Большие данные: современные подходы к хранению и обработке

П.А. Клеменков, С.Д. Кузнецов
parser@cs.msu.su, kuzloc@ispras.ru

Аннотация. Большие данные поставили перед традиционными системами хранения и обработки новые сложные задачи. В данной статье анализируются возможные способы их решения, ограничения, которые не позволяют сделать это эффективно, а также приводится обзор трех современных подходов к работе с большими данными: NoSQL, MapReduce и обработка потоков событий в реальном времени.

Ключевые слова: большие данные; реляционная модель; nosql; mapreduce; hadoop; s4; storm

1. Введение

Мы живем в информационный век. Нелегко измерить общий объем электронных данных, но по оценкам IDC размер «цифровой вселенной» в 2006 г. составлял 0.18 зеттабайт, а к 2011 г. должен был достигнуть 1.8 зеттабайт, продемонстрировав десятикратный рост за 5 лет! Вот только несколько примеров источников таких объемов [1]:

1. Нью-Йоркская фондовая биржа генерирует около терабайта данных в день.

2. Объем хранилища социальной сети Facebook каждый день увеличивается на 500 терабайт.

3. Проект Internet Archive уже хранит 2 петабайта данных и прирастает 20 терабайтами в месяц.

4. Эксперименты на Большом адронном коллайдере могут генерировать около петабайта данных в секунду!

Стремительно растущий объем информации ставит перед нами новые сложные задачи по организации ее хранения и обработки. В статье мы постараемся проанализировать то, как изменился традиционный подход к

организации данных, а также проанализируем новые инструменты работы с большими данными.

2. Что такое «большие данные»?

Итак, данных очень много. Но что такое «много» ? Где тот порог, преодолев который, данные становятся «большими»? Часто используется характеристика, данная исследовательской компанией Gartner: ««Большие данные» характеризуются объемом, разнообразием и скоростью, с которой структурированные и неструктурированные данные поступают по сетям передачи в процессоры и хранилища, наряду с процессами преобразования этих данных в ценную для бизнеса информацию» [2].

Как видно из этого определения, большие данные имеют четыре основные характеристики: объем, разнообразие, скорость и ценность. Рассмотрим их подробнее:

1. **Объем.** Нарастающее количество данных, создаваемых как людьми, так и машинами, предъявляет к IT инфраструктуре новые требования в отношении хранения, обработки и предоставления доступа.

2. **Разнообразие.** Данные содержат разнообразную информацию, представленную разными структурами. Со всем этим, от логов доступа к веб-серверу до операций по кредитным картам, от результатов научных экспериментов до фотографий и видео, необходимо уметь работать.

3. **Скорость.** Важно осознавать, что под скоростью понимается не только скорость, с которой данные поступают в хранилище, но и скорость с которой важная информация из этих данных извлекается.

4. **Ценность.** Большие объемы данных — это ценный ресурс. Но еще ценнее он становится, если позволяет отвечать на насущные вопросы или вопросы, которые могут возникнуть в будущем.

3. Подходы к работе с большими данными

Так сложилось, что инструменты, существовавшие до недавнего времени, оказались не способны справиться с большими объемами. О проблемах, пришедших с эпохой больших данных, способах их преодоления и новых инструментах поговорим в следующих разделах.

3.1. Реляционные СУБД

До определенного момента, практически единственным ответом на вопрос «как хранить и обрабатывать данные?» являлась какая-нибудь реляционная СУБД. Но с увеличением объемов появились проблемы, с которыми классическая реляционная архитектура не справлялась, поэтому инженерам пришлось придумывать новые решения. Попробуем представить те шаги, которые можно предпринять, если СУБД прекращает справляться с объемом выполняемых операций:

1. Естественным первым шагом является попробовать наименее затратные способы. Самый простой, при наличии финансов, способ — это ничего не делать, а просто купить **более мощное оборудование (вертикальное масштабирование)**. Однако бесконечно мощного сервера не существует, а значит вертикальный рост конечен.

2. Более затратный способ — это **оптимизировать запросы**, проанализировав планы их исполнения, и **создать дополнительные индексы**. Такой метод может принести временное облегчение, но дополнительные индексы порождают дополнительные операции, а с ростом объемов обрабатываемых данных эти дополнительные операции приводят к деградации.

