Введение в облачные и распределенные информационные системы

Тимур Машнин

Тимур Машнин

Введение в облачные и распределенные информационные системы

http://www.litres.ru/pages/biblio_book/?art=63619781
ISBN 9785005303110

Аннотация

Облачные и распределенные вычислительные системы – это быстро развивающаяся IT-область хранения и обработки данных.Современные облачные и распределенные вычислительные системы строятся на основе общих концепций и алгоритмов, таких как облако, MapReduce, NoSQL базы данных, распределенные алгоритмы, масштабируемость и многое другое.Познакомьтесь с этими фундаментальными понятиями облачных и распределенных информационных систем и узнайте, как эти системы работают изнутри.

Содержание

Введение в облачные и распределенные информационные системы

Тимур Машнин

ISBN 978-5-0053-0311-0
Создано в интеллектуальной издательской системе Ridero

Введение

Облачные и распределенные системы

Введение

Облачные и распределенные вычислительные системы – это быстро развивающаяся IT-область хранения и обработки данных.

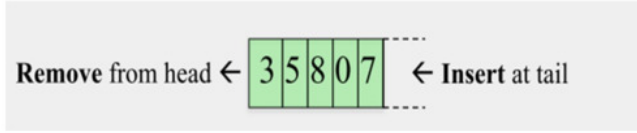
И здесь мы попробуем сделать введение в эту большую тему облачных технологий и систем распределенных вычислений.

Сначала мы рассмотрим общие понятия, которые пригодятся при изучении этой темы.

Давайте обсудим две разные структуры данных.

Первая структура данных – это очередь.

Очередь, это структура данных, где первый зашел, первый вышел.

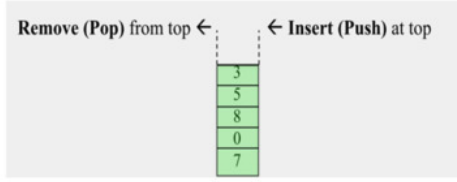


Когда вы удаляете элемент из очереди вы удаляете его из головы очереди.

Когда вы вставляете новый элемент, вы вставляете его в хвост очереди.

Другая структура данных, это стек, который является структурой данных, где первый зашел, последний вышел.

Представьте себе стопку тарелок на столе.

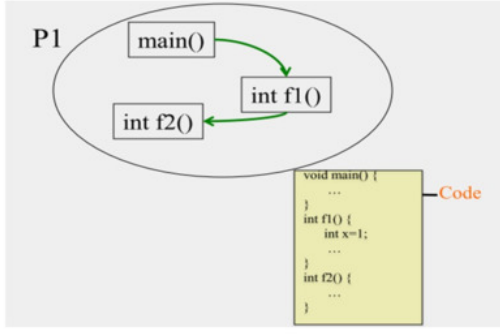


Тарелка, которую вы ставите сверху, вы добавляете последней, и она будет первой, которую вы можете удалить.

Эти две структуры данных, очередь и стек, используются очень широко в информатике, и мы будем использовать понятие стека, когда мы будем обсуждать процессы.

Говоря о процессах, давайте обсудим следующий процесс.

Процесс по существу, это программа в действии.



Этот примерный код состоит из основной функции, которая вызывает функцию f1.

А затем f1 вызывает другую функцию f2.

Это код вы должны скомпилировать и затем выполнить его.

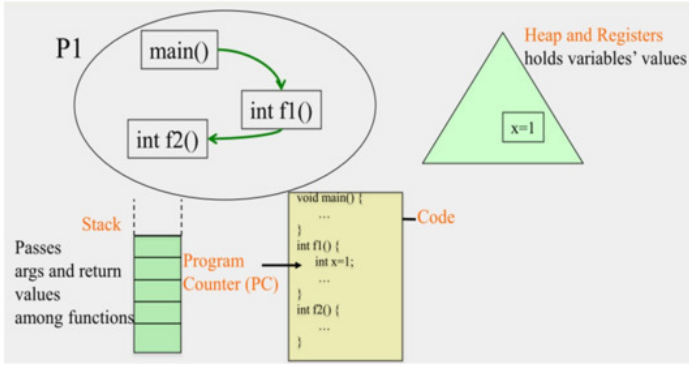
И когда вы его выполняете, когда ваша программа находится в действии, это процесс.

После того, как вы напишете код, он не меняется, и мы не рассматриваем значения переменных как часть кода.

Сам код статичен.

Но существует программный счетчик, который обычно создается компьютером, на котором вы запускаете процесс, который указывает на номер строки кода, где выполняется программа в настоящее время, или скорее, где, процесс в на-

стоящее время выполняется.



Далее, когда функции вызывают друг друга, или в объектно-ориентированной программе методы вызывают друг друга, они используют стек, который содержит аргументы и возвращаемые функциями значения.

Поэтому каждый процесс содержит стек.

Более конкретно, процесс может содержать несколько потоков.

И каждый поток будет содержать собственный стек.

В этом процессе есть только один поток.

Поэтому процесс содержит один стек, и этот стек используется этими функциями или методами, чтобы передать аргументы и вернуть значения.

Так, например, когда main вызывает f1, main внесет аргументы для f1, поверх стека.

И когда f1 начнет выполнение, она вытолкнет или удалит элементы из верхней части стека и будет использовать их для выполнения.

Точно так же, когда f1 вызовет f2, f1 разместит аргументы на вершине стека для f2, а затем f2 вытолкнет их из стека, выполнится, а затем поместит значения результата поверх стека.

f1 затем вытолкнет результат из стека.

И, наконец, когда f1 нужно вернуть значение, она внесет его в верхнюю часть стека.

И когда выполнение вернется к main, она удалит значение из верхней части стека.

Таким образом, стек является важной частью состояния процесса, потому что он сообщает вам, в каком месте исполнения программы вы находитесь, в отношении функций, вызывающих друг друга.

