Компьютерные инструменты в образовании, 2016
№ 1: 5–16
УДК: 004.652
http://ipo.spb.ru/journal

инженерия программного обеспечения

БАЗЫ ДАННЫХ: СОВРЕМЕННЫЙ ПЕЙЗАЖ В ИСТОРИЧЕСКОЙ ПЕРСПЕКТИВЕ

Новиков Борис Асенович, Графеева Наталья Генриховна,
Михайлова Елена Георгиевна

Аннотация

Проблема обработки и хранения BigData, основную часть которых составляет неструктурированная информация, привела к появлению NoSQL баз данных, которые стремительно завоевали популярность. Одно время даже высказывалось мнение, что традиционные реляционные СУБД обречены. Действительно ли это так? Решают ли новомодные системы те задачи, которые стоят в настоящее время перед системами хранения данных?

Ключевые слова: *СУБД, NoSQL, рейтинг DB-Engines, целостность данных, продолжительность транзакций, согласованность данных, доступность данных, устойчивость к разделению.*

Если бы геометрические аксиомы задевали интересы людей, они бы опровергались.
Томас Гоббс

С чего собственно начинались базы данных? Изначально задача создания систем, управляющих базами данных (СУБД), даже не стояла. Зато существовали прикладные задачи, требующие обработки больших (или относительно больших) объемов данных [2]. Разработчики, которым довелось создавать не одно и не два приложения, постепенно стали обращать внимание на то, что в разных по своей сути приложениях им приходится программировать одну и ту же функциональность, связанную с накоплением, поиском и анализом данных. К тому же в середине 60-х годов прошлого столетия появились устройства прямого доступа относительно большой емкости, то есть появились устройства, в которых можно было размещать данные любой структуры и организовывать к ним прямой доступ. Однако доступ к таким структурам требовал (и продолжает требовать) достаточно изощренного программирования, который оказывался по зубам только высококвалифицированным программистам. Но таких специалистов значительно меньше, чем создаваемых приложений. Как следствие, далеко не все приложения представляли собой качественный программный продукт. Кроме того, многократная реализация очень похожей функциональности (причем не всегда удачная) значительно удорожала конечный программный продукт. На этом фоне и возникла идея разграничения функций приложения и функций обработки данных. Появилась идея о создании централизованных систем, обеспечивающих доступ к данным, то есть идея о создании высокоэффективных и качественных универсальных систем, ориентированных не на нужды отдельных приложений, а исключительно на обработку данных. Такого рода систе-

мы могут быть созданы однажды. Для их создания можно использовать программистов высшей квалификации, а сами системы в дальнейшем использовать многократно при создании разнообразных приложений. Эта идея позволяла существенным образом повысить качество и снизить стоимость дальнейшей разработки приложений. В 1969 году сформулировали основные принципы таких систем, которые были впервые продекларированы в документе CODASYL DBTG Report и на долгие годы определили направление развития систем, управляющих базами данных. Ниже изложены основные идеи этих принципов.

Цели создания систем управления данными:

- Снижение стоимости разработки и повышения качества приложений за счет вынесения общей функциональности в СУБД.
- Централизация управления хранением и доступом к данным.
- Поддержка сложных логических структур.

Функции систем управления данными:

- Обеспечение независимости данных и приложений.
- Обеспечение целостности данных (Integrity).
- Поддержка согласованности данных (Consistency).
- Защита от несанкционированного доступа.
- Разграничение прав доступа.
- Поддержка высокоуровневых эффективных языков запросов.

В современных системах управления данными значение и содержание многих из этих функций существенно изменилось. Некоторые их этих функций полностью или частично перешли к другим типам прикладных систем, важность и наличие других систем зависят от архитектуры прикладной информационной системы и типа используемой СУБД. Однако в целом нельзя не отметить, что этот список действительно определил направление развития систем баз данных на много лет вперед. Попытаемся перечислить основные функции современных СУБД и начнем с функциональности, предусмотренной этим списком.

Обеспечение независимости данных и приложений изначально рассматривалось как важнейший элемент систем управления. Те принципы, которые понимаются под независимостью данных и приложений, разбиваются на три возможные группы:

- Одни и те же данные могут использоваться для различных приложений.
- Появление новых требований к данным (например, добавление новых полей, таблиц, бизнес-логики поведения данных и т. п.) не должно оказывать влияния на работу существующих приложений.
- Допустимо асинхронное внедрение новых версий приложений (последний принцип особенно актуален для «толстых» клиентов).