3. Следующим шагом может быть внедрение **кэша на чтение**. При правильной организации такого решения, мы можем избавить СУБД от существенной части операций чтения, но жертвуем строгой консистентностью данных. К тому же, этот подход приводит к усложнению клиентского ПО.

4. Выстраивание операций вставки/обновления в **очередь** — неплохое решение, но размер очереди ограничен. К тому же, для обеспечения строгой консистентности, нам придется организовать персистентность самой очереди, а это непростая задача.

5. Наконец, когда все прочие способы перестают работать, наступает момент пересмотреть способ организации самих данных. В первую очередь — произвести **денормализацию** схемы, чтобы уменьшить число нелокальных обращений.

6. Ну а когда и это не работает, то остается только **масштабировать горизонтально**, т.е. разносить вычисления на разные узлы. Здесь приходится окончательно попрощаться с нормализацией и внешними ключами, к тому же нужно ответить на вопросы «по каким признакам распределять новые кортежи по узлам?» и «как произвести миграцию существующей схемы?».

Подводя итоги, можно заключить, что попытки приспособить реляционную СУБД к работе с большими данными приводят к следующему:

1. Отказу от строгой консистентности.

2. Уходу от нормализации и внедрение избыточности.

3. Потере выразительности языка SQL и необходимости моделировать часть его функций программно.

4. Существенному усложнению клиентского программного обеспечению.

5. Сложности поддержания работоспособности и отказоустойчивости получившегося решения.

Необходимо, правда, отметить, что производители реляционных СУБД осознают все эти проблемы и уже начали предлагать масштабируемые кластерные решения. Однако стоимость внедрения и сопровождения подобных решений зачастую не окупается.

3.2. NoSQL

Взглянув на выводы из предыдущего раздела, в голове сразу рождается довольно очевидная мысль — а почему бы не спроектировать архитектуру, способную адаптироваться к возрастающим объемам данных и эффективно их обрабатывать? Подобные мысли привели к появлению движения NoSQL.

Сразу хочется обратить внимание, что NoSQL не подразумевает бездумного отказа от всех принципов реляционной модели. Более того, термин «NoSQL» впервые был использован в 1998 году для описания реляционной базы данных, не использовавшей SQL [3]. Просто теоретики и практики данного подхода справедливо утверждают, что при выборе инструментария необходимо отталкиваться от задачи, а реляционные СУБД подходят не всегда, особенно в эпоху больших данных. Популярность NoSQL стал набирать в 2009 г, в связи с появлением большого количества веб-стартапов, для которых важнейшей задачей является поддержание постоянной высокой пропускной способности хранилища при неограниченном увеличении объема данных. Рассмотрим основные особенности NoSQL подхода [4,5]:

1. **Исключение излишнего усложнения.** Реляционные базы данных выполняют огромное количество различных функций и обеспечивают строгую консистентность данных. Однако для многих приложений подобный набор функций, а также удовлетворение требованиям ACID являются излишними.

2. **Высокая пропускная способность.** Многие NoSQL решения

обеспечивают гораздо более высокую пропускную способность данных нежели традиционные СУБД. Например, колоночное хранилище Hypertable, реализующее подход Google Bigtable, позволяет поисковому движку Zvent сохранять около миллиарда записей в день. В качестве другого примера можно привести саму Bigtable, способную обработать 20 петабайт информации в день [6].

3. **Неограниченное горизонтальное масштабирование.** В противовес реляционным СУБД, NoSQL решения проектируются для неограниченного горизонтального масштабирования. При этом добавление и удаление узлов в кластере никак не сказывается на работоспособности системы. Дополнительным преимуществом подобной архитектуры является то, что NoSQL кластер может быть развернут на обычном аппаратном обеспечении, существенно снижая стоимость всей системы.

4. **Консистентность в жертву производительности.** При описании подхода NoSQL нельзя не упомянуть теорему CAP. Следуя этой теореме, многие NoSQL базы данных реализуют доступность данных (availability) и устойчивость к разделению (partition tolerance), жертвуя консистентностью в угоду высокой производительности. И действительно, для многих классов приложений строгая консистентность данных — это то, от чего вполне можно отказаться.

