Транзакционные параллельные СУБД: новая волна

С.Д. Кузнецов
kuzloc@ispras.ru

Аннотация. Возможность построения неограниченно масштабируемых кластерных систем привела к резкой активизации исследований и разработок архитектур систем управления данными без совместного использования ресурсов. Образовались два основных фронта: "NoSQL", где отрицаются основные принципы, свойственные СУБД, и "один размер непригоден для всех", где упор делается на специализацию систем при сохранении важнейших свойств СУБД. Особенно интересным является противостояние этих фронтов в области "транзакционных" систем управления данными. Опираясь на "теорему" CAP Эрика Брювера (Eric Bruwer), представители лагеря NoSQL отказываются от обеспечения в своих системах традиционных свойств ACID в транзакциях баз данных. В этой статье обсуждается суть "теоремы" Брювера и обосновывается, что она не имеет отношения к свойствам ACID. Рассматриваются наиболее интересные современные исследовательские работы, обеспечивающие классические ACID-транзакции в параллельных средах без общих ресурсов, а также наиболее здравые подходы, в которых из чисто прагматических соображений свойства ACID частично ослабляются (но совсем не в связи с "теоремой" CAP).

Ключевые слова: транзакционные массивно-параллельные СУБД, "один размер непригоден для всех", NoSQL, ACID-транзакции.

1. Введение

Происходящие важные изменения в области компьютерных аппаратных средств, а именно, возможность сравнительно дешевого построения неограниченно горизонтально масштабируемых кластерных систем (будь то системы, основанные на использовании публичных или частных облачных инфраструктур, или кластеры, конфигурируемые традиционным образом) привели к резкой активизации исследований и разработок пригодных для использования в таких средах архитектур систем управления данных без совместного использования ресурсов (shared nothing). Работы ведутся на двух основных фронтах. (Я не буду здесь говорить про компании-производители основных SQL-ориентированных СУБД, которые всегда стараются решать все проблемы за счет своих собственных, накопленных в течение десятилетий возможностей.)

Первый фронт составляют основные поставщики Internet-услуг, такие как компании Google, Yahoo!, Facebook, Amazon и т.д., которые для своих собственных нужд и для обеспечения публично доступных "облачных" служб разрабатывают средства управления данными, идеологически и архитектурно отходящие от традиционных канонов сообщества баз данных. Во многом именно с деятельностью этих компаний связано возникновения понятия "NoSQL", т.е. (в исходном своем смысле) отказа от существующих решений.

На втором фронте, с моей точки зрения, находятся последователи концепции Майкла Стоунбрейкера (Michael Stonebraker) "один размер непригоден для всех" [1], в которой, по сути, основными являются два соображения: (a) прошло время универсальных систем управления базами данных, пригодных для поддержки приложений всех возможных разновидностей, и (b) при разработке новых систем необходимо пользоваться всеми полезными технологиями и идеями, накопленными в сообществе баз данных. К этому лагерю относятся исследовательские группы ряда университетов США, компании VoltDB, Vertica, Asterdata и т.д.

Следует обратить внимание на то, что представители обоих фронтов сходятся по первому пункту, т.е. их объединяет желание создать новые средства управления данными, специализированные для некоторых классов приложений и поддерживающие эти приложения более эффективно, чем универсальные СУБД. Второй пункт их разделяет: люди из первого лагеря считают (немного утрированно), что технологии баз данных являются тяжким грузом прошлого, а второй лагерь относится к ним, как к ценному наследию, жертвовать которым нельзя.

На обоих фронтах ведутся работы в двух наиболее важных направлениях управления данными – аналитические системы управления данными (т.е. системы, пригодные для построения над ними приложений категории OLAP (online analytical processing, оперативная аналитическая обработка данных)) и транзакционные системы управления данными (т.е. системы, пригодные для построения приложений категории OLTP (online transaction processing, оперативная обработка транзакций)). В первом направлении представителей двух рассматриваемых лагерей в прошлые годы разделяло, прежде всего, отношение к NoSQL-технологии анализа данных *MapReduce* [2]. Не так давно представители лагеря NoSQL считали, что MapReduce заменит в динамических кластерных архитектурах параллельные аналитические системы баз данных, а исследователи из второго лагеря обвиняли создателей MapReduce к возврату в дремучее прошлое, когда технология баз данных не существовала [3-4]. Однако это время, как кажется, миновало. По крайней мере, сообщество баз данных приняло технологию MapReduce и научилось ее использовать [5], и я (может быть, временно) считаю эту тему закрытой (в [5] можно найти много полезных ссылок на эту тему).

1.1. Цель статьи

Цель этой статьи состоит в том, чтобы разобраться в происходящем в области средств управления данными категории OLTP. В общих словах, ситуацию можно охарактеризовать следующим образом. Значительную часть современных Internet-приложений составляют *транзакционные* приложения (онлайновые магазины, аукционы, системы резервирования и т.д.).

Успешные Internet-компании быстро растут, и им просто необходимо уметь легко, быстро и эффективно масштабировать свои системы управления данными, причем, как правило, речь идет именно о *горизонтальном* масштабировании, при котором при увеличении числа узлов в используемом кластере линейно возрастает пропускная способность системы. Другими словами, способность *горизонтального масштабирования (scaling out)* системы управления данными является необходимым условием развития бизнеса. Транзакции в таких приложениях обычно являются очень короткими и состоят из простых операций, так что обычно пользователей вполне устраивает время реакции неперегруженного запросами приложения.

Однако временами при взаимодействии пользователя с онлайновым приложением могут возникать задержки из-за недоступности каких-либо ресурсов на уровне управления данными. В условиях жесткой конкуренции на рынке онлайновых приложений такие задержки могут оказаться губительными для бизнеса. Поэтому не менее важным требованием к системе управления данными является *доступность (availability)*, гарантирующая отсутствие (или, как минимум, по возможности снижающая вероятность возникновения) таких задержек. Хуже всего на отношение клиентов к Internet-компании действуют сообщения типа *service unavailable*. Поэтому многие онлайновые компании готовы согласиться с тем, что в некоторых случаях результаты выполнения транзакций системой управления данными будут некорректными, если при этом доступность системы будет максимально возможной.

В [6] даже выстраивается некоторая "теоретическая" база, оправдывающая перед пользователями возможное некорректное поведение приложений тем, что "потребность в извинениях возникает в любом бизнесе". Другими словами, поскольку в любом бизнесе могут возникать ошибки, за которые придется извиняться перед клиентами, то почему бы не отнести к числу таких ошибок и подобное поведение приложений (рано или поздно наличие некоррректности будет установлено, и соответствующий пользователь получит моральную (или даже материальную) компенсацию).

Более серьезным "теоретическим" основанием NoSQL-разработок, в которых общепринятые полезные свойства систем управления данными приносятся в жертву доступности этих систем, является так называемая "теорема CAP", впервые сформулированная Эриком Брювером (Eric Brewer) [7]. (Здесь и далее в тесте я заключаю слово *теорема* в кавычки, поскольку утверждение, названное Брювером теоремой, таковым я признать не могу из-за отсутствия

какой-либо четкой и хотя бы немного формальной постановки задачи.) Как мне представляется, мало кто из людей из сообщества NoSQL (да и из традиционного сообщества баз данных) серьезно разбирался в сути этой "теоремы", но широко распространено мнение, что она означает невозможность поддержки в одной распределенной системе управления данных свойств согласованности данных (*Consistency*), доступности (*Availability*) и устойчивости к разделению сети (*Partitioning*). Обобщая *consistency* в смысле Брювера до полного набора свойств *ACID (Atomicy, Consistency, Isolation, Durability)* классических транзакций баз данных, сообщество NoSQL с готовностью отказывается от реальной поддержки транзакций в своих системах (поэтому, например, в [8] предлагается переименовать *NoSQL* в *NoACID*).

В этой статье я не буду касаться "экстремистских" решений для поддержки оналайновых "OLTP"-приложений (я заключил OLTP в кавычки, поскольку не понимаю, о каких "транзакциях" можно в этом случае говорить). Не то чтобы я недооцениваю эту ветвь разработок – с практической точки зрения они очень важны, но, во-первых, они просто не вписываются в контекст статьи, а во-вторых, я не вижу у разработок этой категории какой-либо общей идеологии, кроме отрицания SQL и ACID. Но имеется и другая ветвь, наиболее ярким представителями которой мне кажутся исследователи из Федерального швейцарского технологического института (ETH) Цюриха и среди них, прежде всего, Дональд Коссманн (Donald Kossmann).

Одна из основных (не технических) идей, на которых основываются исследования и разработки этой группы в последние годы, состоит в следующем (см., например, [9]). Облачная инфраструктура, в которой все чаще развертываются транзакционные приложения, основана на предоставлении пользователям разнообразных сервисов, в том числе, и сервисов управления данными. В этом случае компании-пользователи платят именно за услуги, и чем сложнее услуга, чем дороже обходится ее поддержка, тем больше за нее приходится платить. По мнению Коссманна, для поддержки ACID-транзакций требуются дополнительные расходы, оплачивать которые пользователи должны только в тех случаях, когда это качество службы управления данными им действительно требуется. Вокруг этих идей выполняются интересные исследования и разработки, заслуживающие, на мой взгляд, анализа.

Среди работ, в которых ни в коей мере не затрагиваются фундаментальные свойства ACID и при этом обеспечивается горизонтальное масштабирование параллельных систем баз данных, мне больше всего нравится проект H-Store [10], в котором участвуют исследователи Массачусетского технологического института, Йельского университета и университета Браун при участии таких "грандов" в области баз данных, как Майкл Стоунбрейкер и Стенли Здоник (Stanley Zdonik). На основе предварительных результатов этого проекта была основана компания VoltDB [11], выпустившая в середине 2010 г. свой первый

коммерческий программный продукт (кстати, с открытыми исходными текстами и лицензией GPL3).

Основной упор в этом проекте делается на достижение максимально возможной эффективности и обеспечение линейной горизонтальной масштабируемости при полной поддержке ACID-транзакций. Причем в последнее время во многих своих публикациях (см., например, [12-13]) Майкл Стоунбрейкер, на мой взгляд, вполне убедительно доказывает, что отказ от свойств ACID никоим образом не способствует повышению уровня доступности распределенных систем управления данными. Высокой же производительности и горизонтальной масштабируемости массивно-параллельных систем баз данных в наибольшей степени мешают распределенные транзакции и, в особенности, их двухфазная фиксация.

Устранению отрицательного влияния распределенных транзакций и посвящается бóльшая часть исследований проекта H-Store. Кроме того, внимания заслуживает работа [14], стоящая несколько особняком от основного направления проекта, но, несомненно, способствующая его успешному выполнению. В параллельных СУБД без совместного использования ресурсов база данных должна быть разделена на части, каждая из которых управляется локальным компонентом СУБД в отдельном узле кластера. В транзакционных системах важно научиться так разделять базу данных, чтобы в рабочих нагрузках появлялось как можно меньше распределенных транзакций. Авторы [14] предлагают интересный подход к обнаружению методов таких разделений.

Наконец, параллельным транзакционным СУБД свойственна еще одна проблема, к которой, на мой взгляд, недостаточно внимательно относятся участники проекта H-Store. Они отказываются от использования общих ресурсов даже внутри одного узла, в котором установлен компьютер с многоядерным процессором. Такой компьютер используется как набор виртуальных изолированных узлов, каждому из которых соответствует ядро процессора. В [15] обосновывается неэффективность такого подхода и предлагается оригинальная архитектура параллельной "одноузловой" СУБД, работающей на машине с многоядерным процессором. В этой архитектуре на физическом уровне все основные ресурсы процессора (основная и дисковая память) являются общими для всех потоков управления СУБД, а на логическом уровне данные разделяются между потоками управления. Демонстрируется, что такая организация СУБД позволяет резко снизить нагрузку на центральный менеджер блокировок и обеспечить хорошее масштабирование системы при возрастании числа ядер.

