv/dryad/

[12] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. European Conference on Computer Systems (EuroSys), Lisbon, Portugal, March 21-23, 2007.

[13] Burrows, M. The Chubby lock service for loosely-coupled distributed systems. In OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation. Berkeley, CA, USA: USENIX Association, 2006, pp. 335-350

[14] Lamport, L. The part-time parliament. ACM Trans. Comput. Syst., vol. 16, no. 2, pp. 133-169, May 1998.

[15] ZooKeeper. http://zookeeper.sourceforge.net/