3.2.1 Классификация NoSQL хранилищ

На сегодняшний день создано большое количество NoSQL решений. Все они основываются на четырех принципах из предыдущего раздела, но могут довольно сильно отличаться друг от друга. Многие теоретики и практики создавали свои собственные классификации, но наиболее простой и общеупотребительной можно считать систему, основанную на используемой модели данных, предложенную Риком Кейтелем (Rick Cattel) [7]:

1. **Хранилища ключ-значение.** Отличительной особенностью является простая модель данных — ассоциативный массив или словарь, позволяющий работать с данными по ключу. Основная задача подобных хранилищ — максимальная производительность, поэтому никакая информации о структуре значений не сохраняется.

2. **Документные хранилища.** Модель данных подобных хранилищ позволяет объединять множество пар ключ-значение в абстракцию, называемую «документ». Документы могут иметь вложенную структуру и объединяться в коллекции. Однако это скорее удобный способ логического объединения, т.к. никакой жесткой схемы у документов нет и множества пар ключ-значение, даже в рамках одной

коллекции, могут быть абсолютно произвольными. Работа с документами производится по ключу, однако существуют решения, позволяющие осуществлять запросы по значениям атрибутов.

3. **Колоночные хранилища.** Этот тип кажется наиболее схожим с традиционными реляционными СУБД. Модель данных хранилищ подобного типа подразумевает хранение значений как неинтерпретируемых байтовых массивов, адресуемых кортежами <ключ строки, ключ столбца, метка времени> [6]. Основой модели данных является колонка, число колонок для одной таблицы может быть неограниченным. Колонки по ключам объединяются в семейства, обладающие определенным набором свойств.

4. **Хранилища на графах**. Подобные хранилища применяются для работы с данными, которые естественным образом представляются графами (например, социальная сеть). Модель данных состоит из вершин, ребер и свойств. Работа с данными осуществляется путем обхода графа по ребрам с заданными свойствами.

Тип	**Примеры**
Хранилища ключ-значение	Redis Scalaris Riak Tokyo Tyrant
Документные хранилища	SimpleDB CouchDB MongoDB
Колоночные хранилища	BigTable Hbase Cassandra
Хранилища на графах	Neo4j

Табл. 1. Классификация NoSQL хранилищ по модели данных.

3.3. MapReduce

Пионером в области больших данных можно считать компанию Google, которая в 2003 г. описала распределенную файловую системы GFS [8], а в 2004 г. представила миру вычислительную модель MapReduce [9]. Именно эти публикации помогли разработчикам свободного поискового движка Apache

Nutch создать проект Hadoop [10], который сегодня фактически стал синонимом термина «большие данные».

3.3.1 Hadoop

Перед тем как рассмотреть Hadoop внимательней, следует ответить на вопрос: «а зачем нужен еще один продукт, если многие NoSQL базы данных предоставляют интерфейсы для MapReduce вычислений?» [11, 12, 13]. Ответ можно получить, рассмотрев производительность современных жестких дисков. Средняя производительность жесткого диска сегодня ~100 МБ/с, что означает возможность прочитать 1 ТБ информации примерно за 2.5 часа. Улучшить такие удручающие показатели можно параллельным чтением с нескольких дисков. Например, тот же самый 1 ТБ можно прочесть со 100 дисков за 2 минуты. Но ведь почти все NoSQL решения поддерживают горизонтальное масштабирование, а следовательно и параллельные дисковые операции? И тут ключевым фактором становится время позиционирования головки. Для того, чтобы операции обновления и чтения были эффективными, NoSQL базам (CouchDB, MongoDB) приходится использовать структуры с произвольным доступом, например B-деревья [14]. А значит, если отказаться от произвольного обновления данных и обрабатывать весь набор последовательно, можно добиться серьезного прироста производительности. Именно этот принцип и положен в основу архитектуры Hadoop.

За хранение и организацию данных в Hadoop кластере отвечает распределенная файловая система HDFS [15]. При проектировании которой использовались следующие принципы:

1. Аппаратные сбои неизбежны. Поэтому HDFS реализует надежные алгоритмы репликации данных, а для метаданных файловой системы поддерживается журнал, позволяющий восстановить требуемое состояние.