1.2. Структура статьи

Оставшаяся часть статьи организована следующим образом. В разд. 2 напоминается исходный смысл свойств ACID транзакций баз данных. Затем анализируется, как связано свойство "consustency" в смысле "теоремы" CAP с

соответствующим свойством ACID. Разд. 3 посвящен рассмотрению наиболее интересных архитектур и методов построения параллельных СУБД, в обязательном порядке поддерживающих ACID-транзакции. В разд. 4 обсуждаются подходы к обоснованному ослаблению свойства согласованности. Наконец, разд. 5 содержит заключение.

2. Классические свойства транзакций и "теорема" CAP

Во введении отмечалось, что многочисленные обсуждения следствий "теоремы" CAP на возможность поддержки ACID-транзакций в распределенных СУБД без совместно используемых ресурсов часто демонстрируют непонимание авторами сути свойств ACID и/или смысла "теоремы" CAP. В этом разделе, прежде всего, напоминается исходный смысл свойств ACID, который имелся в виду изобретателями этой аббревиатуры. Затем я постараюсь прояснить смысл "теоремы" Брювера и подвести читателей к мысли, что *consistensy* в этой теореме имеет мало общего с *consistensy*, входящей в число свойств ACID.

2.1. ACID: вернемся к истокам

Впервые аббревиатура ACID появилась в 1983 г. в статье Тео Хаердера (Theo Haerder) и Адреаса Рейтера (Andreas Reuter) [16]. Для упрощения текста и пущей убедительности я приведу перевод фрагмента этой статьи (с небольшими сокращениями). В этом фрагменте используется пример банковской транзакции из [17], в которой деньги переводятся с одного счета на другой (рис. 1).

Концепция транзакции, включающей в приведенном примере все взаимодействия с базой данных между `$BEGIN_TRANSACTION` и `$COMMIT_TRANSACTION`, требует, чтобы все действия выполнялись *нераздельно (indivisibly)*: либо все действия должным образом отражаются в состоянии базы данных, либо ничего не происходит. Если в какой-либо момент времени до достижения `$COMMIT_TRANSACTION` пользователь вводит оператор `ERROR`, содержащий `$RESTORE_TRANSACTION`, то в базе данных не отражаются никакие изменения. Для достижения такой неделимости транзакция должна обладать следующими четырьмя свойствами:

> *Атомарность (Atomicity).* Транзакция должна иметь описанный выше тип "все или ничего", и, что бы ни произошло, пользователь должен знать, в каком состоянии находится его транзакция.
>
> *Согласованность (*Consistency*).* Транзакция, достигающая своего нормального завершения (EOT – end of transaction, завершение транзакции) и, тем самым, фиксирующая свои результаты, сохраняет согласованность базы данных. Другими словами, каждая успешная транзакция по определению фиксирует только допустимые результаты.

Это условие является необходимым для поддержки четвертого свойства – долговечности.

Изоляция (Isolation). События, происходящие внутри транзакции, должны быть скрыты от других одновременно выполняемых транзакций. Если бы это условие не выполнялось, то по причинам, упомянутым выше, транзакцию было бы невозможно вернуть к своему началу. Для достижения изоляции используются методы, называемые *синхронизацией*...

Долговечность (Durability). После того, как транзакция завершилась и зафиксировала свои результаты в базе данных, система должна гарантировать, что эти результаты переживут любые последующие сбои. Поскольку отсутствует какая-либо область управления, охватывающая наборы транзакций, у системы управления базами данных (СУБД) нет никакого контроля вне пределов границ транзакций. Поэтому пользователю должно гарантироваться, что если система сообщает ему о том, что нечто произошло, то это "нечто" действительно произошло. Поскольку по определению любая (*успешно завершенная – С.К.*) транзакция является корректной, результаты неизбежно появляющихся некорректных транзакций (т.е. транзакций, содержащих ошибочные данные), могут быть устранены только соответствующей "контр"-транзакцией (countertransaction).

Эти четыре свойства – атомарность, согласованность, изоляция и долговечность (ACID) – описывают основные черты парадигмы транзакций, которые влияют на многие аспекты разработки систем баз данных. Поэтому мы считаем, что способность какой-либо системы к поддержке транзакций является пробным камнем (ACID test) качества этой системы.

```
FUNDS_TRANSFER. PROCEDURE,
 $BEGIN_TRANSACTION;
 ON ERROR DO;                          /* in case of error */
        $RESTORE_TRANSACTION,              /* undo all work */
        GET INPUT MESSAGE;               /* reacquire input */
        PUT MESSAGE ('TRANSFER FAILED');     /* report failure */
        GO TO COMMIT;
        END;
 GET INPUT MESSAGE;                      /* get and parse input */
 EXTRACT ACCOUNT_EBIT, ACCOUNT_CREDIT,
  AMOUNT FROM MESSAGE,
 $UPDATE ACCOUNTS                        /* do debit */
          SET BALANCE ffi BALANCE— AMOUNT
    WHERE ACCOUNTS.NUMBER = ACCOUNT_DEBIT;
 $UPDATE ACCOUNTS                        /* do credit */
          SET BALANCE = BALANCE + AMOUNT
```



```
    WHERE ACCOUNTS.NUMBER = ACCOUNT_CREDIT;
 $INSERT INTO HISTORY                    /* keep audit trail */
   <DATE, MESSAGE>;
 PUT MESSAGE ('TRANSFER DONE');          /* report success */
COMMIT:                                  /* commit updates */
 $COMMIT_TRANSACTION;
 END;                                    /* end of program */
```

Рис. 1. Простая программа на языке PL/1-SQL, переводящая средства с одного счета на другой.

Я привел эту длинную цитату, чтобы напомнить, что, по сути, свойства ACID, с одной стороны, можно рассматривать как требования к любой СУБД, претендующей на поддержку транзакций, а с другой стороны, – как *определение* транзакции в системе баз данных. Это определение полностью соответствует житейской практике. Трудно представить, например, чтобы клиент, выполняющий банковскую транзакцию (неважно, при содействии живого человека-операциониста, или с использованием Internet-банкинга), не рассчитывал на удовлетворение банком всех свойств ACID. Банк, не поддерживающий свойства ACID для своих транзакций, в лучшем случае потеряет клиентов, а в худшем – обанкротится.

Очень важно, что свойства ACID являются нераздельными, отбрасывание любого из них делает оставшуюся комбинацию бессмысленной. В частности, если отбросить свойство *согласованности* (в том смысле, в котором оно использовалось в приведенной цитате), то мы потеряем критерий корректности транзакции. Система баз данных не сможет каким-либо осмысленным образом принимать решение о допустимости или недопустимости фиксации транзакций, и все проверки корректности выполнения операций при текущем состоянии базы данных придется выполнять в коде приложений.

Нужно понимать, что в данном случае речь идет о *логической* согласованности. Клиенту банка нужно, чтобы банк работал по установленным им и известным клиентам правилам, чтобы нельзя было выполнять какую-либо транзакцию, нарушающую эти правила, чтобы следующая транзакция того же клиента выполнялась в среде, согласованной в соответствии с этими правилами. Клиенту онлайнового магазина нужно, чтобы заказанный и оплаченный им товар был своевременно ему доставлен (в соответствии с установленными и известными клиенту правилами). Иначе он не будет доверять этому магазину. При этом ни клиенту банка, ни клиенту Internet-магазина нет никакого дела до внутренней кухни предприятия, до того, какие внутренние действия предпринимаются для выполнения его транзакции. Клиенту нет дела до того, каким образом поддерживается *физическая* согласованность этого предприятия, каким образом выполняются операции на физическом уровне.

Если заботу о поддержке логической согласованности транзакций (и базы данных) берет на себя СУБД, то приложения становятся более простыми, понятными и надежными. Вся логика прикладной области (банка, магазина, склада и т.д.), касающаяся транзакций и допустимого состояния данных уходит в систему баз данных. И требования к этой системе очень просты: поддержка ACID-транзакций с учетом правил согласованности, обеспеченной в базе данных приложением. С моей точки зрения, отказ от ACID-транзакций создает немереные трудности для разработчиков приложений, которым, хочешь - не хочешь, придется самим реализовывать нечто похожее, чтобы удовлетворить естественные потребности своих клиентов.

И еще раз замечу, что свойства ACID, фактически, определяют понятие транзакции. На мой взгляд, чтобы иметь хотя бы какую-нибудь возможность говорить о транзакционной системе управления данными, в которой не поддерживается свойство согласованности транзакций, совершенно необходимо определить, что в этом случае понимается под термином *транзакция*. К сожалению, сегодня во многих случаях (в особенности, это свойственно направлению NoSQL) люди говорят о поддержке OLTP-приложений, совершенно не уточняя, что за транзакции имеются в виду. Поэтому в данной статье я буду использовать сочетание *ACID-транзакции* для обозначения настоящих транзакций, а неуточняемый термин *транзакция* будет использоваться в неформальном смысле, разном в разных контекстах.

Займемся теперь "теоремой" CAP и постараемся разобраться, что же означает *согласованность* в смысле Брювера.

2.2. Согласованность по Брюверу

Начнем с того, что Эрик Брювер не является и никогда не объявлял себя специалистом в области баз данных. Он относится к сообществу распределенных систем, и его знаменитый доклад [7], в котором появилась "теорема" CAP, был сделан на конференции "Принципы распределенных вычислений". (Кстати, десять лет спустя, в 2010 г. он еще раз выступил с приглашенным докладом [18] на той же конференции, и в этом докладе привел, в частности, ряд примеров распределенных систем, при разработке которых учитывалась "теорема" CAP.) В этой области имеется свое толкование терминов, используемых в области баз данных.

В частности, термин *мгновенная согласованность (immediate consistency)* означает, что после того как пользователь получает от системы извещение об успешном выполнении некоторой операции обновления данных, результат этой операции становится мгновенно видимым для всех наблюдателей. *Согласованность в конечном счете (eventual consistency)* означает, что если в течение достаточно долгого периода времени в систему не поступают новые операции обновления данных, то можно ожидать, что результаты всех предыдущих операций обновления данных, в конце концов, распространятся по всем узлам системы, и все реплики данных *согласуются* (*по всей*

видимости, это нужно понимать как "у всех реплик будет одно и то же состояние" – С.К.). Скорее всего, в [7] под *согласованностью* понимается именно *мгновенная согласованность* данных.

Имея в виду этот смысл понятия согласованность, можно считать "теорему" Брювера вполне понятной и очевидной: в любой распределенной системе с разделенными данными можно одновременно обеспечить только любые два свойства из *согласованности*, *доступности* и *устойчивости к разделению сети*. В связи с этим Брювер даже противопоставляет набор свойств ACID предлагаемому им набору свойств BASE (Basically Available, Soft-state, Eventual consistency – доступность в большинстве случаев; неустойчивое состояние; согласованность в конечном счете). Но это противопоставление, по моему мнению, неправомерно, поскольку в первом случае речь идет о логических характеристиках транзакций, а во втором – о физических свойствах распределенных систем.

Многие считают, что "теорема" Брювера формально доказана. Действительно, в статье Сета Гильберта (Seth Gilbert) и Нэнси Линч (Nancy Lynch) [19] вводятся некоторые (почти) формальные определения, в контексте которых "теорема" действительно становится теоремой и доказывается. Однако давайте разберемся, как же определяются те три свойства распределенной системы, из числа которых по "теореме" Брювера можно одновременно обеспечить поддержку только двух свойств.

Согласованностью в [19] называется *атомарная, или линеаризуемая согласованность (atomic, or linearizable consistency)*, являющаяся свойством системы, все индивидуальные объекты данных которой являются атомарными (линеаризуемыми). В свою очередь, *атомарным объектом* называется объект с несколькими операциями, такими что вызов операции и получение ответных данных происходят как бы мгновенно, т.е. объект не принимает вызов следующей операции до полного завершения предыдущей операции. При этом порядок приема операций должен быть таким, что если операция типа чтения поступает после выполнения некоторой операции типа записи, то операция чтения должна вернуть значение, записанное этой или какой-либо более поздней операцией записи.

Распределенная система является *постоянно доступной*, если на каждый запрос, полученный не отказавшим узлом, должен быть получен ответ. *Устойчивость* системы *к разделению сети* в [19] моделируется как сохранение жизнеспособности системы при потере произвольного числа сообщений, посылаемых из одного узла в другой.