И наконец, функции могут иметь локальные переменные, такие как x.

Там могут быть и глобальные переменные, и, конечно, в объектно-ориентированных программах, у вас есть объекты, которые хранят много полей.

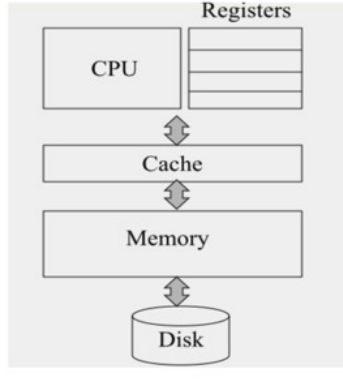
Эти данные хранятся в том, что называется кучей.

Куча – это, по существу, данные, которые были созданы методами, или объектами.

И эти данные также появляются в куче и удаляются из нее в процессе выполнения программы.

Также есть регистры, которые содержат недавние значения, к которым был получен доступ процессом.

Давайте посмотрим упрощенную версию компьютерной архитектуры.



Здесь есть процессор, который выполняет инструкции, которые присутствуют в вашем коде.

Также есть регистры, которые расположены вместе с процессором.

Это небольшие части памяти, к которым CPU можно быстро получить доступ.

И как правило, существует только небольшое количество

регистров, не более нескольких десятков регистров.

Также есть кеш, который является немного большей памятью, чем набор регистров.

И чем больше память, тем медленнее доступ к ней.

Таким образом, доступ к кешу медленнее доступу к регистрам.

Но доступ к кешу все еще довольно быстрый.

Помимо кэша также есть основная память, или Random Access Memory, или RAM, которая еще больше, чем кеш, а, следовательно, медленнее, чем кеш.

И, наконец, есть жесткий диск, у которого намного больше памяти, чем у основной памяти, и доступ к ней еще медленнее.

Таким образом, по мере того, как вы поднимаетесь от диска, к основной памяти, кешу, регистрам, увеличивается скорость и уменьшается память.

И когда вы пишете программу и компилируете ее, она компилируется в машинные инструкции низкого уровня.

Эти машинные инструкции могут быть специфическими для архитектуры машины, на которой вы работаете, или они могут быть кодом для виртуальной машины, как, например, виртуальной машины JVM.

В любом случае, эти низкоуровневые машинные инструкции являются исполняемой версией вашей программы, и они сохраняются в файловой системе на вашем диске.

Когда ваша программа начинает выполняться, когда она

становится процессом, тогда CPU загружает инструкции их в основную память, а затем в кеш и в регистры.

И как правило, кэш и регистры содержат последние несколько обработанных инструкций.

Теперь, выполняя каждую команду, процессор, выполняющий этот процесс, загружает данные, необходимые для инструкции, в память, а затем, если необходимо, в кэш и регистры.

И если есть какие-то изменения, которые происходят с такой переменной, как x, тогда они сохраняются сначала в кеш, а затем в основную память.

Это конечно очень упрощенная картина.

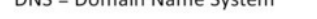
Компьютерные архитектуры могут быть гораздо более сложными, чем эта.

Но чтобы понять, как работают процессы этого достаточно.

Теперь давайте обсудим несколько отвлеченных тем, относящихся к веб приложениям.

Давайте обсудим, что такое DNS.

DNS – это система доменных имен.



Это набор серверов, которые расположены по всему миру, и, DNS очень важен для работы в Интернете.

Как правило, вход в DNS-систему – это URL-адрес.

URL-адрес – это имя, это читаемая пользователем строка, которая уникально идентифицирует объект.

И обычно, когда вы открываете свой браузер, вы вводите URL-адрес, и ваш браузер связывается с DNS-системой и, дает DNS-системе имя этого URL-адреса.

Что возвращает DNS в ваш браузер?

Он возвращает IP-адрес веб-сервера, на котором размещается этот контент, так что ваш браузер может затем может отправить запрос на этот IP-адрес и получить фактическое содержимое этой веб-страницы.

Таким образом, IP-адрес является идентификатором, это

уникальная строка, указывающая на конкретный объект.

Таким образом, по сути, DNS – это система, которая переводит читаемый URL-адрес в уникальный идентификатор, IP-адрес.

IP-адрес может ссылаться либо на фактический веб-сервер, который хранит контент, либо, может быть, на сервер передачи данных, такой как сетевой сервер распространения контента.

Теперь, после обсуждения этих понятий, мы начнем введение в облачные вычисления и рассмотрим, чем отличаются облачные вычисления от предыдущего поколения распределенных систем.

В настоящее время имеется большой интерес к облачным вычислениям.

Но вопрос в том, что такое облачные вычисления?

Есть много облачных провайдеров, о многих из которых вы, возможно, слышали.

Наиболее популярными облачными провайдерами являются Amazon Web Services, Microsoft Azure, и Google Compute Engine, и есть целая группа других компаний.

- AWS: Amazon Web Services
  - EC2: Elastic Compute Cloud
  - S3: Simple Storage Service
  - EBS: Elastic Block Storage
- Microsoft Azure
- Google Compute Engine
- Rightscale, Salesforce, EMC, Gigaspaces, 10gen, Datastax, Oracle, VMWare, Yahoo, Cloudera

Например, Amazon предлагает различные услуги, и три из их самых популярных услуг называются EC2, S3 и EBS.

EC2 представляет собой вычислительное облако, которое обеспечивает вычислительные службы.

S3 – это простая служба хранения, которая предоставляет вам возможность хранения данных, чтобы вы могли получить к ней доступ из любой точки мира.

И EBS – это хранение блоков, к которым экземпляры EC2 могут получить доступ во время работы.

Теперь, существует две категории облаков – это публичное облако и приватное облако.

Приватное облако доступно только для привилегированных пользователей или персонала.

Например, если вы работаете в компании X, а компания X

имеет центр обработки данных или облако, доступное только для сотрудников компании, это приватное облако.