2. Поточная обработка и большие объемы. HDFS устроена таким образом, чтобы обеспечить максимальную производительность поточного доступа к данным. К тому же структуры файловой системы оптимизированы для работы с большими файлами.

3. Локальность данных. Намного эффективней выполнять вычисления рядом с данными. HDFS предоставляет приложениям программный интерфейс, который позволяет выполнять вычисления ближе к необходимым данным, сокращая пересылки между узлами кластера.

Вычисления в Hadoop представляются в виде последовательности map и reduce задач. В начале вычислений входное множество данных разбивается на несколько подмножеств. Каждое подмножество обрабатывается на отдельном узле кластера. Map задача на каждом узле получает на вход множество пар

ключ-значение и возвращает другое множество пар. Далее все пары группируются по ключу, сортируются и подаются на вход reduce задаче, которая формирует финальный результат или вход для другой map задачи [1].

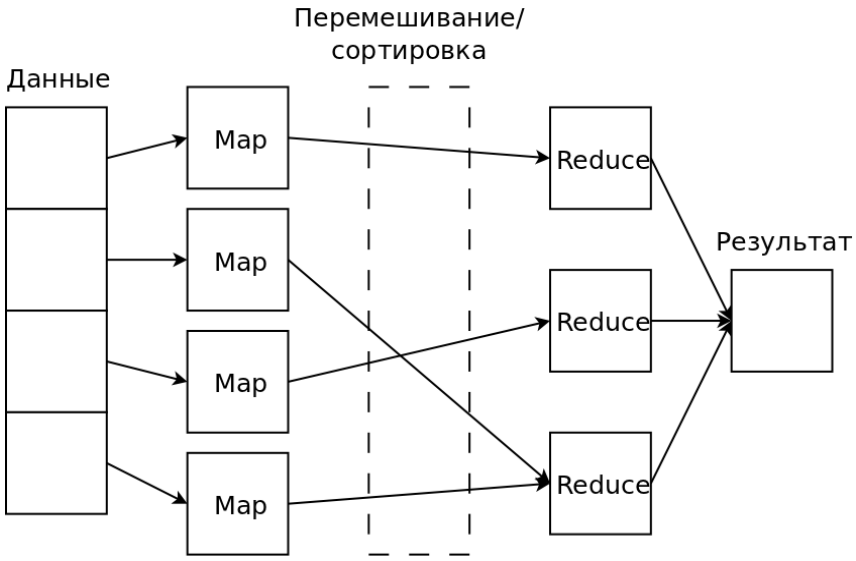

Рис. 1. Схема MapReduce вычислений.

Одним из интересных примеров эффективности Hadoop является тест скорости сортировки [16], целью которого является сортировка записей по 100 байт — 10 байт на ключ и 90 байт на значение. В ноябре 2008 г. рекордом в сортировке 1 ТБ владел кластер компании Google — 68 с. А в мае 2009 г. вышел отчет Yahoo! в котором утверждалось, что их Hadoop кластер отсортировал тестовый набор в 1 ТБ за 62 с! [17]

Вокруг Hadoop сформировалась целая экосистема проектов:

1. Hive — распределенное хранилище данных. Hive управляет данными в HDFS и предоставляет язык запросов HiveQL, основанный на SQL. Запросы HiveQL автоматически транслируются в MapReduce задачи [18].

2. Pig — среда исполнения и высокоуровневый язык, для описания вычислений в Hadoop. Программы Pig также транслируются в MapReduce задачи [19].

3. Hbase — распределенная колоночная база данных. Hbase использует

HDFS, как хранилище и поддерживает как пакетные вычисления, так и произвольный доступ [20].

4. ZooKeeper — высокодоступный координационный сервис, используемый для построения распределенных приложений [21].

3.3.2 Disco

Управление процессами в распределенной системе — сложная задача. А сложнее всего справиться с частичными отказами, когда ошибки в отдельных процессах не должны влиять на вычисления в общем. MapReduce избавляет разработчика от необходимости думать об ошибках, требуется только лишь обеспечить код двух функций: map и reduce. Об остальном позаботится реализация, автоматически определив неудачно завершившиеся задачи и перезапустив их. Такая возможность возникает, потому что задачи независимы, ибо не имеют разделяемых ресурсов [22]. Подобная особенность делает модель акторов [23] идеальной для реализации MapReduce фреймворка.