На основе этих определений Гильберт и Линч формулируют следующую теорему (в асинхронной модели сети отсутствуют часы, и в узлах должны приниматься решения только на основе получаемых сообщений и локальных вычислений):

В асинхронной модели сети невозможно реализовать объект данных с операциями чтения и записи, гарантирующий обеспечение свойств доступности и атомарной согласованности для всех допустимых выполнений (включая те, в которых теряются сообщения).

Эта теорема действительно достаточно просто формально доказывается методом "от противного". Далее в [19] выводится следствие, заключающееся в том, что:

> В асинхронной модели сети невозможно реализовать объект данных с операциями чтения и записи, гарантирующий обеспечение свойств доступности для всех допустимых выполнений и атомарной согласованности для допустимых выполнений, в которых сообщения не теряются.

Кроме того, доказывается истинность основной теоремы для частично синхронной модели сети, в которой в каждом узле присутствуют часы, время, показываемое которыми, увеличивается с одной и той же скоростью, но которые не синхронизованы, т.е. могут показывать разное время в один и тот же реальный момент. Показано, что для этого случая аналогичное следствие не выводится, и, значит, для частично синхронных сетей имеется больше возможностей организации распределенных систем с "хорошими" свойствами.

Да, можно считать, что в некотором смысле (не обязательно совпадающем со смыслом, который имелся в виду Брювером) Гильберт и Линч доказали невозможность одновременного обеспечения в одной распределенной системе свойств атомарной согласованности, доступности и устойчивости к разделению сети. Но какое отношение это имеет к транзакциям баз данных вообще и к ACID-транзакциям в частности?

Вот что пишет по этому поводу в своей заметке [20], посвященной обсуждению "теоремы" CAP и статьи [19], Джулиан Браун (Julian Browne):

> В своем доказательстве Гильберт и Линч используют вместо термина *согласованность* термин *атомарность*, что с технической точки зрения более осмысленно, потому что, строго говоря, *согласованность* в смысле ACID относится к идеальным свойствам транзакций баз данных и означает, что никакие данные не станут долговременно хранимыми, если они нарушают некоторые заранее установленные ограничения. Но если полагать, что заранее установленным ограничением распределенных систем является запрет наличия нескольких разных значений у одного и того же элемента данных, то, по моему мнению, этот изъян в абстракции *согласованности* можно считать несущественным (кроме того, если бы Брювер использовал термин *атомарность*, то появилась бы теорема AAP, название которой было бы чрезвычайно неудобно произносить).

Это написано не очень серьезно, но честно. И, на самом деле, требование атомарной согласованности нельзя перемешивать с требованиями согласованности транзакций в смысле ACID. Ограничения целостности базы данных – это логические, если угодно, бизнес-требования. Они происходят из логики прикладной области. Требование атомарной согласованности совсем другого рода. Это реализационное требование, относящееся к той категории, которую традиционно в области баз данных называли *физической* согласованностью (например, при выполнении любой операции изменения индекса все блоки соответствующего B+-дерева должны содержать корректные значения и быть связаны корректными ссылками).

А вот что уже совсем серьезно пишут в своей заметке [8] представители сообщества баз данных Дэниэль Абади (Daniel Abadi) и Александер Томсон (Alexander Thomson):

> ... все более критичным становится требование к доступности масштабируемых транзакционных систем, и обычно оно удовлетворяется за счет репликации и автоматического перенаправления запросов в случае сбоя одного из узлов. Поэтому разработчики приложений ожидают, что гарантии согласованности (consistency) ACID-систем (первоначально заключавшиеся в локальной поддержке определенных пользователями инвариантов) будут распространены на обеспечение строгой согласованности (того, что все реплики одних и тех же данных в любой момент времени будут являться идентичными копиями, т.е. в этом случае согласованность подразумевается в смысле CAP/PACELC (*про PACELC см. в [21] – С.К.*)).

Другими словами, согласованность по Брюверу не имеет ничего общего с согласованностью в смысле ACID, но именно в системах, ориентированных на обеспечение высокого уровня доступности за счет репликации данных, желательно поддерживать строгую согласованность реплик. Это не свойство ACID, а техническая (физическая) особенность массивно-параллельных СУБД, облегчающая разработку приложений.

Как считает Майкл Стоунбрейкер [12-13], залогом построения качественной современной СУБД является правильный выбор технических компромиссов. При выборе конкретного инженерного решения нужно учитывать множество факторов – требования будущих пользователей, вероятности возникновения различных сбойных ситуаций и т.д., а не руководствоваться догматическим образом каким-либо общими теоретическими указаниями (в том числе, и "теоремой" CAP).

Стоунбрейкер полагает, что в области транзакционных параллельных систем баз данных отказ от согласованности по Брюверу в пользу поддержки высокой доступности и устойчивости к разделению сети является плохим компромиссом, поскольку (a) согласованность реплик является очень полезным свойством системы; (b) транзакционные массивно-параллельные

СУБД не нуждаются в кластерах с очень большим числом узлов, так что ситуации разделения сети маловероятны; (c) система может легко стать недоступной не из-за разделения сети, а, например, из-за наличия регулярно проявляющихся программных ошибок.

Таким образом, высокая активность представителей лагеря NoSQL (читай NoACID), которые часто ссылаются на "теорему" Брювера, связана не с теоретической невозможностью построения массивно-параллельных транзакционных СУБД, поддерживающих ACID-транзакции, а с тем, что упрощенные системы, не поддерживающие не только ACID-транзакции, но и согласованность реплик, создаются проще и быстрее. Из-за своей упрощенной организации они способны обеспечивать очень быструю обработку данных, и для ряда приложений это оказывается более важным, чем все удобства, свойственные технологии баз данных.

Посмотрим, как отвечает на этот вызов сообщество баз данных.

3. Новые транзакционные архитектуры, поддерживающие классические свойства транзакций

Понятно, что конкурировать на рынке OLTP с системами основных поставщиков SQL-ориентированных СУБД (IBM, Oracle и Microsoft), которые в течение многих лет оптимизировались именно для обработки ACID-транзакций, может только СУБД, обладающая некоторыми принципиально отличными качествами. В настоящее время такими потенциально достижимыми новыми качествами являются существенно (в десятки раз) бо́льшая пропускная способность транзакций и горизонтальная масштабируемость (т.е. возможность линейного повышения пропускной способности при наращивании аппаратных ресурсов). Все более распространенным (и подтверждаемым практикой) мнением является то, что такие качества можно обеспечить только при использовании архитектур без совместного использования ресурсов (shared-nothing). Одним из наиболее ярких свидетельств правильности этого мнения являются проект H-Store [10] и достижения компании VoltDB [11], которым посвящается подраздел 3.1.

С другой стороны, нельзя не учитывать, что продолжает действовать закон Мура, в соответствии с новой трактовкой которого экспоненциально возрастает число ядер в микропроцессорах. В действительности в узлах массивно-параллельной СУБД (даже основанной на использовании компьютеров категории массового спроса) применяются мощные компьютеры, позволяющие эффективно использовать локальные СУБД с совместным использованием ресурсов. Другими словами, для построения предельно эффективной транзакционной массивно-параллельной СУБД нельзя не позаботиться об эффективности локальных СУБД, работающих на многоядерных процессорах. В этом направлении исследований мне хочется выделить проект DORA [15], в котором оригинальным образом сочетаются

подходы *shared-everything* на физическом уровне архитектуры СУБД и *shared-nothing* на логическом уровне. Этот проект обсуждается в подразделе 3.2.

3.1. H-Store: ничего лишнего

Впервые краткое описание исходных идей проекта H-Store появилось в 2007 г. в [22]. Эта статья была последней в цикле "один размер непригоден для всех" (см. также [1, 23]), в котором доказывалось, что прошло время универсальных, пригодных для поддержки любых приложений баз данных SQL-ориентированных СУБД, и обосновывались преимущества специализированных архитектур. В [22] речь идет исключительно о специализированных транзакционных системах, основанных на следующих пяти основных соображениях.

i. В основной памяти недорогой массивно-параллельной системы уже сейчас можно разместить базу данных объемом до одного терабайта. Этого достаточно для большинства приложений OLTP. Поэтому будущее за системами транзакционных баз данных, полностью размещаемых в основной памяти.

ii. В системах OLTP транзакции являются очень легковесными. При работе с базой данных в основной памяти время выполнения наиболее тяжелой транзакции из тестового набора TPC-C составляет менее одной миллисекунды. В большинстве приложений OLTP при выполнении транзакций отсутствуют задержки по вине пользователей. Поэтому имеет смысл выполнять все операции каждой транзакции последовательно в одном потоке управления (*если транзакция не затрагивает данные нескольких узлов – С.К.*).

iii. Кажется правдоподобным, что в следующем десятилетии будут доминировать компьютерные системы без общих ресурсов, и все СУБД следует оптимизировать в расчете на использование такой архитектуры. Если система с N узлами не обеспечивает достаточной мощности, должна иметься возможность добавления к ней дополнительных K узлов без потребности в каких-либо сложных действиях над используемой СУБД (*то самое горизонтальное масштабирование – С.К.*).

iv. В будущем высокий уровень доступности и встроенные средства восстановления после отказов станут важными чертами рынка OLTP. Из этого следует несколько выводов.
   a. В любой СУБД, ориентированной на поддержку OLTP, потребуется согласованная репликация данных.
   b. Наиболее эффективной является поддержка архитектуры *shared-nothing* на всех уровнях системы (*как я уже говорил, это неочевидно, см. следующий подраздел – С.К.*)
   c. Наилучший способ поддержки архитектуры без общих ресурсов состоит в использовании нескольких машин в одноранговой (peer-to-peer) конфигурации. Тогда нагрузка

OLTP может быть распределена между несколькими машинами, а межмашинную репликацию можно использовать для обеспечения отказоустойчивости.

d. В мире высокой доступности не требуется поддержка журнала повторного выполнения операций, а нужен только временный, сохраняемый в основной памяти журнал откатов.

e. Основные расходы IT-подразделений уходят на содержание персонала. Единственным выходом из этого положения является перевод систем на «самообслуживание» (самовосстановление, автоматическое техническое обслуживание, автоматическую настройку и т.д.). Требуется полный пересмотр процесса настройки системы без явных ручек управления.

Эти соображения приводят к следующим выводам.

i. Основным препятствием для достижения высокой производительности системы почти наверняка станет журнал повторного выполнения операций, сохраняемый в дисковой памяти. Без него можно обойтись за счет подсистемы поддержки высокого уровня доступности и обработки отказов.

ii. Следующим по значимости узким местом системы является вызов в ней операций и возврат результатов в приложение. Наиболее эффективным способом решения этой проблемы является выполнение логики приложений в виде хранимых процедур внутри системы баз данных.

iii. Во всех возможных случаях следует отказаться от поддержки и журнала откатов транзакций, поскольку он тоже будет сдерживать производительность.

iv. Следует приложить все усилия, чтобы максимально освободиться от затрат на синхронизационные блокировки.

v. Желательно освободиться и от синхронизации на основе "защелок" при доступе к одним и тем же структурам данных из нескольких потоков управления. С учетом кратковременности транзакций эти накладные расходы можно устранить путем перехода к однопотоковой модели выполнения транзакций.

vi. По мере возможности следует избегать применения двухфазного протокола фиксации распределенных транзакций.

3.1.1. Свойства схем транзакционных баз данных и типичных транзакций

В H-Store требуется наличие заранее специфицированного набора классов транзакций, которые могут входить в рабочую нагрузку системы. Каждый класс характеризует транзакции с одними и теми же операторами SQL и логикой приложения, различающиеся только значениями констант времени выполнения. Это требование не является неестественным, поскольку для

транзакционных приложений нехарактерно наличие непредвиденных запросов, явно вводимых пользователями во время выполнения транзакции. Таким образом, задержки выполнения транзакций по вине пользователей невозможны.