Для поддержки соответствующих требований были, в частности, введены языки, декларирующие описание структуры и бизнес-логики поведения данных; словари, представляющие соответствующую информацию, хранимые в системах управления данными, и т. п. Словари данных и языки описания структуры и бизнес-логики данных в том или ином виде включены почти во все современные традиционные СУБД. Большинство систем строго придерживаются принципов независимости данных, однако следует отметить, что последнее время стали появляться СУБД, отступающие от этих, казалось бы, безоговорочно полезных принципов. Нарушение принципов независимости первой из упомянутых выше групп означает, что появляются хранилища данных, узко ориентированные на специфику данных и функциональность одного приложения. Это могло бы

показаться очень нерациональным, если бы не то обстоятельство, что такие приложения носят эксклюзивный характер и востребованы миллионами пользователей. Нарушение принципов независимости из второй и третьей групп очевидным образом сказывается на разработке, сопровождении и внедрении приложений, работающих с такими системами.

Кроме соответствия структуре описанных данных, система управления данными должна обеспечивать выполнение ряда условий. Этим условиям должны удовлетворять хранимые данные (например, отметка за экзамен студента должна быть в интервале от 1 до 5; предмет, по которому ставится отметка, должен присутствовать в списке читаемых курсов и т. п.). Такого рода условия называют ограничениями целостности (Integrity), и они также описываются с помощью специальных языков описания данных. За проверку выполнения заданных ограничений отвечает СУБД. Выполнение любой операции, пытающейся нарушить эти ограничения, отвергается. Типовыми ограничениями целостности, которые на сегодняшний день присутствуют практически в большинстве крупных СУБД, являются ограничения ссылочной целостности, запрет на неопределенные значения, запрет повторяющихся значений и контроль возможного диапазона значений атрибутов. Однако существуют СУБД, в которых правила целостности либо вообще отсутствуют, либо присутствуют лишь частично. При этом следует отметить, что если сама природа данных такова, что требует учета каких-то ограничений, то эти ограничения все равно придется учитывать на каком-либо уровне, и если нет возможности учесть их на уровне СУБД, то разработчики будут вынуждены их учитывать на уровне создаваемого приложения.

Поддержка согласованности (Consistency) связана с состояниями данных. Некоторые состояния базы данных называются согласованными. Задача СУБД — обеспечение механизмов, позволяющих переводить данные из одного согласованного состояния в другое. Набор операций, переводящих базу из одного согласованного состояния в другое, принято называть транзакцией. Поддержка согласованности данных со стороны СУБД означает, что после удачного или неудачного завершения работы приложения база данных в любом случае окажется в согласованном состоянии, то есть все транзакции приложения либо будут выполнены полностью, либо не оставят никаких следов в данных (то есть СУБД обеспечит откат не полностью завершенных транзакций). Это требование называют атомарностью транзакций (Atomicity). После фиксации транзакции изменения в данных становятся постоянными. Такое свойство называют продолжительностью транзакций (Durability). Поддержка согласованности существенно усложняется, если СУБД допускает параллельное (или конкурентное) выполнение транзакций. Для обеспечения этой задачи в традиционных СУБД были разработаны разнообразные изощренные алгоритмы исполнения транзакций, создающие для пользователей иллюзию изолированности (Isolation) работы с данными. Про СУБД, поддерживающие все эти свойства, говорят, что они обладают ACID-свойствами. Однако в действительности за этими понятиями могут скрываться существенно различающиеся уровни изолированности. Согласно стандарту SQL-92 различают следующие уровни изолированности:

- *Read uncommitted*
- *Read committed*
- *Repeatable read*
- *Serializable*

Read uncommitted обеспечивает самый слабый уровень изоляции, допускающий чтение данных незафиксированных транзакций — так называемое «грязное» чтение.