Этим и воспользовались разработчики исследовательского центра Nokia, создав проект Disco [24]. Особенностью Dicso является то, что ядро системы разработано на функциональном языке Erlang [25], снискавшем славу инструмента, отлично подходящего для программирования распределенных вычислений на основе модели акторов. Компания Ericsson, использующая язык для программирования коммутационных узлов своей телефонной сети, даже заявила о достижении показателя отказоустойчивости оборудования в 99.9999999% [26].

Hadoop, Disco и подобные им проекты отлично выполняют задачу распределенной пакетной обработки больших объемов данных. Фокус на пакетной обработке, в частности, приводит к тому, что вычисления происходят с большой задержкой. Подобные задержки могут быть неприемлемы для целого класса задач, где ответы на вопросы нужно получать незамедлительно.

3.4. Обработка потоков событий в реальном времени

Несколько лет, с момента первого публичного релиза Hadoop, пакетная обработка являлась, пожалуй, единственным способом анализа больших данных. Однако развитие таких приложений как поисковые системы реального времени, высокочастотная торговля и социальные сети диктовало необходимость мгновенно реагировать на новую информацию. Индустрии все больше не хватало «Hadoop реального времени». Эту нехватку восполнил фреймворк S4 (Simple Scalable Streaming System) от компании Yahoo!.

3.4.1 Yahoo! S4

Разработчики S4 писали, что изначально рассматривали возможность адаптировать Hadoop для вычислений на неограниченных потоках событий в

реальном времени, но архитектура Hadoop оптимизирована для пакетной обработки статических данных, что делает создание универсальной системы слишком трудоемким и сложным процессом [27].

Проектируемая система должна была удовлетворять следующим требованиям:

1. Предоставлять простой программный интерфейс поточной обработки данных.

2. Обеспечивать высокую доступность кластера.

3. Минимизировать задержки, используя только оперативную память узлов кластера.

4. Архитектура должна быть децентрализованной и симметричной. Все узлы идентичны, нет единой точки отказа.

Для упрощения первоначальной реализации были введены следующие допущения:

1. Восстановление от ошибок может приводить к потере состояния процессов на данном узле.

2. Динамическое добавление и удаление узлов в кластер невозможно.

Важно отметить, что первое допущение так и оставалось в реализациях S4, вплоть до версии вышедшей в августе 2012 г. Добавление механизма контрольных точек позволило частично решить проблему утери текущего состояния.

Вычисления в S4 естественным образом представляются графом, вершинами которого являются вычислительные элементы (PE), а ребрами — потоки событий. Событие представляет из себя кортеж именованных значений. Именование является важным для реализации группирования потоков. Так как архитектура S4 подразумевает хранение состояния в памяти PE, то часто бывает важно направлять кортежи, удовлетворяющие определенным критериями, на заданные узлы. Достигается это объединением потоков по именам значений в кортежах. Вычислительный элемент, получая на входе события из одного или нескольких потоков, может или создать новый выходной поток или опубликовать результат. Еще одним примитивом S4, который, однако, не является элементом графа, является адаптер. Адаптеры производят преобразование внешних входных/выходных потоков в необходимый формат.

О производительности S4 можно судить по результатам,опубликованным авторами в [27]. В качестве эксперимента приводится приложение производящее расчет CTR различных блоков на странице поисковой выдачи. Целью приложения является определение рекламных блоков с низким CTR и

исключение их дальнейшего показа. В работе утверждается, что производительность кластера из 16 четырехядерных серверов с 2 ГБ памяти начала падать при достижении пропускной способности в 7268 событий/с или 9.7 Мбит/с.

В другой работе [28] было проведено синтетическое нагрузочное тестирование кластера S4. Из эксперимента сделаны следующие выводы:

1. Добавление узлов в кластер не всегда приводит к повышению производительности.