Аналогичным образом, считается, что заранее известна логическая схема базы данных, над которой будут выполняться эти транзакции. Авторы [22] обнаружили, что многие транзакционные базы данных обладают древовидной логической схемой, в которой каждая таблица (кроме одной – корневой) имеет связь *n*:1 ровно с одной таблицей-предком (т.е. она естественным образом соединяется только с одной таблицей). Для баз данных с древовидной схемой имеется очевидный метод горизонтального разделения данных: корневая таблица разделяется по диапазонам значений первичного ключа (или на основе хэширования этих значений); каждая таблица-потомок разделяется таким образом, чтобы при естественном соединении с каждым разделом таблицы-предка потребовались бы строки только одного раздела таблицы-потомка, причем этот раздел размещается в том же узле, что и соответствующий раздел таблицы-предка.

Если в каждом операторе SQL каждой транзакции содержится условие, выделяющее ровно одну строку корневой таблицы (например, любая операция относится к некоторому клиенту онлайнового магазина, и таблица клиентов является корневой), то при таком разделении каждый оператор будет выполняться ровно в одном узле (будет являться *локальным* для этого узла). Если все операторы каждой транзакции локальны для одного и того же узла, то соответствующее приложение называется *приложением над ограниченным деревом (constrained tree application, CTA)*. Ценное свойство CTA-приложений состоит в том, что все его транзакции могут быть полностью выполнены в одном узле, т.е. являются *одноузловыми*. Такие транзакции выполняются без каких-либо задержек из-за коммуникации с другими узлами (кроме возможных задержек из-за синхронизации обновления реплик).

По опыту авторов, многие приложения OLTP сразу разрабатываются в стиле CTA, а во многих других случаях их можно преобразовать в CTA-приложения. (Заметим, что здесь и далее очень заметно влияние на ранней стадии проекта H-Store Пэта Хелланда (Pat Helland), который входит в число авторов [22], а ранее, будучи сотрудником Amazon, написал статью [24], где высказывал схожие соображения. Интересно также отметить, что в дальнейших работах, посвященных H-Store, влияние идей Хелланда почти незаметно.)

Для преобразования к виду CTA приложений, изначально таковыми не являющихся, в [22] предлагалось использовать два подхода. Во-первых, можно выделить все таблицы, которые во всех транзакциях только читаются. Такие таблицы можно реплицировать во всех узлах. Если некоторое приложение обладает свойством CTA по отношению ко всем остальным таблицам, то после такой репликации оно станет CTA-приложением. Во-

вторых, имеется еще один важный класс OLTP-приложений, части транзакций которых можно выполнять параллельно без потребности в передаче между узлами промежуточных результатов, причем результаты операций SQL никогда не требуются при выполнении последующих операций. Транзакции таких приложений *с одноразовым использованием результатов (one-shot)* можно преобразовать в набор одноузловых планов, каждый из которых выполняется только в одном узле. Часто такие преобразования можно произвести за счет *вертикального* разделения таблиц между узлами (только читаемые вертикальные разделы реплицируются).

Некоторые классы транзакций обладают свойством *двухфазности (two-phase)* или могут быть преобразованы к двухфазным транзакциям. На первой фазе такой транзакции выполняются только операции чтения, и только на этой фазе допускается аварийное завершение транзакции по ее собственной инициативе (т.е. в это время производятся все возможные проверки, вследствие которых может потребоваться аварийное завершение транзакции). Важным свойством двухфазных транзакций является то, что при их выполнении можно обойтись без журналов откатов транзакций. Класс двухфазных транзакций называется *строго двухфазным (strongly two-phase)*, если на первой фазе выполнения транзакции во всех узлах, участвующих в ее выполнении, принимается одно и то же решение относительно ее продолжения или аварийного завершения.

Две параллельно выполняемые транзакции (из одного или разных классов) называются *коммутативными*, если при любом чередовании их одноузловых планов производится одно и то же окончательное состояние базы данных (если обе транзакции фиксируются). Класс транзакций, коммутативных со всеми транзакциями, называется *стерильным (sterile)*.

3.1.2. Как был устроен и как работал начальный вариант H-Store

H-Store выполняется в кластере компьютеров (*почему-то в [22] эта аппаратная среда упорно называется grid'ом – С.К.*). При конфигурировании системы можно указать желаемый уровень ее надежности – число узлов, при выходе из строя которых система может восстановить работоспособность без потери выполняемых транзакций в течение заданного времени. (*Поскольку восстановление системы основано на использовании реплик, то, очевидно, уровень надежности коррелирует с числом поддерживаемых реплик данных – С.К.*).

В каждом узле строки разделов таблиц размещаются вплотную одна к другой, и доступ к ним производится на основе B-деревьев (т.е. строки размещаются в порядке сортировки по значениям ключа B-дерева). Размер блока B-дерева соответствует размеру блока кэша второго уровня используемого процессора. (*Сравнительно ясно, что является ключом B-дерева для баз данных с древовидной схемой – первичный ключ для корневой таблицы и внешний ключ*

для любой таблицы-потомка. Что выбирается в качестве ключа B-дерева раздела таблицы при наличии других схем, неясно – С.К.).

В каждом узле H-Store поддерживает ровно один поток управления, в котором полностью, без каких-либо задержек выполняется каждая поступающая операция SQL. Узлы, в процессорах которых имеется несколько ядер, разбиваются на соответствующее число логических узлов. В H-Store каждый логический узел трактуется так же, как и любой физически независимый узел, и основная память многоядерного компьютера разделяется между логическими узлами.

Транзакции представляются в виде хранимых процедур базы данных, и в системе поддерживается только одна внешняя операция `Execute transaction (parameter_list)`, позволяющая в любом узле инициировать выполнение любой предопределенной транзакции с передачей ей значений параметров. Внутри таких хранимых процедур (для написания которых в исходном прототипе использовался язык C++) сочетается логика приложений и операции манипулирования базами данных, причем вызовы SQL производятся как локальные вызовы. Журнал повторного выполнения операций не поддерживается, а журнал отката (сохраняемый в основной памяти и освобождаемый при завершении транзакции) ведется только для транзакций, не являющихся двухфазными.

В исходном прототипе H-Store отсутствовал компилятор SQL, и планы всех операций SQL генерировались и оптимизировались вручную. Однако в [22] отмечалось, что планируется разработка компилятора SQL с оценочной (cost-based) оптимизацией, и что этот компилятор-оптимизатор должен быть сравнительно простым, поскольку в типичном OLTP-запросе всегда идентифицируется некоторый *опорный кортеж (anchor tuple)*, с которым соединяются несколько (немного) таблиц. И некоторый компилятор SQL действительно появился в коммерческом варианте H-Store – VoltDB (см., например, [25]), хотя по доступной документации системы трудно судить, какие возможности оптимизации в нем реализованы.

Для обеспечения возможности использования H-Store без потребности в "ручках управления" планировалось создание средства автоматического проектирования физических схем баз данных (*дизайнера баз данных*), определяющего горизонтальное разделение, репликацию и выбор ключей индексов. Цель дизайнера состоит в том, чтобы сделать как можно больше транзакций *одноузловыми* (т.е. избежать появления распределенных транзакций). (*И, кроме того, насколько я понимаю, добиться выявления двухфазных и стерильных транзакций – С.К.*).

Мне с самого начала возможность создания такого средства казалась сомнительной. Уж очень трудна задача статического анализа многочисленных хранимых процедур с многочисленными вызовами операций SQL. К настоящему времени (конец 2010 г.) эта задача, по всей видимости, не решена. В документации VoltDB [25] разработчикам приложений предлагается лишь

методика физического проектирования баз данных, да и то подчеркивается необходимость многократного выполнения тестовых испытаний (benchmarking, benchmarking, benchmarking!) на реальных данных до вывода приложения в производственный режим. С другой стороны, некоторую надежду на продвижение в этом направлении дает работа [14], хотя она основывается уже не на статическом анализе, а на анализе трасс выполнения рабочей нагрузки (см. ниже).

Выполнение транзакций в исходном прототипе H-Store происходило по следующей схеме. На входе в систему каждой транзакции назначалась *временная метка (timestamp)* в формате (`site_id, local_unique_time-stamp`). Если поддерживается порядок на множестве узлов кластера, то все метки являются уникальными и полностью упорядоченными. Предполагалось, что локальные часы в каждом узле некоторым образом синхронизируются.

Если все классы транзакций являются *одноузловыми*, то каждая транзакция может быть направлена в узел с требуемыми копиями данных и полностью в нем выполнена. Если не все классы транзакций являются *стерильными*, то узел, в котором завершилось выполнение некоторой транзакции, в течение небольшого времени (времени сетевых задержек – для локальной сети в пределах нескольких миллисекунд) ожидает поступления транзакций от других узлов-инициаторов, чтобы выполнение транзакций происходило в порядке временны*х* меток. За счет этого все реплики будут обновляться в одном и том же порядке, и их состояние после конца любой транзакции будет идентично. Следовательно, для выполнения одноузловых транзакций не требуется журнал повторного выполнения операций, какое-либо управление параллелизмом и распределенная фиксация транзакций. Если же, в придачу ко всему остальному, транзакция является двухфазной, то для нее не требуется и журнал отката.

Если все транзакции являются *стерильными*, то обычно для их выполнения не требуется какое-либо управление параллелизмом. Более того, в этом случае не требуется назначение транзакциям временны*х* меток и их выполнение в одном и том же порядке над всеми репликами. Но если транзакция распространяется на несколько узлов, то отсутствует гарантия, что она будет во всех узлах успешно выполнена или аварийно завершена. Поэтому каждый исполнитель должен послать диспетчеру выполнения транзакции (в том узле, в котором она была инициирована) сообщение "аварийное завершение" или "нормальное завершение" в той точке своей части транзакции, после которой соответствующее решение изменить уже нельзя (для *двухфазных* транзакций – в конце первой фазы). Диспетчер, в свою очередь, должен рассылать эти сообщения в другие узлы-исполнители. Другими словами, в этом случае приходится выполнять обычную процедуру фиксации распределенной транзакции. Если транзакция является *строго двухфазной*, этих накладных расходов можно избежать.

В общем случае (когда транзакция не является *одноузловой* или *стерильной*), приходится применять средства управления параллелизмом. При реализации исходного прототипа H-Store было принято решение отказаться от традиционной для SQL-ориентированных СУБД пессимистической схемы синхронизационных блокировок в пользу более оптимистических методов (*как показывают более поздние статьи, посвященные H-Store в целом [26] и управлению транзакциями в H-Store [27-28], это решение не является стратегическим – поиск методов продолжается; кстати, совершенно неясно, какая схема управления параллелизмом применяется в VoltDB – С.К.*).

В схеме управления параллелизмом, описываемой в [22], применяются три стратегии – *основная*, *промежуточная* и *усложненная*. Для каждого класса транзакций определяются классы транзакций, с которыми транзакции данного класса могут конфликтовать (*по-видимому, выявляются стандартные конфликты "запись-запись", "чтение-запись" и "запись-чтение" на уровне таблиц – С.К.*). Каждая транзакция инициируется в некотором узле системы, и ее выполнение отслеживается *координатором транзакций* в этом узле. Координатор играет роль *диспетчера транзакций* в узле инициации и рассылает фрагменты транзакции в соответствующие узлы.

При применении *основной* стратегии сайт-*исполнитель* получает фрагмент и выжидает в течение времени сетевых задержек до момента, когда возможно поступление транзакции с *большим* значением временно*й* метки. Если после этого в узле обнаруживаются транзакции с более поздними временны*ми* метками из классов, потенциально конфликтующих с данной транзакцией, то исполнитель не выполняет ее фрагмент и посылает координатору сообщение "аварийное завершение". В противном случае фрагмент выполняется, и результирующие данные возвращаются в узел, из которого этот фрагмент был получен. Если координатор получает сообщение о нормальном выполнении фрагментов от всех узлов, он продолжает выполнение транзакции при наличии следующего набора ее фрагментов или же фиксирует транзакцию. В противном случае транзакция завершается аварийным образом.