Serializable обеспечивает самый сильный уровень изоляции, каждый последующий включает все предыдущие. Большинство традиционных СУБД по умолчанию используют уровень *Read commited*. На этом уровне изоляции обеспечивается защита от «грязного» чтения, однако при повторном чтении одного набора данных результаты могут оказаться разными, если в промежутке между чтениями данные были изменены успешно зафиксированной транзакцией. Уровень *Repeatable read* не позволяет изменять данные, считанные активной транзакцией, но не препятствует добавлению новых данных другими транзакциями. Самый сильный уровень изоляции (*Serializable*) согласно стандарту должен обеспечивать абсолютную иллюзию автономной работы (как будто других транзакций не существует). Только этот уровень изоляции обеспечивает защиту от так называемого «фантомного чтения». В действительности некоторые промышленные СУБД вместо уровня *Serializable* используют уровень *Snapshot Isolatio*n, который несколько слабее *Serializable*, но сильнее *Repeatable read*. Транзакция, работающая на данном уровне, видит только те изменения данных, которые были зафиксированы до её запуска, то есть ведёт себя так, как будто получила при запуске моментальный снимок базы данных и работает с ним. Очевидно, что поддержка уровня изоляции *Serializable* наиболее сложна, и именно поэтому по умолчанию в большинстве систем используются более низкие уровни, хотя системы с уровнем изоляции *Serializable* и конкурентным выполнением миллиардов транзакций также существуют.

Алгоритмы, обеспечивающие поддержку упомянутых выше уровней изолированности, блестяще справляются со своими задачами. Трудно представить, что когда-то такие задачи нужно было решать силами разработчиков разнообразных приложений. Однако в последнее время стали появляться СУБД, которые пытаются себя противопоставить традиционным СУБД и в качестве от одного из принципов выдвигают отказ от уровня изоляции *Serilizable* [13]. Такой отказ выглядит, по крайней мере, странным на фоне того, что лишь незначительная часть приложений, использующих традиционные СУБД, действительно использует уровень изоляции *Serializable*.

Функции защиты данных от несанкционированного доступа и разграничение доступа является перманентной задачей практически любой СУБД. С этой целью были разработаны разнообразные приемы, позволяющие скрывать реальные данные от пользователей (например, представления), разграничивать права доступа (роли, привилегии, мандатный доступ), а в последних версиях некоторых промышленных СУБД и разнообразные приемы для отображения, а иногда и намеренного искаженного отображения данных (ORACLE 12C). Еще одно перспективное направление развития защиты данных — изоляция администраторов серверов баз от собственно данных. В большинстве сегодняшних СУБД администраторы серверов базы могут видеть и изменять все данные в БД. Новые средства защиты позволяют администраторам выполнять все операции по администрированию БД, но не позволяют видеть и менять данные (ORACLE 12C). Однако следует отметить, что, несмотря на изобилие механизмов защиты данных в рамках СУБД, во многих сегодня создаваемых приложениях реализована двухуровневая защита данных, смысл которой сводится к тому, что на первом уровне проверяются права конечных пользователей на уровне приложения, и только на следующем уровне — права унифицированного пользователя на уровне СУБД. Более того, в последнее время наметилась тенденция по созданию трехуровневой защиты данных, которая начинается с проверки пользователей на уровне операционной системы. Вынос механизмов управления пользователями из СУБД и приложений в единую централизованную систему организации в большинстве случаев действительно упрощает управление пользователями в масшта-

бах организации и позволяет в значительной степени автоматизировать весь процесс управления.