С другой стороны, общедоступное облако – это облако, к которому может получить доступ любой человек в любой точке мира.

Таким образом, среди публичных облаков, есть такие популярные, как Amazon AWS, Google Compute Engine и Microsoft Azure.

Amazon S3 – это простая служба хранения, которая позволяет хранить произвольные наборы данных, и вы платите за гигабайты в месяц, которые вы храните.

EC2 – это вычислительное облако, которое позволяет загружать и запускать произвольные образы ОС.

По сути, это виртуальные машины, и вы платите за процессорные часы, которые вы используете.

Google AppEngine или Compute Engine предлагает возможность для разработки приложений с последующей загрузкой их в облако.

Например, вы можете использовать Google облако для размещения ваших собственных веб-сервисов.

И Microsoft Azure предлагает аналогичные Google продукты.

Таким образом, облачные вычисления привлекательны для клиентов, потому что они позволяют им экономить как время, так и деньги.

Потому что для развертывания своего собственного сер-

вера нужно его сначала приобрести, затем подключить, установить программное обеспечение, и все это занимает несколько недель.

Облако позволяет все это сделать за несколько минут.

И в результате неудивительно, что сотни стартапов в Силиконовой долине используют облачных провайдеров.

Итак, что такое облако?

Когда вы спросите разных людей в компаниях или в академических кругах, что такое облако, они дадут вам очень разные ответы.

Некоторые скажут: «это просто кластер, куча серверов, соединенных сетью».

Другие могут сказать «это суперкомпьютер. У него больше мощности чем у простого кластера, который вы можете запустить в вашей лаборатории».

Другие могут сказать: «облако хранит много данных, поэтому оно отличается от суперкомпьютера, потому что оно скорее хранит данные, чем занимается вычислениями».

И нет единого определения для облака.

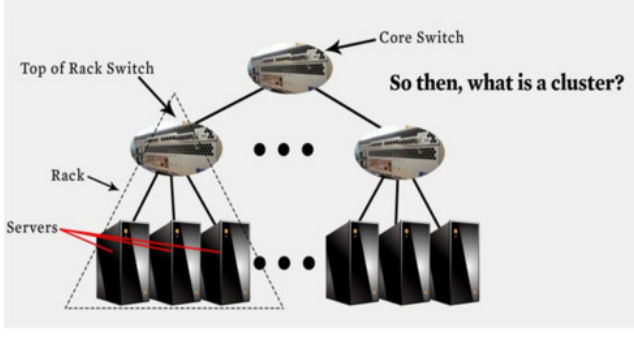
Мы будем исходить из очень простого определения.

Облако состоит из большого количества ресурсов хранения вместе с вычислительными циклами.

И есть два вида облаков – это облако с одним сайтом и географически распределенное облако.

Облако с одним сайтом часто называют центром обработки данных, и оно состоит из серверов или вычислительных

узлов, которые сгруппированы в стойки.



Стойка представляет собой единицу из нескольких серверов, которые имеют общее питание и общий переключатель верхнего уровня.

Эти переключатели стоек часто связаны через сетевую топологию.

Например, одна из самых популярных – это двухуровневая древовидная топология, в которой переключатели стоек находятся на одном уровне, а затем у вас есть основные переключатели, соединяющие все стойки между собой.

В дополнение к этим, узлам, которые используются для вычислений, есть также внутренние узлы в стойках, но они используются для хранения.

Это могут быть узлы, которые имеют SSD диски.

И, наконец, есть программное обеспечение, которое работает на всех этих серверах, а также маршрутизаторы.

Программное обеспечение включает в себя операционные системы, различные приложения пользовательского уровня, поддержку IP-протокола, коммутации и маршрутизации.

Так что это единственный центр данных.

И как правило, такой центр обработки данных размещается в одном здании.

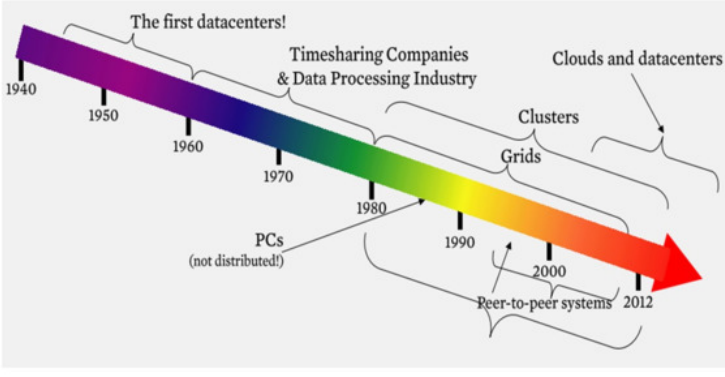
Но компания также может иметь несколько географически распределенных центров обработки данных, и они могут быть подключены друг к другу.

Таким образом, в этом случае, есть много сайтов, каждый из которых является центром обработки данных, и это часто называют географически распределенным облаком.

Но теперь остается вопрос «чем это отличается от кластера?», потому что это похоже на кластер.

Облачные вычисления не являются первой распределенной системой, которая появилась.

Первые несколько компьютеров, которые были построены в 1940-х годах, были построены на основе архитектуры или подобной архитектуры, такой, как мы знаем сегодня.



Они были фактически центрами данных; они занимали целые, большие залы и большие лаборатории.

Это была эпоха индустрии обработки данных, когда обработка данных была сосредоточена в таких центрах.

Затем в 1980-х годах появились персональные компьютеры, которые упростили создание кластеров или сетей рабочих станций, и это привело к появлению распределенных вычислений, а затем крупномасштабных систем, таких как одноранговые системы Peer-to-peer (P2P) в 1990-х и 2000-х годах.

С появлением персональных компьютеров эпоха индустрии обработки данных ушла.

И в настоящее время мы совершили круг и вернулись к эпохе индустрии обработки данных, путем создания

крупномасштабных кластеров, которые обрабатывают очень большие объемы данных.