Если при использовании *основной* стратегии возникает слишком много аварийных завершений транзакций (*т.е. в данной рабочей нагрузке имеется высокий уровень потенциальной конфликтности транзакций – С.К.*), применяется более сложная *промежуточная* стратегия. В этом случае каждый исполнитель до принятия решения о выполнении или аварийном завершении полученного фрагмента выжидает интервал времени величиной `MaxD * среднее_время_обмена_сетевыми_сообщениями` (где `MaxD` – максимальное число межузловых сообщений, которое потребуется для выполнения любой потенциально конфликтующей транзакции), чтобы установить, не появится ли фрагмент транзакции с *меньшим* значением временной метки. В этом случае исполнитель получает возможность корректного упорядочивания фрагментов, что уменьшает вероятность аварийного завершения транзакций.

Наконец, *усложненная* стратегия, к которой планировалось прибегать в тех случаях, когда ни *основная*, ни *промежуточная* стратегии не позволяют в достаточной степени сократить число аварийных завершений транзакций, – это достаточно традиционная стратегия оптимистического управления параллелизмом. В этом случае конфликты распознаются во время выполнения, для чего в каждом узле отслеживаются *наборы прочитанных данных (read set)* и *наборы измененных данных (write set)* каждой транзакции. Любой исполнитель запускает выполнение любого фрагмента и аварийно его завершает, если это требуется для разрешения динамически обнаруженной конфликтной ситуации.

Авторы [22] сравнивали производительность начального прототипа H-Store с производительностью неназываемой коммерческой SQL-ориентированной СУБД на эталонном тестовом наборе TPC-C. В обоих случаях транзакции реализовывались в виде хранимых процедур и запускались внешней командой, передаваемой по сети. За счет специально подобранного метода разделения базы данных TPC-C и тщательного анализа транзакций удалось преобразовать все классы транзакций к стерильному и строго двухфазному виду. В результате на одной и той же аппаратной конфигурации (один компьютер с двухъядерным процессором) H-Store показала производительность, в 82 раза превышающую производительность традиционной СУБД. По наблюдениям авторов, главным тормозом традиционной СУБД стала система журнализации изменений в дисковой памяти. На втором месте – накладные расходы на управление параллелизмом.

Кстати, в 2008 г. основные авторы [22] опубликовали результаты более глубокого исследования влияния на производительность традиционных механизмов транзакционных СУБД [29]. Это исследование, фактически, объясняет, за счет чего в H-Store удалось добиться такой производительности. Авторы взяли не очень известную систему Shore [30] с открытыми исходными текстами, сконфигурировали ее таким образом, чтобы требуемая для их экспериментов база данных полностью помещалась в основной памяти, и измерили производительность полученной системы базы данных на смеси двух транзакций из тестового набора TPC-C. Затем они последовательно стали удалять из состава Shore компоненты журнализации, синхронизации и управления буферным пулом, и в результате получили вариант системы с ограниченной функциональностью, которая показала на том же тестовом наборе производительность, в 20 раз большую, чем у исходной Shore.

3.1.3. Спекулятивное выполнение транзакций

Как отмечалось выше, оптимистическая схема управления распределенными (не одноразделными) транзакциями, разработанная в начале проекта H-Store, не стала стратегической схемой этого проекта. Продолжались (и, по-видимому, продолжаются) исследования, направленные на поиск более эффективных схем. Результаты одного из перспективных исследований

описываются в [27]. Идея заключается в том, чтобы постараться избежать простаивания потоков управления, когда выполняемые в них фрагменты распределенных транзакций вынуждаются ожидать поступления сетевых сообщений.

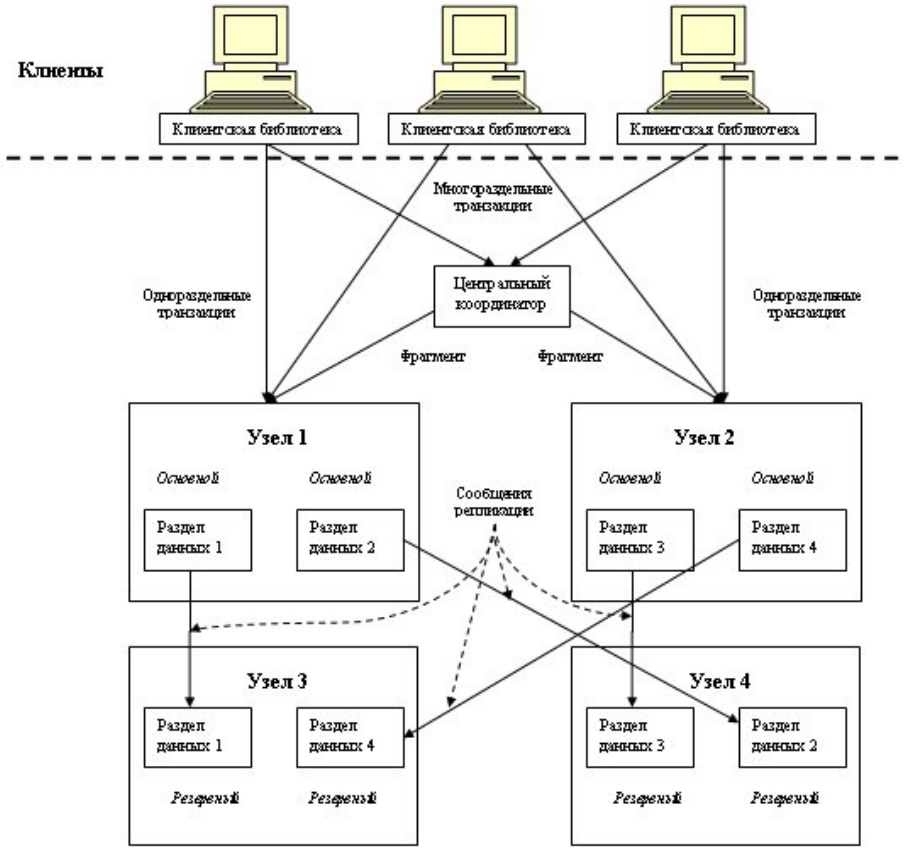

Рис. 2. Архитектура системы

В этом исследовании использовался прототип H-Store, архитектура которого показана на рис. 2. Для каждого раздела данных имеются один основной поток управления и k-1 резервных потоков управления (где k – это упоминавшийся в предыдущем пункте уровень надежности системы; у системы на рис. 2 k=2). В отличие от [22], в архитектуре на рис. 2 используется только один координатор распределенных транзакций (авторы объясняют это тем, что исследования способов полного упорядочивания транзакций при наличии нескольких координаторов, т.е. способов обеспечения уникальных и

полностью упорядоченных временны*х* меток транзакций, пока еще не завершены). Наличие единственного центрального координатора ограничивает число одновременно выполняемых распределенных транзакций, но этот вариант конфигурации системы является временным.

Каждая транзакция разбивается на *фрагменты* – части работы, каждую из которых можно выполнить над данными только одного раздела. Транзакции делятся на *одноразделные* и *многоразделные* (*в [22] они назывались* одноузловыми *и* многоузловыми *соответственно; с моей точки зрения, новые термины являются более правильными, поскольку более точно характеризуют природу транзакций; тем не менее, в предыдущем пункте я решил следовать терминологии [22] – К.С.*).

Одноразделной транзакцией называется такая транзакция, для полного выполнения которой достаточно данных некоторого одного раздела (т.е. одноразделная транзакция состоит из одного фрагмента). Как показывает рис. 2, одноразделные транзакции запускаются сразу в потоках управления соответствующих разделов, минуя центральный координатор. (В прототипе H-Store приписывание фрагментов транзакций к разделам делается вручную, хотя в перспективе это должен делать автоматический планировщик). Для обеспечения долговечности результатов транзакций используется протокол репликации. Поток управления основного раздела, получив заявку на выполнение транзакции, пересылает ее всем потокам управления резервных разделов. После этого в основном разделе выполняется транзакция, в конце которой поток управления убеждается в получении подтверждения от потоков резервных разделов. Фактически, если получено подтверждение хотя бы от одной реплики, транзакцию можно считать зафиксированной.

В ходе выполнения одноразделной транзакции в соответствующем потоке управления отсутствует какой-либо параллелизм, поэтому не требуется синхронизация. Если в транзакции невозможно аварийное завершение по инициативе пользователя, то в большинстве случаев не требуется поддерживать журнал отката транзакции. (*В используемом прототипе H-Store, а также и в текущей версии VoltDB, пользователи могут сопровождать транзакции аннотациями, сообщающими системе, что в транзакции нет инициируемых пользователями откатов, – своеобразное приближение к двухфазным транзакциям, обсуждавшимся выше. – С.К.*). При наличии потребности журнал отката поддерживается в основной памяти и освобождается при завершении транзакции. (*Кстати, похоже, что если в транзакции возникает откат по инициативе пользователя, то он должен возникнуть и во всех репликах, поскольку во всех них выполняется один и тот же код над одними и теми же данными. – С.К.*).

Многораздельные транзакции поступают в систему через центральный координатор, устанавливающий их глобальный порядок. Координатор разбивает транзакцию на фрагменты и посылает их потокам управления соответствующих разделов. После получения ответов от этих потоков

управления в координаторе выполняется код приложения, определяющий дальнейшее выполнение транзакции, для которого может потребоваться рассылка по потокам управления разделов дополнительных фрагментов. В каждом разделе фрагменты выполняются последовательно.

При выполнении всех фрагментов многораздельных транзакций поддерживаются журналы откатов. Они требуются для поддержки двухфазного протокола фиксации. Координатор посылает сообщение "подготовиться к фиксации" в каждый поток управления основного раздела, участвующий в выполнении данной распределенной транзакции, вместе с последним предназначенным этому потоку фрагментом транзакции. Получив это сообщение, поток основного раздела рассылает все полученные им ранее фрагменты этой транзакции в потоки резервных разделов, выполняет последний фрагмент транзакции, дожидается подтверждений от потоков резервных разделов и отправляет свои окончательные результаты координатору.

Если координатор получил все подтверждения от всех участников транзакции, он завершает транзакцию, посылая сообщение "фиксация" всем участникам и возвращая результаты транзакции приложению. Если же хотя бы от одного участника распределенной транзакции подтверждение о готовности к фиксации не получено, координатор посылает всем остальным участникам сообщение "аварийное завершение", принуждая их откатить все свои фрагменты транзакции. В результате, при выходе из строя некоторых узлов кластера или потере связности сети можно продолжать выполнять транзакции над разделами, оставшимися доступными клиентам (для одноразделных транзакций) и/или координатору (для многораздельных транзакций).

Тем самым, при выполнении многораздельных транзакций в потоках управления основных разделов могут возникать задержки из-за ожидания получения по сети сообщений от других потоков управления. Причем при использовании гигабитного Ethernet за время обмена сообщениями между двумя машинами по сети (40 миллисекунд) в потоке управления можно выполнить почти две одноузловые транзакции из тестового набора TPC-C. В [27] предлагаются и сравниваются методы, позволяющие системе выполнять полезную работу в то время, в которое она в противном случае простаивала бы.

Самый простой способ управления многоузловыми транзакциями заключается в том, что после получения первого фрагмента многораздельной транзакции поток управления основного раздела не обрабатывает фрагменты других многораздельных транзакций до тех пор, пока не обработает последний фрагмент данной транзакции. Фрагменты всех остальных транзакций ставятся в очередь, т.е. блокируются. Можно считать, что система расценивает все транзакции, затрагивающие один и тот же раздел, как *конфликтующие*, и выполняет их по очереди. Естественно, метод *блокирования транзакций* не позволяет системе выполнять полезную работу во время простоев из-за

ожидания сетевых сообщений. В [27] этот метод используется в качестве отправной точки для выявления преимуществ и сравнения более развитых методов.

Оригинальным методом выполнения транзакций, предложенным в [27], является метод *спекулятивного* выполнения. Основная идея состоит в том, что когда в некотором потоке управления основного раздела возникает задержка из-за выполнения двухфазного протокола распределенной фиксации транзакций, этот поток не простаивает, а выполняет другие фрагменты транзакций из своей входной очереди. Это выполнение на фоне фиксации транзакции является спекулятивным (*рискованным*), потому что результаты выполненных таким образом транзакций будут некорректными, если транзакция, на фоне фиксации которой эти транзакции выполняются, не зафиксируется, а откатится. Поэтому все спекулятивно выполненные транзакции не фиксируются до тех пор, пока не будет зафиксирована первая транзакция, в случае ее отката – все тоже откатываются (тем самым, для всех спекулятивно выполняемых транзакций необходимо поддерживать журналы откатов). Очевидно, что спекулятивная схема обеспечивает сериальный план выполнения транзакций.