Наличие высокоуровневого и эффективного языка запросов является, пожалуй, наиболее важной отличительной чертой СУБД. С этой целью в рамках разнообразных систем управления были реализованы высокоуровневые декларативные языки для манипуляций над данными, из которых особого внимания заслуживает SQL. Элегантность и независимость от специфики СУБД, а также его поддержка ведущими производителями баз данных сделали язык SQL основным стандартом для обработки данных. Но одной элегантности языка явно недостаточно для обеспечения его эффективности (быстрого и не слишком ресурсоемкого выполнения запросов). Поэтому практически во всех СУБД, появились так называемые оптимизаторы запросов. Основная функция оптимизатора запросов состоит в построении возможных планов выполнения запросов и выборе оптимального из них. Как правило, наилучшим планом является тот, на выполнение которого необходимо меньше времени, однако в некоторых случаях критерии могут зависеть от требований приложения. Например, требуется минимизировать время получения первых строк запроса или время полного исполнения запроса. Оптимизаторы выбирают планы на основе явно или неявно определенной функции стоимости. В зависимости от того, как определена функция стоимости, различают два режима работы оптимизаторов: оптимизатор, работающий на основе правил, и оптимизатор, работающий на основе статистических характеристик фактически хранимых данных. В настоящее время алгоритмы оптимизации достаточно хорошо проработаны и реализованы в промышленных СУБД, и только в достаточно редких случаях требуется ручная настройка (с помощью подсказок оптимизатору) или полное переписывание запросов с целью улучшения их производительности. Казалось, что в этом направлении возможно только движение вперед, то есть появление новых декларативных конструкций, новые приемы оптимизации и т. п. Однако в последнее десятилетие стали активно появляться так называемые NoSQL–системы, которые отказываются от многих принципов, заложенных в основе ранее появившихся СУБД, а некоторые из них отказываются даже от декларативных языков запросов [9]. Отказ от использования высокоуровневых языков запросов приводит к достаточно очевидным последствиям: разработчики приложений вынуждены формулировать низкоуровневые запросы, требующие значительно больше времени на этапе разработки, что, в конечном счете, приводит к растягиванию и удорожанию всего периода создания приложения. Разработчик, оказавшийся в таком положении, напоминает автогонщика, которому вместо скоростного болида предложили самокат. По сути дела у разработчиков приложений есть два пути: использовать дорогую СУБД с декларативным языком запросов и быстро создать приложение или использовать недорогую (или вообще бесплатную) СУБД с низкоуровневым языком запросов и потратить значительно больше времени (и средств) на создание приложения. У каждого из этих путей есть свои плюсы. Первый обеспечивает гарантированное качество запросов приложения при небольших затратах на разработку, второй — занятость большого числа разработчиков. Как бы это ни показалось странным, первый из упомянутых выше плюсов не для всех компаний является достаточно убедительным. Сегодня в IT-индустрии есть ряд компаний, услугами которых пользуются миллионы пользователей во всем мире. Такие компании могут позволить себе огромный штат высококвалифицированных разработчиков. Приложения, создаваемые такими компаниями, зачастую используют системы управления данными с низкоуровневыми языками запросов. Успех, сопутствующий этим системам, создает иллюзию простоты создания приложений. Но на самом

деле за этими приложениями стоит труд высококвалифицированных разработчиков, тестеров, администраторов, аналитиков и т. п. Разработчикам из небольших компаний следует отдавать себе отчет в том, что формула создания таких приложений носит эксклюзивный характер и не масштабируется.

Какое-то время назад еще можно было классифицировать СУБД по типам поддерживаемых моделей данных: реляционных, иерархических, объектно-ориентированных, XML и т. д. На сегодняшний день такая классификация имеет смысл только для некоторых NoSQL систем, имеющих, как правило, узкую специализацию. В то же время большинство крупных производители СУБД, которые первоначально позиционировались как реляционные (например, ORACLE и DB2) декларируют и действительно поддерживают разнообразные модели данных, включая неструктурированные. Но каковы бы ни были особенности модели той или иной СУБД, соответствующая ей база данных состоит из объектов. А значит, одной из важнейших функций современной СУБД является идентификация объектов. Можно выделить основные классы методов идентификации:

- *Идентификация по свойствам.* Использует значения некоторых атрибутов идентифицируемых объектов. Этот класс методов идентификации хорош тем, что он очень естественен по своему содержанию и напоминает идентификацию объекта в естественной среде по приметам. Например, идентификация личности по отпечаткам пальцев.
- *Позиционная идентификация.* Основана на информации о положении объекта в пространстве или по отношению к другим объектам. К этому классу можно отнести географические координаты, любые адреса, а также относительные указания (например, «сразу после третьего моста»).
- *Суррогатная идентификация.* Основана на присвоении объекту искусственных атрибутов, не связанных с его свойствами. Такой идентификатор генерируется при первом появлении объекта в системе и в дальнейшем никогда не изменяется.

Другой аспект обработки данных — способ обзора (поиска данных). В современных системах различают два вида обзора:

- Навигация по ссылкам.
- Ассоциативный поиск по значениям.

Последний способ обзора также является очень естественным по своей природе и характерен для традиционных систем, в которых можно осуществлять поиск объекта (или объектов) при наличии сведений о некоторых его признаках. Например, найти всех рыжих котов с зелеными глазами.

Еще одним аспектом обработки данных выступает вид обработки, то есть какие единицы данных могут быть обработаны с помощью одной операции. Различают два вида обработки:

- Обработка отдельных объектов.
- Массовая (bulk) обработка.