И отличие сейчас от 1960-х и 70-х в масштабах и мощностях.

Итак, чем облачная инфраструктура отличается от предыдущих поколений распределенных вычислительных систем.

Сегодняшние облака имеют масштабирование, доступ по требованию, интенсивность данных и новую парадигму облачного программирования.

Масштабирование означает, что датацентры очень большие, они содержат десятки тысяч, а иногда и сотни тысяч серверов, и вы можете запускать ваши вычисления на стольких серверах, как вы захотите.

Доступ по требованию означает, что вы не подписывайте контракт по покупке ресурсов заранее.

Нет никаких предварительных обязательств, вы платите только за то, что используете, и каждый может использовать эти ресурсы.

Третий аспект, это интенсивность данных.

То, что раньше было мегабайтами, стало терабайтами данных.

И эти данные нужно сохранить, и их необходимо обработать, и, возможно, в режиме реального времени.

Наконец, появляются новые парадигмы облачного программирования, которые упрощают обработку такие больших объемов данных.

Эти, новые программные парадигмы и парадигмы хранения являются доступными, их легко программировать и легко настраивать, и многие из них являются системами с открытым исходным кодом.

Так, это Hadoop – проект для разработки и выполнения распределённых программ.

И это MongoDB – NoSQL база данных и так далее.

И облачные сервисы классифицируются исходя из характера предоставляемых услуг.

HaaS означает «аппаратное обеспечение как услуга».

HaaS: Hardware as a Service

IaaS: Infrastructure as a Service

PaaS: Platform as a Service

SaaS: Software as a Service

По сути, это означает, что вы получаете доступ к голым машинам, и вы можете делать с ними все, что хотите.

Например, если вы покупаете кластер, приватное облако,

тогда, вы запускаете аппаратное обеспечение как сервис.

Но предоставлять аппаратное обеспечение как сервис другим пользователям, особенно тем, которым вы не доверяете, может быть, не очень хорошая идея из-за рисков безопасности.

Поэтому существует IaaS – инфраструктура как сервис, которая позволяет получить к машинам, и установить свои собственные операционные системы, но без доступа администратора root.

По сути, здесь используется виртуализация, чтобы, вы могли установить собственные виртуальные машины.

PaaS – платформа как сервис, по сути, представляет собой разновидность IaaS.

Вы не получаете доступ к самим виртуальным машинам, но вы пишете свой код, и он тесно интегрирован с программной платформой.

Например, App Engine от Google позволяет писать код на Python, Java или Go, а затем автоматически масштабирует ваше приложение в зависимости от входящей нагрузки.

При этом вы не задумываетесь об установке дополнительных виртуальных машин.

И наконец, SaaS – программное обеспечение как сервис дает вам доступ к приложениям как сервисам, когда они вам понадобятся, и снова вы платите по требованию.

Например, это Google документы или облачный Microsoft Office.

Итак, какая связь между облаками и распределенными системами?

Облако на самом деле является распределенной системой.

Облако состоит из сотен тысяч компьютеров на стороне центра обработки данных в интегрированном центре.

Это мы называем серверной стороной.

На стороне клиента может быть от тысяч до миллионов машин, которые получают доступ к услугам, и которые размещаются на этих серверах.

Это возможно, веб-страницы, сайты, объекты, которые хранятся, и различные сервисы.

Серверы на стороне центра обработки данных обмениваются между собой данными.

А клиенты напрямую взаимодействуют с серверами.

И каждый клиент может связываться с одним или более серверами.

Клиенты могут также взаимодействовать друг с другом через облако.

Тот факт, что серверы общаются между собой, означает, что это распределенная система, состоящая из множества разных серверов, отправляющих и получающих сообщения между собой.

Это называется кластером.

Клиенты, взаимодействующие с серверами, также создают распределенную систему.

Клиенты, общающиеся друг с другом, создают peer-to-

peer распределенную систему.

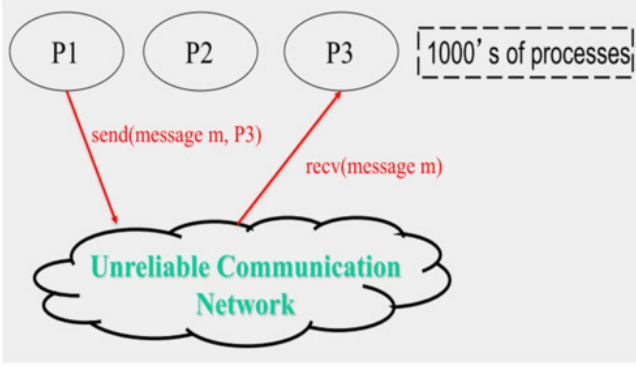
Все это означает, что облака являются особым классом распределенных систем.

Теперь давайте попытаемся определить термин распределенная система.

Давай примем, что распределенная система представляет собой совокупность объектов, каждый из которых является автономным, программируемым, асинхронным, но и подверженным сбоям, и эти объекты обмениваются информацией через ненадежную среду связи.

Здесь объекты – это по существу процессы.

Таким образом, каждый объект является процессом, который работает на каком-либо устройстве.



И каждый объект является автономным и программируемым, так как это процесс.

Асинхронный также очень важно, это означает, что каждый процесс или каждый объект работает в соответствии со своим собственным циклом или часами, так как каждое устройство имеет свое системное время, и эти часы не синхронизируются друг с другом.

И эти объекты подвержены сбоям, и их коммуникация не надежна.

Асинхронность отличает распределенные системы от параллельных систем.

Параллельные системы включают в себя многопроцессорные системы и суперкомпьютеры.

Но, по сути, при этом очень большое количество процессоров используют одну и ту же материнскую плату.

Они взаимодействуют друг с другом по тесно связанной сети, и все они имеют синхронизированные циклы или часы.