Достаточно проста схема спекулятивного выполнения одноразделных транзакций. Для каждого потока управления основного раздела поддерживаются две очереди: очередь *невыполненных* транзакций и очередь *незафиксированных* транзакций (при спекулятивном выполнении первой в этой очереди всегда будет многораздельная транзакция, на фоне фиксации которой происходит спекулятивное выполнение). Если многораздельная транзакция откатывается, то по очереди откатываются все спекулятивно выполненные одноразделные транзакции и ставятся в начало очереди невыполненных транзакций для повторного выполнения (*в [27] по этому поводу ничего не сказано, но похоже, что в этом случае перед откатом каждого фрагмента требуется послать требования отката во все потоки управления соответствующих резервных разделов; нельзя ставить одноразделную транзакцию в очередь для повторного выполнения, пока не выполнен откат резервных разделов. – С.К.*). Если многораздельная транзакция фиксируется, то окончательно фиксируются и все транзакции из очереди незафиксированных транзакций, т.е. они удаляются из этой очереди, и их результаты посылаются в приложение. После обработки всех спекулятивно выполненных незафиксированных транзакций система возобновляет выполнение транзакций в не спекулятивном режиме.

Что касается спекулятивного выполнения многораздельных транзакций, то в [27] предлагается поддержка такого выполнения только для *простых многораздельных транзакций*, т.е. для транзакций, в которых с каждым разделом базы данных работает не более чем один фрагмент. Отмечается распространенность таких транзакций в транзакционных рабочих нагрузках

(например, все транзакции эталонного тестового набора TPC-C относятся к этой категории).

Пусть, например, в некотором потоке управления основного раздела на фоне выполнения двухфазного протокола фиксации многораздельной транзакции T_1 после спекулятивного выполнения транзакций T_2, ..., T_{n-1} спекулятивно выполняется фрагмент другой многораздельной транзакции T_n. Тогда после завершения выполнения этого фрагмента координатору посылаются его результаты (вместе с подтверждением готовности к фиксации T_n), а также указывается, что эти результаты зависят от T_1.

Если координатор принимает решение зафиксировать T_1, то спекулятивные результаты T_n являются корректными, и координатор может учесть это при фиксации T_n. В этом случае окончательно фиксируются транзакции T_1, ..., T_{n-1}, а на фоне ожидания решения координатора о фиксации транзакции T_n продолжается спекулятивное выполнение транзакций из очереди необработанных транзакций потока управления.

Если же координатор принимает решение об откате T_1, то он просто отбрасывает результаты транзакции T_n, полученные из данного потока управления. При обработке аварийного завершения T_1 откатываются и заново ставятся в очередь необработанных фрагментов все спекулятивно выполненные транзакции, включая T_1.

Достоинством метода спекуляций является возможность избежать простоев в работе потоков управления основных разделов без использования синхронизационных блокировок и отслеживания наборов чтения и записи транзакций. Недостатками и ограничениями являются:

- возможность перехода в спекулятивный режим выполнения только после обработки последнего фрагмента многораздельной транзакции (т.е. простои между выполнением последовательных фрагментов одной и той же транзакции возможны);
- потребность в едином координаторе для все многораздельных транзакций, чтобы можно было допустить их спекулятивное выполнение (как отмечалось ранее, центральный координатор может ограничивать потенциально возможную пропускную способность системы);
- потенциальная возможность лишних откатов, поскольку неявно предполагается, что все транзакции, работающие с одним и тем же разделом базы данных, конфликтуют.

Наконец, последняя схема, рассматриваемая в [27], основывается на *синхронизационных блокировках*. Идейно эта схема очень близка к схеме применяемой в [15] (см. подраздел 3.2). Поскольку все фрагменты транзакций, адресуемые одному разделу базы данных, выполняются в одном потоке управления, нет потребности в менеджере блокировок, доступ к которому производился бы из разных потоков управления. Тем самым, нет потребности

в какой-либо синхронизации доступа к структурам данных, разделяемым между несколькими потоками управления. Более того, если в некотором потоке управления отсутствуют активные (незафиксированные) многораздельные транзакции, то поступающие в него одноразделные транзакции могут выполняться без блокировок (и без журналов откатов, если в них невозможно аварийное завершение по инициативе пользователей). Но как только в этом потоке начинает обрабатываться многораздельная транзакция, все остальные транзакции должны выполняться с блокировками и с поддержкой журналов откатов (для разрешения возможных тупиковых ситуаций).

В этом случае перед выполнением любой операции чтения или изменения записи в некотором фрагменте транзакции производится попытка установки синхронизационной блокировки этой записи в соответствующем режиме. Если эта попытка удается, выполнение фрагмента продолжается, иначе выполнение приостанавливается, фрагмент заносится в список блокированных фрагментов, и выбирается следующий фрагмент из очереди необработанных фрагментов. Аналогично, следующий фрагмент выбирается на обработку и при завершении выполнения очередного фрагмента.

Если завершается последний фрагмент многоузловой транзакции, то все ее выполненные фрагменты посылаются в потоки управления резервных разделов. Там они выполняются последовательно без всякой синхронизации, поскольку синхронизационные блокировки продолжают удерживаться в потоке основного раздела. После выполнения фрагментов транзакции в резервных узлах она становится готовой к фиксации в данном потоке управления.

Как точно выполняется фиксация многораздельной транзакции при использовании синхронизационных блокировок, в [27] не описывается. Говорится только, что в этом случае не требуется центральный координатор, поскольку сериализацию транзакций гарантирует поддержка строго двухфазного протокола фиксации в потоках управления каждого основного раздела, и фрагменты многораздельной транзакции рассылаются по потокам управления прямо из клиентов. Думаю, что имеется в виду, что каждый клиент координирует свою распределенную транзакцию, поскольку непонятно, как обойтись без использования двухфазной фиксации.

По всей видимости, именно этому "частному" координатору после завершения последнего фрагмента транзакции в некотором потоке управления посылается сообщение, подтверждающее готовность к фиксации. Во время ожидания последнего сообщения от координатора в этом потоке управления могут выполняться другие фрагменты. Если от координатора поступает сообщение "фиксация", транзакция может освободить свои блокировки данных в соответствующем разделе и, тем самым, обеспечить возможность продолжения выполнения некоторых ранее блокированных фрагментов. Если

же от координатора приходит сообщение "аварийное завершение", то транзакция должна послать всем потокам управления резервных разделов сообщение, требующее отката всех фрагментов, дождаться подтверждения (в это время могут выполняться фрагменты других транзакций), выполнить собственный откат и, в конце концов, освободить блокировки. Еще раз замечу, что это мои собственные домыслы, поскольку в [27] об этом ничего не говорится.

Естественно, при использовании схемы с синхронизационными блокировками возможно возникновение как локальных (в пределах одного потока управления), так и распределенных (затрагивающих блокировки объектов разных разделов в разных потоках управления) синхронизационных тупиков. Локальные тупики выявляются путем обнаружения циклов в графах ожидания, а распределенные – за счет таймаутов. При обнаружении тупика, для его разрешения, система старается аварийно завершать одноразделные транзакции, поскольку их дешевле выполнить повторно (*кроме того, как мне представляется, само по себе аварийное принудительное завершение распределенной транзакции тоже стоит недешево – С.К.*).

Авторы [27] выполнили ряд интересных и содержательных экспериментов, которые, в частности, показали, что:

- блокировочная схема почти всегда существенно уступает и спекулятивной схеме, и схеме с синхронизационными блокировками;
- спекулятивная схема работает значительно лучше двух других схем при наличии многораздельных транзакций с одним циклом коммуникаций транзакции с координатором (т.е. для *простых многораздельных транзакций*, в которых для каждого используемого раздела имеется один фрагмент) и небольшой доли аварийно завершающихся транзакций;
- схема с синхронизационными блокировками обеспечивает наилучшие результаты при наличии многих транзакций с несколькими циклами коммуникаций (*для справедливости замечу, что спекулятивный способ выполнения таких транзакций настолько сложен и непонятен, что авторы его даже и не описывают – С.К.*).

Общий вывод состоит в том, что в системе стоило бы вести статистику разновидностей транзакций во время выполнения и выбирать метод, следуя, например, модели, показанной в табл. 1.

		Редкие аварийные завершения		Частые аварийные завершения	
		Мало конфликтов	Много конфликтов	Мало конфликтов	Много конфликтов
Мало транзакций с несколькими циклами коммуникации	Много многораздных транзакций.	Спекулятивное выполнение	Спекулятивное выполнение	Синхрятивное блокирование	Синхрятивное блокирование и спекулятивное выполнение
	Мало многораздных транзакций.	Спекулятивное выполнение	Спекулятивное выполнение	Блокировка выполнения или синхронизация блокировки	Блокировка выполнения
Много транзакций с несколькими циклами коммуникации		Синхронизация блокировки	Синхронизация блокировки	Синхронизация блокировки	Синхронизация блокировки

Табл. 1. Наилучшие схемы управления параллелизмом для разных ситуаций.

3.1.4. Детерминированное выполнение транзакций

Как видно из материала предыдущих пунктов, в H-Store имеются две основные проблемы, обе связанные с поддержкой многораздельных транзакций: отображение результатов транзакции в репликах базы данных и двухфазная фиксация. В [28] полагается, что эти проблемы можно решить путем перехода к полностью детерминированной схеме выполнения транзакций. (*Замечу, что при этом описываемое исследование выполнено не в контексте общей архитектуры H-Store, хотя имеет явное отношение к этому проекту. – С.К.*) Связанное с поддержкой свойств ACID требование *сериализации* транзакций, по мнению авторов [28], традиционно формулируется слишком слабо, поскольку требуется эквивалентность плана выполнения смеси транзакций *какому-либо* плану их последовательного выполнения, что оставляет простор для недетерминизма. При *детерминированной* сериализации смеси транзакций $\{T_1, T_2, ..., T_n\}$ требуется эвивалентность плана выполнения этих транзакций некоторому *предопределенному* последовательному плану их выполнения ($T_{i1}, T_{i2}, ..., T_{in}$).

Простейшим способом детерминированного выполнения смеси транзакций, гарантирующим эквивалентность предопределенному последовательному плану было бы последовательное выполнение транзакций в порядке этого

плана без какого-либо параллелизма. Однако в большинстве случаев это привело бы к неоптимальному использованию компьютерных ресурсов и, тем самым, к плохой производительности системы. Поэтому авторы предлагают использовать синхронизационные блокировки, но при соблюдении следующих ограничений, гарантирующих эквивалентность получаемого сериального плана предопределенному плану:

- Если двум транзакции T_i и T_j требуются блокировки одной и той же записи *r*, и в предопределенном плане T_i находится раньше, чем T_j, то T_i должна запросить блокировку *r раньше*, чем T_j (т.е. должна использоваться схема *упорядоченного запроса блокировок* – помимо прочего, легко видеть, что при использовании такой схемы невозможно возникновение синхронизационных тупиков).

- Каждая транзакция, не ожидающая удовлетворения запроса блокировки, должна продолжать выполняться до тех пор, пока не зафиксируется или не будет аварийно завершена *детерминированным образом* (т.е. в соответствии с логикой приложения). Если выполнение какой-либо транзакции задерживается (например, из-за какого-то сбоя в системе), то система должна поддерживать эту транзакцию активной, пока она не завершится, или пока не будет ликвидирована сама система.

Эксперименты авторов показывают, что если возможны длительные задержки при выполнении какой-либо транзакции, то детерминированная схема приводит к быстрому "загромождению" системы блокированными транзакциями и резкому падению производительности. Поэтому, в частности, детерминированная схема непригодна для СУБД, работающих с базами данных в дисковой памяти. Однако в системах, обрабатывающих данные исключительно в основной памяти, ситуация меняется. В ряде случаев детерминизм позволяет повысить производительность систем, упрощая при этом репликацию и избавляя от потребности в двухфазном протоколе фиксации распределенных транзакций.