К примеру, язык SQL изначально ориентирован на массовую обработку данных. С помощью одной операции можно обработать большое количество строк. Более того, особенностью всех популярных реализаций SQL в рамках традиционных СУБД является то обстоятельство, что массированные операции обработки выполняются значительно эффективнее обработки единичных объектов, то есть эффективнее обработать с помощью

одной операции 10000 объектов базы, чем 10000 раз выполнять обработку отдельных объектов. Но следует отметить, что системы, в которых языки запросов ориентированы на обработку отдельных элементов, также существуют и, более того, активно развиваются и обсуждаются.

В последнее десятилетие появилось новое направление развития систем управления данными с достаточно амбициозным названием NoSQL. Эти системы позиционируют себя как альтернативные системы, которые в состоянии хранить и обрабатывать гигантские объемы данных [11]. Из-за большого разнообразия существующих сегодня NoSQL систем трудно понять, что они на самом деле из себя представляют, и, тем более, дать рекомендации по использованию их для конкретных прикладных задач. Попробуем описать это явление подробнее с примерами реально существующих систем. Большинство систем NoSQL создавались с расчетом на легкое масштабирование и работу на кластерах из недорогих компьютеров в облачной среде. Как известно, поддержка транзакций и строгой согласованности данных в условиях распределенных систем требует значительного числа синхронных взаимодействий. Это обстоятельство не только снижает время отклика системы, но и приводит к уменьшению ее отказоустойчивости. Кроме того, есть ряд задач, когда объективно нет необходимости в поддержке транзакций. Например, аналитические приложения, в которых нет регулярных обновлений и для которых информация, поступающая в последние часы, минуты и секунды, не является существенной. Казалось бы, что последнее обстоятельство является достаточным аргументом для создания систем без поддержки транзакций. Но на практике все сложилось иначе. На фоне обсуждения проблем с производительностью и отказоустойчивостью появилось эмпирическое утверждение Эрика Брюэра, опубликованное в статье [4], которое впоследствии стали называть «теоремой CAP» (Consistence, Availability, Partition tolerance — согласованность, доступность, устойчивость к разделению). Это утверждение гласит: «Распределенная система не может гарантировать одновременно доступность при сбоях узлов, согласованность данных и устойчивость к разделению сети». Никакого математического доказательства утверждения ни в оригинальной, ни в какой другой статье нет, а следовательно, строго говоря, нет и теоремы. И, собственно, термин <теорема CAP> стал использоваться как эффектное название для эмпирического утверждения и не более того. Некоторые разработчики новых систем, возможно, даже не прочитали оригинальную статью Эрика Брюэра, но использовали термин «теорема САП» как формальное обоснование для создания всевозможных хранилищ данных, основанных на компромиссах между упомянутыми тремя свойствами. На сегодняшний день трудно найти какую-нибудь статью на тему NoSQL хранилищ, в которой не упоминается «теорема САП». Более того, на основе «теоремы CAP» даже возникла одна из возможных классификаций новых систем [2]:

- CA — удовлетворяет требованиям по согласованности и доступности,
- CP — удовлетворяет требованиям по согласованности и устойчивости к разделению,
- AP — удовлетворяет требованиям по доступности и устойчивости к разделению.

Разумеется, ни в одном из упомянутых выше случаев не будет выполнено свойство ACID (Atomicity, Consistency, Isolation, Durability), в той или иной мере соблюдаемое в приложениях, основанных на традиционных реляционных СУБД. Тем не менее, именно проблемы, существующие на тот момент в некоторых традиционных СУБД в управлении с большими объемами данных, и появление новых методов управления распределенны-

ми системами на волне <теоремы CAP> привели к возникновению нового класса систем, который впоследствии был назван NoSQL.

Происхождение термина NoSQL приписывается Йохану Оскарсону (Johan Oskarsson), который использовал его на конференции по нереляционным базам данных в 2009 году [11]. Сам термин NoSQL понимается как <Not Only SQL>. Термин используется в качестве главной классификации, подразумевающей большое разнообразие хранилищ данных, многие из которых не основаны на реляционной модели данных, в том числе узко ориентированных на очень специфические модели данных, например, графы. Несмотря на разнообразие решений NoSQL, следующий набор характеристик часто связывают с этим классом систем [7]:

- Простые и гибкие не реляционные модели данных, предназначенные для широкого круга задач. Современные NoSQL модели данных принято делить на четыре категории [7], а именно: хранилища типа ключ-значение, документные хранилища, колоночные хранилища и граф-ориентированные.
- Возможности масштабирования по горизонтали. Некоторые хранилища обеспечивают масштабирование для хранения данных, другие концентрируются на масштабировании чтения/записи данных.
- Обеспечение высокой степени доступности данных. Многие NoSQL хранилища данных ориентированы на распределенные сценарии использования данных. Для обеспечения высокой степени доступности данных эти системы предпочитают пожертвовать согласованностью данных в пользу доступности, что приводит к появлению решений типа AP(Availability, Partition tolerance).
- Системы NoSQL, как правило, не поддерживают ACID транзакций, как это предусмотрено в традиционных СУБД. Их иногда называют BASE–системами (*Basically Available, Soft state, Eventually consistent*) [8]. В этом названии *Basically Available* означает, что данные доступны всегда, когда к ним происходит обращение, даже если часть из них в настоящий момент недоступна; *Soft state* означает, что данные могут находиться в рассогласованном состоянии какой-то период времени, а *Eventually consistent* означает, что, в конечном счете, после некоторого периода времени данные в хранилище окажутся в согласованном состоянии. Тем не менее, некоторые NoSQL хранилища данных, например CouchDB [3] обеспечивают поддержку ACID транзакций.

Для многих систем этого класса при обработке данных характерно использование технологии MapReduce [8]. Согласно этой технологии, обработка данных состоит из двух шагов — Map и Reduce. На шаге Map происходит предварительная обработка данных, на шаге Reduce — свертка полученных на предыдущем шаге результатов. Основное преимущество технологии MapReduce — возможность продолжать работу при сбоях в оборудовании, то есть высокая отказоустойчивость. Эта технология особенно эффективно работает в тех задачах, где абсолютная точность не нужна.

Модели данных и методы работы с данными систем класса NoSQL чрезвычайно разнообразны. Их характерной особенностью является ориентация на узкий класс задач. Перечислим основные категории NoSQL хранилищ [7] и укажем наиболее популярных представителей с описанием их основных характеристик.

Наиболее популярными представителями систем категории класс–значение, согласно рейтингу DB-Engines Ranking [5], на момент написания статьи являются: Redis, Memcached, Amazon DynamoDB (мультимодельная система), Riak. Утверждается, что эти

системы позволяют добиваться высокой производительности, могут легко масштабироваться, но эффективный поиск в таких моделях возможен только по уникальному ключу. При этом сам запрашиваемый объект может быть как последовательностью байтов, так и более сложной структурой. Вызывает некоторое сомнение утверждение про высокую производительность, которая декларируется разработчиками таких систем, так как для объективного сравнения производительности систем категории класс-значение и традиционных СУБД необходимо произвести сравнение на основе сопоставимых операций. Операции, выполняемые в традиционных СУБД с помощью одного высокоуровнего запроса, в системах типа класс-значение могут потребовать десяток, а то и сотню низкоуровневых операций. Сравнение на основе мелких операций типа ключ-значение представляется не совсем справедливым, так как сильной стороной высокоуровневых запросов является именно массированная, а не единичная обработка данных. Авторам этой статьи не удалось найти ни результатов, ни предложений о самой технологии сравнения производительности для такого рода систем, а следовательно, нет объективных оснований говорить о том, что производительность систем категории ключ-значение выше производительности традиционных СУБД. Кроме того, серьезным недостатком этих систем является полное отсутствие отношений между сущностями. Это означает, что система управления хранилищем не может проконтролировать целостность отношений, и соответствующая функциональность целиком ложится на приложения. Тем не менее, для некоторого класса задач этого вполне достаточно, и такие хранилища используются весьма эффективно (например, для хранения изображений, видеороликов, файлов и т. п).

Среди систем, ориентированных на обработку документов, согласно рейтингу DB-Engines Ranking, заслуживают упоминания: MongoDB, CouchDB, CouchBase и уже упомянутая выше мультимодельная система Amazon DynamoDB. В рамках документарной модели эти системы хранят объекты (документы) в формате JSON (Java Script Object Notation) или BSON (Binary JSON). Форматы JSON и BSON допускают атрибуты простых типов, массивы, а также вложенные объекты. Эти системы поддерживают индексы на полях документов и позволяют строить сложные запросы. Полноценных ACID-транзакций в них также нет. Тем не менее, операции обновления на уровне одного документа обычно являются атомарными. Такие хранилища данных эффективно применяются в системах управления содержимым, издательском деле, документальном поиске и т. п.