И этим параллельная система отличается от распределенной системы.

Таким образом, процесс, это автономная работающая программа, которая может иметь несколько потоков.

Процесс выполняет множество задач, которые могут быть распределены по потокам, и которые могут выполняться на одном или нескольких процессорах.

В распределенных системах, эти процессы работают на разных устройствах, системные времена которых не син-

хронизированы.

В параллельных системах каждый поток процесса выполняется на своем процессоре или ядре процессора, и задачи таким образом выполняются параллельно.

При этом процессоры имеют синхронизированное системное время.

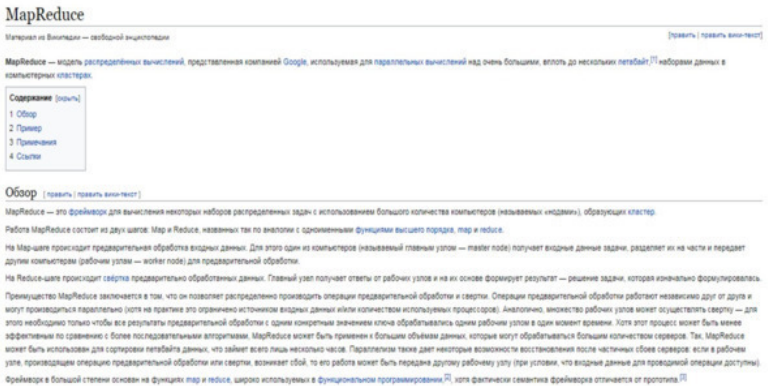
MapReduce

Облачные и распределенные системы

MapReduce

MapReduce – это модель распределённых вычислений, представленная компанией Google.

И эта модель используется для параллельных вычислений над очень большими наборами данных в компьютерных кластерах.



MapReduce

Материал из Википедии — свободной энциклопедии

[править | править вики-текст]

MapReduce — модель распределённых вычислений, представленная компанией Google, используемая для параллельных вычислений над очень большими, вплоть до нескольких петабайт,[1] наборами данных в компьютерных кластерах.

Содержание [скрыть]

1 Обзор
2 Пример
3 Примечания
4 Ссылки

Обзор [править | править вики-текст]

MapReduce — это фреймворк для вычисления некоторых наборов распределенных задач с использованием большого количества компьютеров (называемых «нодами»), образующих кластер.

Работа MapReduce состоит из двух шагов: Map и Reduce, названных так по аналогии с одноимёнными функциями высшего порядка, map и reduce.

На Map-шаге происходит предварительная обработка входных данных. Для этого один из компьютеров (называемый главным узлом — master node) получает входные данные задачи, разделяет их на части и передает другим компьютерам (рабочим узлам — worker node) для предварительной обработки.

На Reduce-шаге происходит свёртка предварительно обработанных данных. Главный узел получает ответы от рабочих узлов и на их основе формирует результат — решение задачи, которая изначально формулировалась.

Преимущество MapReduce заключается в том, что он позволяет распределенно производить операции предварительной обработки и свертки. Операции предварительной обработки работают независимо друг от друга и могут производиться параллельно (хотя на практике это ограничено источником входных данных и/или количеством используемых процессоров). Аналогично, множество рабочих узлов может осуществлять свертку — для этого необходимо только чтобы все результаты предварительной обработки с одним конкретным значением ключа обрабатывались одним рабочим узлом в один момент времени. Хотя этот процесс может быть менее эффективным по сравнению с более последовательными алгоритмами, MapReduce может быть применен к большим объёмам данных, которые могут обрабатываться большим количеством серверов. Так, MapReduce может быть использован для сортировки петабайта данных, что займет всего лишь несколько часов. Параллелизм также дает некоторые возможности восстановления после частичных сбоев серверов: если в рабочем узле, производящем операцию предварительной обработки или свертки, возникает сбой, то его работа может быть передана другому рабочему узлу (при условии, что входные данные для проводимой операции доступны).

Фреймворк в большой степени основан на функциях map и reduce, широко используемых в функциональном программировании,[2], хотя фактически семантика фреймворка отличается от прототипа.[3]

Термины map и reduce, которые составляют термин MapReduce, заимствованы из функциональных языков, таких как Lisp.

Например, вы хотите вычислить сумму квадратов.

(map square '(1 2 3 4))

- Output: (1 4 9 16)

[processes each record sequentially and independently]

(reduce + '(1 4 9 16))

- (+ 16 (+ 9 (+ 4 1)))
- Output: 30

[processes set of all records in batches]

Функция map – функция, которая может быть применена к любому из этих целых чисел и вычисляет квадрат каждого числа.

Так что map здесь является мета функцией, которая обрабатывает каждую запись.

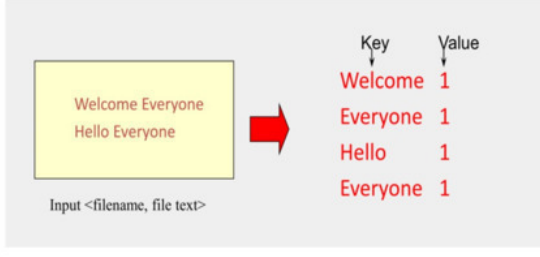
Это первая часть.

Вторая часть – это функция reduce, которая получает на вход список соответствующих квадратов целых чисел и просто суммирует их.

reduce здесь снова является мета функцией, которая применяется к группе записей.

Предположим, что у нас есть текст, и нам нужно произвести подсчет для каждого слова, которое появляется в этом наборе данных.

Как сделать это? Особенно, когда вы имеете дело с большими объемами данных?



Здесь и появляется парадигма MapReduce.

Таким образом, map как задача или как объект обрабатывает отдельные записи для генерации промежуточных ключей / значений.

Если это простой файл, можно пройти через эти записи последовательно.

Но вы можете сделать этот процесс параллельным, особенно когда у вас большой набор данных.