2. Распределение событий по узлам неравномерно, что может привести к общему падению производительности.

3. Механизм обеспечения отказоустойчивости также может приводить к падению производительности. При выходе из строя одного из узлов S4 пытается перераспределить нагрузку на другие узлы, что может привести к их деградации.

4. Для обмена сообщениями используется протокол UDP без программного подтверждения доставки, что может приводить к потере данных.

3.4.2 Storm

Публичный релиз проекта Storm [29] от компании Twitter был сделан примерно год спустя первого публичного релиза S4. Не удивительно, что концептуально фреймворки очень похожи. Однако Storm, как заявляют авторы, устраняют некоторые недостатки продукта Yahoo!.

Основными свойствами Storm являются:

1. Широкий набор вариантов использования. Storm может быть использован для непрерывных вычислений над потоками событий, непрерывного обновления баз данных, распараллеливания сложных вычислений (распределенный RPC) и др.

2. Масштабирование. Storm поддерживает прозрачное горизонтальное масштабирование.

3. Гарантия сохранности данных. Storm, в отличие от S4, гарантирует обработку каждого сообщения.

4. Отказоустойчивость. Если во время вычислений происходит отказ оборудования или возникает ошибка, Storm перераспределяет задания на другие узлы.

5. Независимость от языка программирования. Вычисления можно программировать на любом языке.

Вычисления в Storm представляются графами, называемыми топологиями (рис. 2). Вершины графа определяют вычисления, а ребра создают маршруты передачи данных. Фреймворк определяет два основных вида вершин: труба (spout) и молния (bolt). Трубы, как и следует из названия, являются источниками потоков данных в топологии. По сути они определяют способы получения внешних данных и преобразования их в потоки кортежей. Молнии подключаются к одному или нескольким потокам, производят вычисления и, возможно, создают новые потоки.

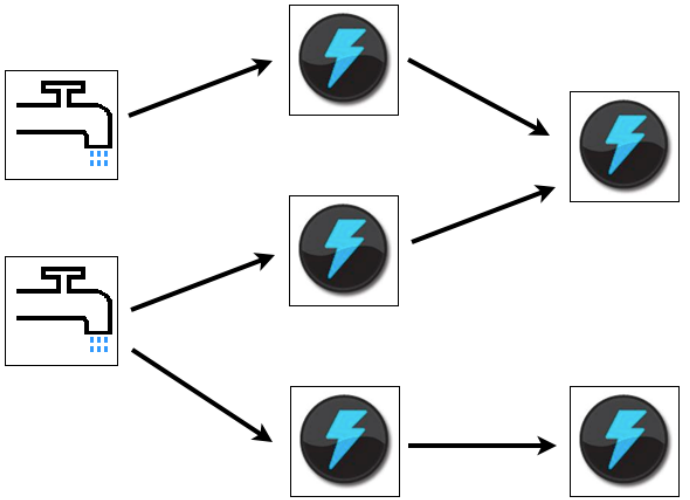

Рис. 2. Топология Storm.

Интересным представляется то, как Storm гарантирует обработку всех событий. Учитывая графовую структуру топологии, каждый кортеж может порождать целое дерево производных кортежей. Storm отслеживает состояние этого дерева и считает первоначальный кортеж полностью обработанным, когда обработан каждый элемент дерева. Если кортеж полностью не обработан по истечению заданного временного интервала, то вычисления повторяются. Этот механизм, правда, может приводить к повторной обработке одних и тех же данных и получению неверных результатов. Решается эта проблема использованием транзакционных топологий, гарантирующих единственность вычислений над каждым событием.

В виду молодости проекта Storm достоверных публикаций о производительности найти не удалось. Однако авторы утверждают, что одно из первых приложений, разработанное с использованием фреймворка

обрабатывало 1 млн сообщений/с на кластере из 10 машин, одновременно делая несколько сотен запросов/с к базе данных.