Особое место среди NoSQL — систем занимают колоночные хранилища данных. Их также принято относить к NoSQL. Группу колоночных хранилищ, согласно рейтингу DB-Engines Ranking, на момент написания статьи возглавляют: Cassandra, HBase, Accumolo, Hypertable. Основная особенность хранилищ этого типа заключается в том, что данные представляются в виде таблиц, а хранение и фрагментация этих данных возможны не по строкам, как это принято в традиционных СУБД, а по столбцам. Кроме того, для многих систем этого класса характерно наличие SQL-подобных языков высокого уровня. Как известно, основными принципами реляционных СУБД являются: представление данных в виде таблиц и наличие языков высокого уровня. При этом не имеет значения, как данные хранятся. Следовательно, колоночные хранилища – это реляционные СУБД, которые используют систему хранения и фрагментации данных, отличающуюся от системы хранения традиционных реляционных СУБД (ORACLE, DB2, Postgress, Mycrosoft SQL Server, MySQL и т. п.).

Системы, ориентированные на обработку графов, специально предназначены для хранения узлов графов и связей между ними. Как правило, такого рода системы поз-

воляют задавать для узлов и связей еще и набор произвольных атрибутов и выбирать узлы и связи по этим атрибутам. Кроме того, системы поддерживают алгоритмы обхода графов и построения маршрутов. Согласно рейтингу DB-Engines Ranking, список граф-ориентированных хранилищ возглавляют: Neo4G, OrientDB, Titan, ArangoDB. При этом системы OrientDB и ArangoDB также позиционируются как мультимодельные. Граф-ориентированные хранилища эффективно используются для задач, связанных с анализом социальных сетей, выбором маршрутов и т. п.

Достаточно часто NoSQL системы позволяют задавать различные конфигурации кластера и тем самым добиваться требуемых свойств приложения (например, допускать или не допускать чтение со вторых реплик, задавать возможное количество реплик и т. п.). Кроме того, если в системах поддерживается разделение данных (sharding), то крайне важным предстает правильный выбор ключей, по которым будет происходить распределение данных между узлами системы для балансировки нагрузки.

Итак, в NoSQL движении присутствует не только большое разнообразие моделей (включая реляционную), способов обработки, но и множество тонко настраиваемых вариантов конфигурации, существенным образом влияющих на свойства систем. А что же традиционные реляционные СУБД? Неужели они так морально устарели, как говорят об этом некоторые авторы? Разумеется, нет! Реляционные базы данных существуют уже более 30 лет. За это время в IT индустрии вспыхивали несколько революций, которые намеревались положить конец реляционным хранилищам данных. Однако время показало, что ни одна из этих революций не состоялась и не уничтожила реляционные базы данных. Время от времени каждая новая технология для работы с базами данных провозглашается «новым открытием», которое уничтожит реляционные БД. В девяностых это были объектно-ориентированные базы данных, в двухтысячных — XML хранилища. Сейчас это целый ряд новых технологий, объединившихся под флагом NoSQL. Но что в действительности происходит, когда появляются такие технологии? Разработчики баз данных изучают новые технологии, оценивают их жизнеспособность, а затем лучшие технологические решения внедряют в традиционные СУБД. Присмотритесь к последним версиям! Там уже давно присутствует разнообразие моделей данных (реляционная, объектно-ориентированная, XML, JSON, ключ-значение и т. п), совершенствуются методы обработки данных, появляются приемы для распараллеливания и т.д. Последние версии ориентированы на работу в облаках со всеми необходимыми для этого характеристиками. Традиционные СУБД уже давно не являются только реляционными СУБД, как это было когда-то, и если их по-прежнему так называют, то это всего лишь дань традиции и не более того. О популярности традиционных СУБД убедительно говорит все тот же DB-Engines Ranking, который на момент написания статьи выдает следующее (см. рис 1).

Несмотря на растущую популярность NoSQL движения, представляется, что реляционные базы данных не теряют своей значимости и востребованности на современном рынке. Но параллельно с реляционными СУБД будут использоваться и NoSQL системы. Для различных потребностей будут использоваться разные хранилища данных. Сама природа данных, предполагаемые объемы и решаемые задачи подскажут, что выбрать для реализации.