Вы можете параллельно обрабатывать отдельные записи для генерации промежуточных пар ключ / значение.

Если у вас очень большой набор данных, вы можете разделить свой входной набор данных.

И назначить задачу map для каждого куска данных.

И соответствующий результат будет таким же, как если бы у вас была только одна задача map.

И это поможет существенно ускорить процесс.

После результата map, у нас есть ввод для reduce.

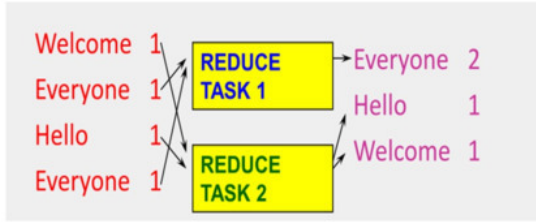
Reduce производит слияние промежуточных результатов в один результат, исходя из ключей значений.

Как распараллелить эту фазу reduce?

Фаза reduce не обрабатывает эти записи независимо, другими словами, одна запись и другая запись должны обрабатываться вместе, так как они имеют одинаковые ключи.

Таким образом, единственный способ распараллелить этот процесс, это разделить задачи reduce по ключам.

Существуют разные способы разбиения ключей на задачи.



Один из способов разделения – это использование хэшей. Вы берете ключ, и обрабатываете его хеш-функцией.

Затем делите хэш на количество задач reduce и в остатке от деления получаете к какой reduce задаче данный ключ относится.

Например, если есть 10 задач reduce, эта операция вернет значения от 0 до 9 для всех ключей.

У парадигмы MapReduce есть реализация с открытым исходным кодом Apache Hadoop, это набор утилит, библиотек и фреймворк для разработки и выполнения распределённых программ, работающих на кластерах из сотен и тысяч узлов.

Итак, вот что такое Map в Hadoop.

```
public static class MapClass extends MapReduceBase
            implements Mapper<LongWritable, Text, Text,
              IntWritable> {
  private final static IntWritable one =
    new IntWritable(1);
  private Text word = new Text();

  public void map( LongWritable key, Text value,
            OutputCollector<Text, IntWritable> output,
            Reporter reporter)
    throws IOException {
    String line = value.toString();
    StringTokenizer itr = new StringTokenizer(line);
    while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            output.collect(word, one);
    }
  }
}
```

У вас есть MapClass, который расширяет базовый класс и реализует интерфейс.

И главная функция здесь – это map.

Эта функция принимает значение, которое в этом случае является текстом.

Значением может быть одна строка текста во входном файле.

Эта строка разбивается на слова.

И для каждого слова вы выводите пару ключ-значение.

Таким образом это промежуточная выходная пара ключ-значение.

Здесь у нас есть ReduceClass, который имеет функцию reduce, получающую на вход ключ и значения, потому что у вас могут быть много значений, связанных с данным клю-

чом.

```
public static class ReduceClass extends MapReduceBase
            implements Reducer<Text, IntWritable, Text,
            IntWritable> {
  public void reduce(
            Text key,
            Iterator<IntWritable> values,
            OutputCollector<Text, IntWritable> output,
            Reporter reporter)
      throws IOException {
            int sum = 0;
            while (values.hasNext()) {
              sum += values.next().get();
            }
            output.collect(key, new IntWritable(sum));
  }
}
```

Эта функция reduce вызывается для каждого ключа, который относится к данной задаче reduce.

Таким образом, reduce будет проходить через все значения и суммировать их и вырабатывать пару «ключ-значение», где ключ совпадает с ключом ввода, а значение – фактически сумма входных значений.

Также у нас есть некоторый код, который имеет функцию запуска, указывая имя работы, определяя ключи и выходные значения, и в конце запуская работу.

```
// Tells Hadoop how to run your Map-Reduce job
public void run (String inputPath, String outputPath)
            throws Exception {
  // The job. WordCount contains MapClass and Reduce.
  JobConf conf = new JobConf(WordCount.class);
  conf.setJobName("mywordcount");
  // The keys are words
  (strings) conf.setOutputKeyClass(Text.class);
  // The values are counts (ints)
  conf.setOutputValueClass(IntWritable.class);
  conf.setMapperClass(MapClass.class);
  conf.setReducerClass(ReduceClass.class);
  FileInputFormat.addInputPath(
            conf, newPath(inputPath));
  FileOutputFormat.setOutputPath(
            conf, new Path(outputPath));
  JobClient.runJob(conf);
}
```

Посмотрим пример приложения, который использует MapReduce.

Это распределенный grep.

Предположим, у вас есть большой набор файлов с большими текстами в них.

И у вас есть шаблон, который может быть регулярным выражением или просто словом, или набором слов, и вы хотите вывести все строки текста, соответствующие этому шаблону.

Таким образом, Map будет принимать на вход каждую строку текста и проверять ее на соответствие шаблону, а затем выводить эту строку как ключ.

Reduce будет просто копировать промежуточные данные на выход, не выполняя никакой обработки, если вы конечно не захотите, например, соединить все строки.

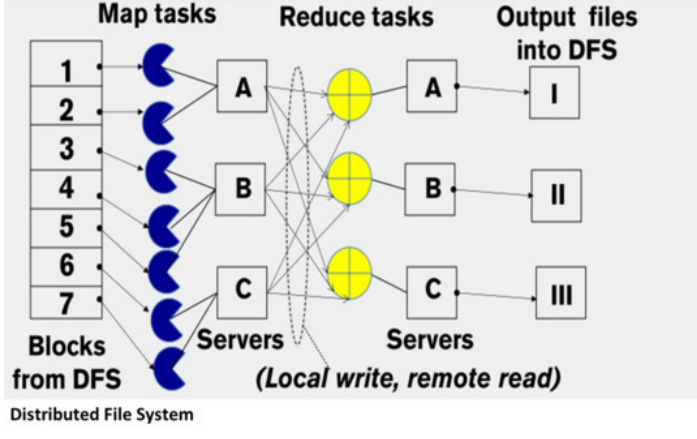
Решая такую простую задачу на одной машине, для больших объемов данных, вы можете потратить очень много времени.