4. Заключение

В этой работе были рассмотрены проблемы, которые поставили перед реляционными СУБД большие данные. Проанализировав возможные пути решения этих проблем, мы указали на те концептуальные ограничения, которые не позволяют классической реляционной архитектуре справляться со стремительно возрастающим объемом информации. Далее были рассмотрены три подхода к работе с большими данными: NoSQL, MapReduce и обработка потоков событий в реальном времени. Мы обратили внимание на те архитектурные особенности, которые позволяют каждому из них эффективно решать поставленную задачу. Важно отметить, что ни один из представленных подходов не предлагает решения всех возможных задач, которые возникли в контексте больших данных. Каждый из них эффективно решает свой класс задач, позволяя пользователю выбрать наиболее подходящий для него инструмент.

Список литературы

[1] Tom White. Hadoop: The Definitive Guide, 3rd Edition. O'Reilly Media, 2012, 688 p.
[2] Mark A. Beyer, Douglas Laney. The Importance of «Big Data»: A Definition. http://www.gartner.com/DisplayDocument?id=2057415, 21 June 2012.
[3] Carlo Strozzi. NoSQL: A Relational Database Management System.http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home%20Page
[4] Jaroslav Pokorny. NoSQL databases: a step to database scalability in web environment. Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services, p. 278-283, ACM New York, NY, USA, 2011.
[5] Christof Strauch. NoSQL Databases. http://www.christof-strauch.de/nosqldbs.pdf
[6] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber. Bigtable: a distributed storage system for structured data. Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation, vol. 7, p. 15-15, USENIX Association Berkeley, CA, USA, 2006.
[7] Rick Cattel. Scalable SQL and NoSQL data stores. ACM SIGMOD Record, 39(4), p. 12-27, ACM New York, NY, USA, December 2010.
[8] Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung. The Google File System. 19th ACM Symposium on Operating Systems Principles, Lake George, NY, October, 2003.
[9] Jeffrey Dean, Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation, vol. 6, p. 10-10, USENIX Association Berkeley, CA, USA, 2004.
[10] Apache Hadoop. http://hadoop.apache.org/
[11] Apache CouchDB. http://couchdb.apache.org/
[12] MongoDB. http://www.mongodb.org/
[13] Riak. http://basho.com/products/riak-overview/
[14] J. Chris Anderson, Jan Lehnardt, Noah Slater. CouchDB: The Definitive Guide. O'Reilly Media, 2010, 272 p.

[15] Konstantin Shvachko, Hairong Kuang, Sanjai Radia, Robert Chansler. The Hadoop Distributed File System. MSST '10 Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), 2010, pp. 1-10.
[16] Sort Benchmark Home Page. http://sortbenchmark.org/
[17] Ajay Anand. Hadoop Sorts a Petabyte in 16.25 Hours and a Terabyte in 62 Seconds. http://developer.yahoo.com/blogs/hadoop/posts/2009/05/hadoop_sorts_a_petabyte_in_162/, 2009.
[18] Apache Hive. http://hive.apache.org/
[19] Apache Pig. http://pig.apache.org/
[20] Apache Hbase. http://hbase.apache.org/
[21] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: wait-free coordination for internet-scale systems. USENIXATC'10: Proceedings of the 2010 USENIX conference on USENIX annual technical conference. Berkeley, CA, USA: USENIX Association, 2010, pp. 11–11.
[22] Сергей Кузнецов. К свободе от проблемы Больших Данных. «Открытые системы», №02, 2012.
[23] G. Agha. Actors: A Model of Concurrent Computation in Distributed Systems. Cambridge, MA, USA: MIT Press, 1986.
[24] The Disco Project. http://discoproject.org/
[25] Erlang Programming Language. http://www.erlang.org/
[26] Joe Armstrong. Concurrency Oriented Programming in Erlang. http://ll2.ai.mit.edu/talks/armstrong.pdf, November 2002.
[27] Leonardo Neumeyer, Bruce Robbins, Anish Nair, Anand Kesari. S4: Distributed Stream Computing Platform. Data Mining Workshops (ICDMW), 2010 IEEE International Conference, 2010.
[28] Jagmohan Chauhan, Shaiful Chowdhury and Dwight Makaroff, Performance Evaluation of Yahoo! S4: A First Look, IEEE Seventh International Conference on P2P, Parallel, GRID, Cloud and Internet computing, 2012.
[29] Storm: Distributed and Fault-tolerant realtime computation. http://storm-project.net/