Rank Jan 2016	Rank Dec 2015	Rank Jan 2015	DBMS	Database Model	Score Jan 2016	Score Dec 2015	Score Jan 2015
1.	1.	1.	Oracle	Relational DBMS	1496.08	-1.47	+56.92
2.	2.	2.	MySQL	Relational DBMS	1299.26	+0.72	+21.75
3.	3.	3.	Microsoft SQL Server	Relational DBMS	1144.06	+20.90	-54.55
4.	4.	↑ 5.	MongoDB +	Document store	306.03	+4.64	+55.13
5.	5.	↓ 4.	PostgreSQL	Relational DBMS	282.40	+2.31	+27.91
6.	6.	6.	DB2	Relational DBMS	196.37	+0.24	-3.76
7.	7.	7.	Microsoft Access	Relational DBMS	134.04	-6.17	-5.10
8.	8.	8.	Cassandra +	Wide column store	130.95	+0.11	+32.20
9.	9.	9.	SQLite	Relational DBMS	103.74	+2.89	+7.54
10.	10.	10.	Redis +	Key-value store	101.16	+0.62	+6.92

Рис. 1. Наиболее популярные СУБД в рейтинговом списке DB-Engines Ranking

Список литературы

1. *Кузнецов С., Посконин А.* Возможно ли сотрудничество SQL и NoSQL? // Открытые системы, 2013. № 9. С. 38–41.
2. *Селезнев К.* От SQL к NoSQL и обратно / / Открытые системы, 2012. № 02. С. 25–29.
3. Apache CouchDB / http://couchdb.apache.org/ (дата обращения 11.01.2016).
4. *Brewer E.* Towards Robust Distributed Systems // ACM Symposium on the Principles of Distributed Computing, Portland, Oregon, 2000.
5. DB-Engines Ranking / http://db-engines.com/en/ranking/ (дата обращения 11.01.2016).
6. *DeCandia G, Hastorun D, Jampani M, Kakulapati G, Lakshman A, Pilchin A, Sivasubramanian S, Vosshall P., Vogels W.* Dynamo: Amazon's highly available Key-value store // ACM SIGOPS Operating Syst. Rev., 2007. 41: 205.
7. *Hecht R., Jablonski S.* NoSQL evaluation: A use case oriented survey // Proc 2011 Int. Conf. Cloud Serv. Computing. P. 336–341.
8. *Jeffrey Dean and Sanjay Ghemawat.* MapReduce: Simplified Data Processing on Large Clusters // OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.
9. *Katarina Grolinger, Wilson A Higashino, Abhinav Tiwari andMiriam AM Capretz.* Data management in cloud environments: NoSQL and NewSQL data stores // Journal of Cloud Computing: Advances, Systems and Applications, 2013. Vol. 2(22).
10. *Murty J.* Programming Amazon Web Services: S3, EC2, SQS, FPS, and SimpleDB // O'Reilly Media, Inc., 2008.
11. NOSQL meetup. San Francisco: Eventbrite. http://nosql.eventbrite.com/ (дата обращения 11.01.2016).
12. *Pritchett D.* BASE: An ACID Alternative. Queue // 2008— 6:48.
13. *Vogels W.* Eventually Consistent // Communication of the ACM. January, 2009. Vol. 52, № 1. P. 40–44.

DBMS: MODERN LANDSCAPE IN HISTORICAL PERSPECTIVE

Novikov B.A., Grafeeva N.G., Mikhailova E.G.

Abstract

The problem of storage and processing of big data, the main part of which presented unstructured information, led to the development of NoSQL databases, which have rapidly gained popularity among researchers and users. Some of them once even expressed an opinion that traditional relational databases would become obsolete. Is it indeed so? Would newfangled NoSQL DBMS respond to the challenges currently facing the data management systems? In the present paper we are addressing these issues.

Keywords: *DBMS, NoSQL, DB-Engines Ranking, Integrity, Durability, Consistence, Availability, Partition tolerance.*

Новиков Борис Асенович,
профессор, заведующий кафедрой ИАС СПбГУ,
borisnov@acm.org

Графеева Наталья Генриховна,
доцент кафедры ИАС СПбГУ,
N.Grafeeva@spbu.ru

Михайлова Елена Георгиевна,
доцент кафедры ИАС СПбГУ,
E.Mikhaylova@spbu.ru