Преимущество распределенного grep здесь в скорости обработки.

С помощью MapReduce вы можете запускать ваше приложение, даже если ваши данные распределены на нескольких серверах.

Итак, как программировать с MapReduce?

С точки зрения пользователя, пользователь записывает программу map, ее метод map, а также записывает программу reduce, и ее метод reduce.



Затем запускает работу, определяя количество задач map и reduce, и затем ожидает результата.

По сути, работа пользователя очень простая, потому что пользователю не нужно много знать о Hadoop или распреде-

ленном программировании.

Это внутри, реализация парадигмы MapReduce, и собственно планировщик должен обеспечить распараллеливание map, он должен разделить данные между различными задачами map.

И он должен передать данные из map в reduce, при этом разделяя ключи по reduce задачам.

А также необходимо распараллелить reduce.

Другими словами, необходимо запланировать сами задачи reduce.

И, наконец, необходимо реализовать хранилище для ввода map, для вывода map, которое совпадает с вводом reduce, а также реализовать вывод reduce.

Кроме того, нужно обеспечить, чтобы фаза reduce стартовала только после окончания фазы map.

Итак, как решить все эти проблемы?

В облаке распараллелить map легко, потому что каждая задача map является независимой от другой задачи map, и поэтому эти задачи map могут быть определены для выполнения любому серверу.

Обычно задачи map назначаются серверу, к которому эти данные наиболее близко находятся, чтобы уменьшить сетевые издержки.

Далее необходимо гарантировать, чтобы все исходящие записи map с одним и тем же ключом были присвоены одному и тому же reduce.

И это поможет перевести данные с map на reduce.

В этом случае вы используете функцию partitioning.

Например, как мы обсуждали ранее, может использоваться функция хэш-разбиения, когда каждому ключу присваивается номер задачи, который получается путем вычисления остатка от деления хеша ключа на количество reduce задач.

Завершить фазу reduce также легко, потому что каждая задача reduce не зависит от другой.

Каждой задаче reduce присваивается набор ключей, и эти наборы ключей не пересекаются друг с другом.

И поэтому их можно запустить независимо друг от друга.

Наконец, вам нужно реализовать хранилище.

Ввод map в начале идет из распределенной файловой системы, вывод map идет в локальную файловую систему map узла.

Ввод reduce идет из множества удаленных дисков, используя локальные файловые системы.

Вывод reduce идет в распределенную файловую систему.

Эта распределенная файловая система запускается обычно на тех же серверах, где выполняются задачи map и reduce.

Например, Apache Hadoop использует HDFS, известную как распределенная файловая система Hadoop.

Обычно эта файловая система хранит множественные копии одного и того же входного блока данных.

Она копирует файловые блоки как минимум три раза и эти три файловые копии размещаются на трех разных сер-

верах.

И поэтому, когда запускается задача map, необходимо извлечь блок данных, который является его блоком входных данных с одного из серверов, который хранит его в настоящее время.

Задача запрашивает онлайн-файловую систему HDFS, чтобы сделать это, и эта передача выполняется быстрее, если сервер, на котором расположен этот конкретный блок, фактически является тем же сервером, на котором выполняется задача map.

Вывод map не хранится в распределенной файловой системе.

Вместо этого вывод map сохраняется на локальном диске на сервере, на котором выполняется задача map.

И ввод данных reduce производится с этих удаленных дисков.

Причина, по которой этот промежуточный траффик между map и reduce использует локальную файловую систему – это скорость передачи данных и потому что эти данные не нужны внешнему пользователю.

Наконец, когда результат reduce получен, он записывается в распределенную файловую систему обратно, где он становится доступен.

Давайте немного посмотрим, как работает планировщик.

Планировщик YARN – это планировщик, который используется в Apache Hadoop.

YARN означает Yet Another Resource Negotiator.

Он обрабатывает каждый сервер как коллекцию контейнеров.

Контейнер – это процессор с некоторой памятью.

Таким образом, каждый сервер состоит из коллекции контейнеров.

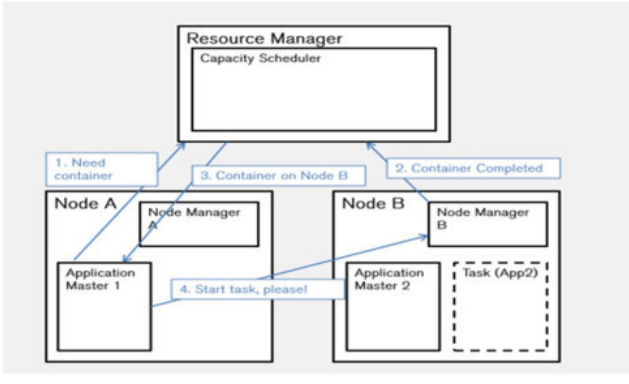
Так, например, если сервер имеет 4 ядра и 4 гигабайта ОЗУ, в каждом контейнере есть одно ядро и 1 гигабайт ОЗУ, и у этого сервера есть 4 контейнера и, по существу, он может выполнять четыре задачи по одной в каждом контейнере.

YARN имеет три основных компонента.

Это менеджер ресурсов, администраторы узлов и мастера приложений.

И существует один глобальный менеджер ресурсов, который запускает планировщика.

Существует один менеджер узла на один сервер в системе.



Это Daemon, который отвечает за все специфическое управление сервером, а также отвечает за мониторинг сбоев задач, которые выполняются на этой конкретной машине.

Затем есть Application Master, или мастер приложения, который также работает на одном из серверов, и отвечает за согласование контейнеров с диспетчером ресурсов и менеджерами узлов.