Возможная архитектура детерминированной реплицированной СУБД показана на рис. 3. Запросы на образование транзакций (как и раньше, транзакции являются заранее определенными и сохраняемыми в виде хранимых процедур) от пользовательских приложений поступают в *препроцессор*, являющий границей детерминированной системы. Препроцессор выполняет всю необходимую недерминированную подготовительную работу (например, параметризует транзакции в соответствии с указаниями пользователей) и упорядочивает транзакции. После этого транзакции объединяются в пакет и надежно сохраняются. С этого момента система обязуется выполнить все поступившие транзакции, и все выполнение далее производится в соответствии с установленным в пакете

порядком выполнения транзакций. Наконец, пакет транзакций надежным образом отсылается во все системы, содержащие реплики базы данных (в этой работе неявно полагается, что базы данных реплицируются целиком, так что любую транзакцию можно полностью выполнить в любой реплике).

Рис. 3. Детерминированная система.

От каждой системы-реплики требуется только то, чтобы в ней была реализована некоторая модель выполнения, гарантирующая отсутствие синхронизационных тупиков и эквивалентность порядку выполнения транзакций, установленному препроцессором. Все системы реплики работают полностью независимо (*по всей видимости, сообщая препроцессору о завершении обработки полученного пакета транзакций – С.К.*). В случае отказа какой-либо системы реплики ее восстановление производится на основе реплик, сохранивших работоспособность (*кстати, в контексте [28] это не очень важно, поскольку эксперименты производились на прототипе, не поддерживающем репликацию, – С.К.*).

Для поддержки упорядоченности запросов блокировок в [28] предлагается запрашивать все блокировки, требуемые для каждой транзакции, перед началом ее выполнения (*заметим, что если, как в предыдущем пункте, для*

каждого раздела базы данных используется только один поток управления, то, как и раньше, до появления первой многораздельной транзакции все предшествующие одноразделные транзакции можно выполнять без синхронизационных блокировок – С.К.). Если это возможно, то после этого транзакция выполняется вплоть до своего завершения (над данным разделом), не освобождая блокировок. Однако не для всех транзакций заранее известно, какие записи в них будут читаться и изменяться. Например, возможна следующая транзакция:

```
T(x):
y := read(x)
write(y)
```

Здесь параметр x указывает на некоторую запись, содержащую значение первичного ключа той записи, которую требуется обновить. В этом случае невозможно сразу запросить синхронизационную блокировку второй записи, поскольку значение ее первичного ключа неизвестно до выполнения первой операции чтения. Такие транзакции в [28] называются *зависимыми*. В предлагаемой схеме зависимые транзакции разбиваются на несколько транзакций, из которых все транзакции, кроме последней, занимаются исключительно выяснением состава наборов чтения и записи, а последняя транзакция начинает свое выполнение при наличии полного знания наборов записей, которые она будет читать и изменять. Например, транзакцию T можно разбить на следующие транзакции T_1 и T_2:

```
T1(x):
    y := read(x)
    запрос_следующей_транзакции(T2(x, y))
T2(x, y):
    y´ := read(x)
    if (y´ ≠ y)
        запрос_следующей_транзакции(T2(x, y))
        abort()
    else
        write(y)
```

Препроцессор не включает транзакцию T_2 в пакеты транзакций, пока не получит результат транзакции T_1. При начале выполнения T_2 блокируются записи с ключами, содержащимися в x и y, а затем проверяется, что за время, прошедшее между завершением T_1 и началом T_2 не изменилось содержимое записи, на которую указывает x. Если эта проверка оказывается успешной, выполнение T_2 продолжается. В противном случае T_2 аварийно завершается с освобождением своих блокировок. Об этом оповещается препроцессор, который снова включает T_2 в следующий пакет транзакций. Все действия по аварийному завершению транзакций и выполнению новых попыток детерминированы; они одинаковым образом выполняются во всех системах-репликах.

В этом примере для разбиения транзакции *T* требуется только одна дополнительная транзакция (в *T* имеется зависимость *первого порядка)*. По наблюдениям авторов [28], такие транзакции часто встречаются в реальных рабочих нагрузках OLTP. Транзакции с зависимостями более высокого порядка встречаются реже, но теоретически с ними можно справиться с помощью того же приема. Эксперименты и аналитическое моделирование показали, что наличие в рабочей нагрузке OLTP транзакций с зависимостями первого порядка не может служить основанием для отказа от детерминированной схемы выполнения транзакций.

Наиболее интересен детерминизм при выполнении многораздельных транзакций. Общая схема выполнения многораздельной транзакции в [28] не описывается, но идея состоит в том, что препроцессор разбивает каждую многораздельную транзакцию на фрагменты, каждый из которых затрагивает данные только одного раздела. Для каждого фрагмента обеспечивается информация о том, какие сообщения могут поступить от других фрагментов (включая сообщения, содержащие данные, и сообщения о детерминированном аварийном завершении). Фрагмент, получивший сообщение об аварийном завершении, освобождает свои синхронизационные блокировки и завершается. Фрагмент, получивший от других фрагментов все требуемые данные, выполняется до своего логического конца и освобождает синхронизационные блокировки. Не требуется использование какого-либо протокола фиксации транзакции, поскольку отказ любого узла означает отказ данной системы-реплики. Транзакция зафиксируется в какой-либо другой реплике, и на ее основе будет восстановлена отказавшая система.

Еще раз подчеркну, что это только идея. Думаю, что авторы [28] сами до конца не продумали эту схему, которая в общем случае может оказаться очень нетривиальной. В своих экспериментах, демонстрирующих преимущество детерминированного выполнения многораздельных транзакций, они использовали очень простые транзакции из тестового набора TPC-C, а на общий случай пока не замахивались. В этой статье я не берусь разобраться со всеми возникающими сложностями, но нельзя не согласиться, что перспектива обойтись без двухфазного протокола фиксаций за счет детерминированного выполнения транзакций кажется очень привлекательной и потенциально достижимой. Как бы только не оказалось, что для этого требуется еще более сложный статический анализ транзакций, чем тот, который упоминался ранее в этом подразделе.

3.1.5. Автоматизация методов разделения и реплицирования баз данных

Понятно, что как не борись с двухфазной фиксацией распределенных транзакций, многораздельные транзакции останутся более дорогостоящими, чем одноразделные. И хотя в общем случае от многораздельных транзакций никуда не денешься, нужно стремиться к тому, что в любой рабочей нагрузке

OLTP их было как можно меньше. Понятно, что поскольку для любой системы баз данных рабочая нагрузка является независимым внешним фактором, можно лишь стремиться физическим образом организовать разделенную и реплицированную базу данных таким образом, чтобы среди транзакций именно этой рабочей нагрузки было по возможности меньше многораздельных транзакций.

Методы циклического разделения (каждый следующий кортеж направляется в следующий раздел) и хэш-разделения (кортежу назначается раздел в соответствии со значением некоторой хэш-функции от значения его атрибута разделения), часто с успехом применяемые в аналитических массивно-параллельных системах баз данных, как правило, не подходят для транзакционных массивно-параллельных систем баз данных, поскольку способствуют появлению большого числа многораздельных транзакций. Хорошие результаты может обеспечить разделение по диапазонам значений кортежей (выбор раздела для данного кортежа основывается на вычислении логического выражения, построенного на основе вхождений значений атрибутов кортежа в заданные диапазоны значений), но выбор соответствующих диапазонов с учетом заданной рабочей нагрузки вручную производить очень трудно.

Подход к решению этой проблемы представлен в [14]. В общих словах, на основе одноразделного представления базы данных и заданной рабочей нагрузки производится разделение базы данных на заданное число сбалансированных разделов с целью минимизации числа многораздельных транзакций в рабочей нагрузке. Далее система пытается аппроксимировать полученное разделение разделением по диапазонам значений на основе автоматически производимого условного выражения. Наконец, производится сравнение числа распределенных транзакций в исходной рабочей нагрузке, которые образуются при применении построенного метода разделения, с числом распределенных транзакций, которые возникают при использовании хэш-разделения и разделения на уровне таблиц, и для реального использования выбирается метод, обеспечивающий наилучший результат.

На первом этапе строится граф, в котором вершины соответствуют кортежам всех таблиц базы данных, и дуги связывают все кортежи, используемые в одной и той же транзакции. Вес каждой дуги – число транзакций, обращающихся к данной паре кортежей. На рис. 5 изображен граф для базы данных, состоящей из одной таблицы `ACCOUNT` с пятью кортежами, и рабочей нагрузки из четырех транзакций, показанных на рис. 4. На рис. 5 показаны четыре части графа, соответствующие четырем транзакциям с рис. 4.

База данных

ACCOUNT

id	name	balance
1	Carlo	80K
2	Evan	100K
3	Sam	129K
4	Eugene	29K
5	Yang	12K

Рабочая нагрузка

I

```
BEGIN
UPDATE account SET bal=bal-1k
  WHERE name="carlo";
UPDATE account SET bal=bal+1k
  WHERE name="evan";
COMMIT
```

II

```
BEGIN
UPDATE account SET bal=60k
  WHERE id=2;
SELECT * FROM account
  WHERE id=5;
COMMIT
```

III

```
BEGIN
SELECT * FROM account
  WHERE id IN {1,3};
ABORT
```

IV

```
BEGIN
UPDATE account SET bal=bal+1k
  WHERE bal < 100k;
COMMIT
```

Рис. 4. Примерные база данных и рабочая нагрузка

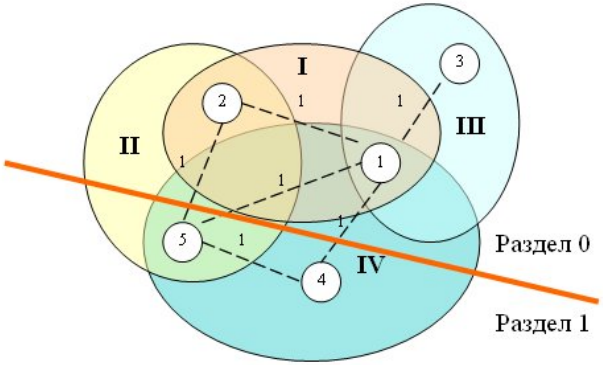

Рис. 5. Общая идея графа, используемого для разделения базы данных.

На рис. 6 показано расширенное представление графа с учетом возможности репликации на уровне кортежей. Для этого каждая вершина, соответствующая кортежу, к которому обращается $n > 1$ транзакций, заменяется "звездообразным" подграфом из n+1 вершин. Веса дуг, соединяющих вершины-реплики с центральной вершиной, характеризуют стоимость репликации данного кортежа и определяются как число транзакций в данной рабочей нагрузке, обновляющих данный кортеж. Например, на рис. 6 показано, что кортеж с id=1 представлен четырьмя вершинами, поскольку к нему обращаются три транзакции (I, III и IV). Веса дуг у соответствующего подграфа равны 2, поскольку только транзакции I и IV обновляют этот кортеж. Такая графовая структура позволяет алгоритму разделения соблюдать баланс между стоимостью репликации и выигрышем от ее применения.