Он также отвечает за взаимодействие с менеджерами узлов, чтобы выяснить, умер ли какой-либо из них, чтобы перенести с него запущенные задачи.

Теперь давайте посмотрим, как MapReduce разбирается с ошибками.

Наиболее частой ошибкой является отказ самого сервера, и отказ сервера может привести к сбою нескольких компо-

нентов Hadoop планировщика YARN.

Серверы запускают менеджеров узлов, у них запущены задачи, на одном из серверов работает диспетчер ресурсов, а также может работать мастер приложений.

И для решения проблем с отказами серверов, есть пульсация.

Менеджер узла на каждом сервере отправляет пульсацию центральному менеджеру ресурсов.

И если сервер не работает, и эта пульсация останавливается, менеджер ресурсов знает, что менеджер узла не работает.

Он дает знать об этом всем мастерам приложений, и мастера приложений перенаправляют свои задачи.

Менеджер узлов отслеживает каждую задачу, запущенную на своем сервере, поэтому, если одна из задач выходит из строя, эта задача помечается как протаивающая, и, либо менеджер узла ее перезапускает, если это возможно, либо он сообщает диспетчеру ресурсов или мастеру приложения, что эта задача не выполнена.

И наконец, мастер приложения также периодически пульсирует менеджеру ресурсов.

Если, мастер приложения не работает, менеджер ресурсов перезапустит мастер приложения, и он затем синхронизируется с его запущенными задачами.

Далее сам менеджер ресурсов может перестать работать.

В этом случае, чтобы справиться с этим, поддерживается вторичное жесткое резервное копирование, чтобы вторич-

ный менеджер ресурсов мог сразу заработать после отказа менеджера ресурсов.

Протокол Gossip

Облачные и распределенные системы

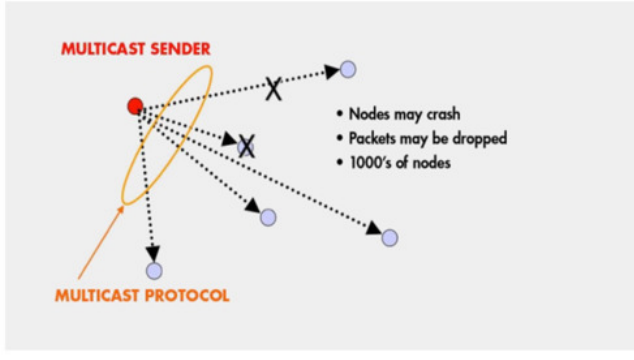
Протокол Gossip

Далее мы рассмотрим класс протоколов, называемый gossip протоколами сплетен или эпидемическими протоколами.

Но сначала начнем с постановки задачи.

Задача, которую gossip пытается решить, называется групповой передачей multicast.

Итак, что такое групповая передача?



Предположим, что у вас есть группа процессов, или группа узлов.

И каждый из этих процессов или каждый из этих узлов является процессом на каком-либо хосте в Интернете или процессом, подключенном к сети.

И, по сути, все, что нам нужно, это чтобы эти процессы или узлы могли взаимодействовать друг с другом, отправляя и получая сообщения.

Это задача многоадресной рассылки.

Это задача, когда вы хотите получить информацию от других членов вашей группы и, конечно же, здесь я показываю только одно сообщение многоадресной рассылки, но может быть одновременно много сообщений многоадресной рассылки, каждое из которых от потенциально другого отпра-

вителя.

Теперь, многоадресная рассылка отличается от широковещательной трансляции, где у вас есть блок информации, который вы хотите отправить на всю сеть, – многоадресная рассылка более ограничена.

Она работает только внутри определенной группы узлов или группы процессов.

Итак, какие требования для протокола многоадресной рассылки?

Ну, два из самых важных требований для облачных вычислений, это отказоустойчивость и масштабируемость.

Вы хотите, чтобы ваша многоадресная рассылка была надежной, и чтобы все определенные получатели получили эту рассылку, несмотря на сбои и задержки, которые могут произойти в сети.

Вы также хотите, чтобы протокол многоадресной рассылки был масштабируемым, и накладные расходы в пересчете на узел не росли быстро, при росте количества узлов.

Обычно протокол многоадресной передачи реализован на уровне приложения, что означает, что он не работает с низлежащей сетью, но это не является правилом, так как существует IP протокол многоадресной рассылки, который взаимодействует с уровнем низлежащей сети.

Обычно IP протокол многоадресной рассылки протокол реализован в маршрутизаторах и коммутаторах.

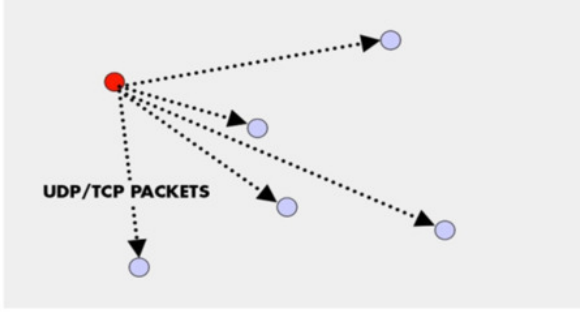
Однако этот протокол может быть не включен во многих

маршрутизаторах и коммутаторах.

И большинство протоколов многоадресной рассылки являются протоколами уровня приложения, что означает, что они обеспечивают взаимодействие процессов между собой, не беспокоясь о том, что происходит в низлежащей сети.

И одним из самых простых способов многоадресной рассылки является централизованный подход.

У вас есть отправитель, который имеет список получателей, и он просто в цикле for или while отправляет каждому из этих получателей пакет либо UDP User Datagram Protocol, либо TCP Transmission Control Protocol, который содержит информацию.



Теперь, одна из проблем с этой реализацией, это то, что

она не обладает отказоустойчивостью.

Если отправитель перестанет работать в середине цикла, тогда только половина получателей получит рассылку.

Конец ознакомительного фрагмента.