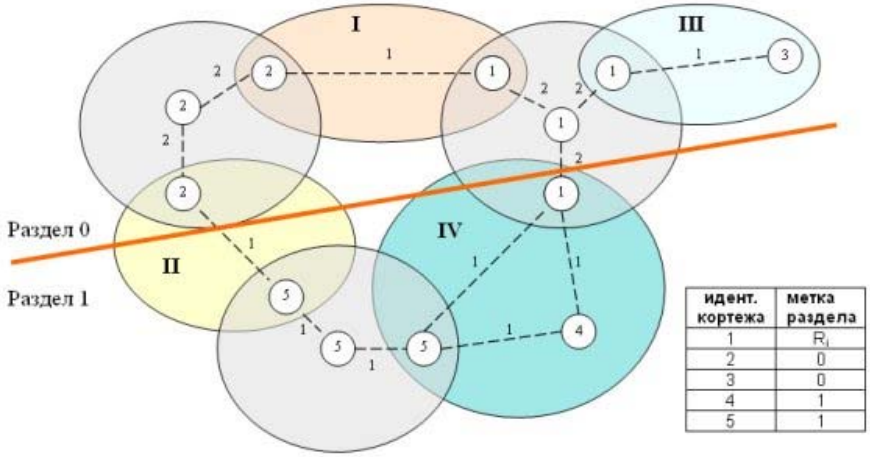

Рис. 6. Граф с учетом возможности репликации

Далее этот граф расщепляется на k разделов без общих вершин с минимизацией общей стоимости разрезания дуг (суммы весов дуг, концы которых оказываются в разных разделах графа). Другим ограничением является примерная балансировка весов разделов (допустимое отклонение является параметром системы). Весом раздела считается сумма весов вершин, входящих в этот раздел. Вес узла можно определять по-разному, и авторы [14] экспериментировали со случаями, когда вес вершины равен числу байт в соответствующем кортеже (в этом случае выполняется балансировка по размеру базы данных) и когда вес раздела равен числу обращений к кортежу (балансировка по рабочей нагрузке).

Расщепление графа на *k* частей при наличии подобных ограничений – это *NP*-полная проблема. Однако оказывается, что подобные задачи часто приходится решать в области автоматизации проектирования сверхбольших интегральных схем. За последние десятилетия были найдены сложные эвристические правила расщепления графов и созданы хорошо оптимизированные свободно доступные библиотеки программного обеспечения, позволяющие обрабатывать графы с сотнями миллионов дуг. В [14] использовались программные средства *METIS* [31]. Утверждается, что расщепление графа "производится быстро" (хотя абсолютные цифры не приводятся). Вместе с тем, отмечается, что при росте графа скорость обработки быстро возрастает, из-за чего авторы выработали ряд эвристик, позволяющих сдерживать размер графов (см. ниже).

В результате расщепления графа порождается отображение вершин-кортежей на набор меток разделов. Одним из способов использования этого результата является сохранение отображения в некоторой таблице типа той, которая показана в правом нижнем углу рис. 6. Как видно из этой таблицы, для нереплицируемых кортежей можно прямо указать номер раздела. Реплицируемые кортежи помечаются специальным образом, позволяющим понять, в каких разделах должны размещаться реплики.

В распространенном случае, когда в разделах `WHERE` операторов SQL содержатся условия сравнения на равенство с константой или вхождения в диапазон заданных значений, поисковые таблицы можно непосредственно использовать для направления операции в соответствующий(ие) раздел(ы). При наличии плотного множества идентификаторов кортежей и не более 256 разделов в 16 гигабайтной основной памяти можно хранить таблицу о разделении 15 миллиардов кортежей. Кортежи, заново вставляемые в базу данных, сначала могут помещаться в произвольные разделы, а после пересчета разделения графа их можно переместить в нужные разделы. Однако для очень крупных систем баз данных при наличии рабочей нагрузки с интенсивной вставкой кортежей этот подход может оказаться неудовлетворительным. Поэтому авторы [14] разработали дополнительное инструментальное средство, позволяющее аппроксимировать разделение, получаемое при обработке графа, методом разделения по диапазонам значений.

Это инструментальное средство основывается на методах машинного обучения, и в нем активно используются возможности свободно доступного пакета программных средств интеллектуального анализа данных WEKA [32]. Сначала на основе трассы рабочей нагрузки создается обучающая выборка. Для сокращения времени работы из трассы выделяется представительные образцы кортежей, которые помечаются метками разделов, полученными при расщеплении графа.

Затем разбираются операторы SQL, присутствующие в трассе рабочей нагрузки, и выделяются атрибуты кортежей, наиболее часто присутствующие

в условиях разделов `WHERE`. Выбранные атрибуты обрабатываются компонентом *отбора признаков (feature selection)* пакета WEKA, которые отбирает атрибуты, коррелирующие с метками разделов.

Наконец, на основе обучающей выборки и отобранного набора атрибутов строится классификатор в виде дерева решений (используется реализация J48 из пакета WEKA). На выходе классификатора получается набор условий, аппроксимирующих разделение на уровне кортежей, которое было произведено при расщеплении графа.

Авторы называют процедуру получения аппроксимирующих условий *толкованием (explanation)* разделения графа. Отмечается, что получить разумное толкование не всегда возможно, и толкование является полезным только при выполнении следующих условий: (1) оно основывается на атрибутах, часто используемых в запросах; (2) не слишком снижает качество разделения за счет неправильной классификации кортежей; (3) успешно работает для операторов SQL, не использованных для построения обучающей выборки.

Последним шагом подхода Schism при выборе метода разделения является сравнение числа распределенных транзакций, которые обеспечиваются при заданной рабочей нагрузке схемой поисковой таблицы, полученной при разделении графа; схемой разделения по условиям вхождения в диапазоны значений, сгенерированной при толковании разделения графа; схемой хэш-разделения по наиболее часто используемым атрибутам и схемой репликации базы данных на уровне таблиц. Выбирается схема, приводящая к наименьшему числу распределенных транзакций. Если несколько схем обеспечивают близкие результаты, то выбирается наименее сложная схема.

Поскольку с ростом графа, связывающего рабочую нагрузку с кортежами базы данных, значительно возрастает время его расщепления, в [14] предлагается ряд эвристик, позволяющих сократить размер графа без существенного влияния на результаты разделения, толкования и т.д. К числу этих эвристик относятся взятие образцов на уровне транзакций, взятие образцов на уровне кортежей, отбрасывание операций SQL, приводящих к сканированию больших частей таблиц и т.д. Эксперименты, описанные в [14], показывают, что подход Schism позволяет добиться качества разделения баз данных, соизмеримого с качеством наилучших схем, получаемых вручную.

Подводя итог обсуждению разных аспектов проекта H-Store, еще раз отмечу его основные черты:

- бескомпромиссное использование подхода shared-nothing – каждому разделу базы данных (основному или резервному) соответствует в точности один поток управления, и в потоках управления не используются общие ресурсы, даже если они реализуются ядрами одного и того же процессора;

- поддержка баз данных в основной памяти; долговременность хранения обеспечивается только за счет репликации данных в разных узлах кластера;
- выполнение транзакций поблизости от данных без потребности в передаче по сети операций SQL и их результатов;
- стремление к минимизации числа распределенных транзакций за счет статического анализа и преобразований транзакций, а также за счет разделения данных с учетом рабочей нагрузки.

Последнее желание понятно, поскольку подход H-Store показывает чудеса производительности именно при наличии только одноразделных транзакций. К сожалению, даже если удастся реализовать все мыслимые и немыслимые преобразования транзакций, появление распределенных транзакций полностью исключить не удастся, и поэтому основной текущей задачей проекта H-Store является нахождение способов выполнения распределенных транзакций, которые позволили бы свести к минимуму накладные расходы на их фиксацию.

Несмотря на наличие ряда нерешенных проблем, на основе промежуточных результатов проекта H-Store успешно стартовала компания VoltDB. А это означает, что имеются пользователи, для которых новый уровень производительности транзакционных систем важнее технологической завершенности предлагаемых решений.

3.2. DORA: почти «shared-nothing» в среде «shared-everything»

Подход H-Store с отказом от совместного использования ресурсов лишен компромиссов: в этой архитектуре любой аппаратно поддерживаемый поток управления рассматривается как отдельный узел системы со своими собственными ресурсами. В этой бескомпромиссности кроется много плюсов: в частности, параллельная СУБД строится в виде набора полностью идентичных компонентов, каждый из которых обладает единственной активностью. Но, с другой стороны, при этом приносятся в жертву аппаратные возможности многоядерных процессоров, позволяющие *на физическом уровне* использовать все ресурсы компьютера в потоках управления всех ядер.

В связи с этим мне кажется очень поучительным проект DORA [15], в котором разрабатывается архитектура СУБД, обладающая свойствами shared-nothing на логическом уровне, но использующая аппаратные возможности shared-everything на физическом уровне. В качестве экспериментальной аппаратной платформы в DORA использовался компьютер Sun T5220 "Niagara II". В микропроцессоре Niagara II имеется 8 ядер, в каждом из которых на аппаратном уровне поддерживается 8 потоков управления, т.е. на уровне операционной системы в компьютере имеется 64 процессора, каждому из которых доступны все остальные ресурсы системы.

Исследование [15] выполнялось не в том же контексте, что H-Store; в DORA основной упор делается на сокращение взаимодействий с центральным менеджером блокировок. Поэтому в большей части этого раздела я опишу основные идеи DORA в том виде, как они подаются авторами [15]. Однако, с моей точки зрения, между проектами H-Store и DORA имеется более глубокая связь, чем это отмечается авторами, например, [14] и [15], и на этой связи я остановлюсь в заключение подраздела.

3.2.1. Проблемы блокировок в традиционных многопоточных СУБД

Авторы [15] использовали в своем проекте транзакционную систему управления базами данных Shore-NT [33], являющуюся модифицированным вариантом системы Shore [30] с многопотоковым ядром. В Shore-NT (как и в Shore) поддерживаются все основные возможности традиционных транзакционных СУБД: ACID-транзакции с обеспечением полной изоляции на основе иерархических блокировок, управление буферным пулом, индексы на основе B-деревьев, классическое управление журнализацией и восстановлением баз данных. Каждой транзакции назначается отдельный поток управления. Как утверждается в [15], выбор пал на Shore-NT, поскольку среди всех реализаций СУБД с открытыми исходными текстами эта система лучше всего масштабируется при росте числа ядер в процессоре [34]. Однако я думаю, что не менее важную роль при этом выборе сыграло и то, что в разработке Shore-NT активно участвовали именно участники проекта DORA.

С использованием Shore-NT были выполнены эксперименты, полностью подтвердившие выводы авторов [29] (которые экспериментировали с однопотоковым вариантом Shore) об отрицательном влиянии традиционных блокировок на производительность транзакционных СУБД. При традиционном назначении каждой транзакции отдельного потока управления в них приходится использовать большое число критических участков для координации доступа к совместно используемым ресурсам. Для организации критических участков в многоядерных процессорах приходится использовать *защелки (latch)*, основанные на применении *спинлоков* [35], что в ряде случаев вынуждает аппаратный поток управления "зависать" при входе в критический участок.

В Shore-NT (как и большинстве других многопотоковых систем с использованием общих ресурсов) каждой логической блокировке соответствует структура данных (описатель блокировки), содержащая режим блокировки, указатель на списки удовлетворенных или ожидающих удовлетворения запросов блокировки, а также *защелку*. Когда некоторая транзакция пытается получить некоторую блокировку, менеджер блокировок сначала проверяет, что для этой транзакции уже удерживаются блокировки намерений для объектов более высокого уровня, и в случае потребности

устанавливает требуемые блокировки (как уже упоминалось, в Shore-NT используется иерархическая схема блокировок, подробности об этой схеме см., например, в [36]). Если оказывается, что запрашиваемая блокировка транзакции не требуется, поскольку для нее уже удерживается подходящая блокировка более высокого уровня, текущий запрос сразу удовлетворяется.

В противном случае менеджер блокировок через хэш-таблицу ищет описатель требуемой блокировки (образуя его в случае отсутствия). Описатель блокировки "защелкивается", и запрос блокировки добавляется к списку запросов. Если запрос блокировки можно удовлетворить (требуемый объект не заблокирован или текущий режим его блокировки совместим с режимом запрашиваемой блокировки), то запрос помечается как удовлетворенный, защелка освобождается, и продолжается выполнение транзакции. Иначе запрос блокировки не удовлетворяется, и транзакция блокируется (защелка описателя блокировки при этом освобождается).

В каждой транзакции поддерживается список полученных ей блокировок. При завершении транзакции они освобождаются в хронологическом порядке. Для освобождения очередной блокировки менеджер блокировок защелкивает ее описатель, удаляет из списка запрос освобождаемой блокировки, устанавливает новый режим блокировки и удовлетворяет все отложенные запросы блокировки, которые можно удовлетворить (активизируя соответствующие транзакции). После этого защелка описателя блокировки освобождается.

По мере роста числа активных транзакций (т.е. числа доступных потоков управления, которое, очевидно, коррелирует с числом ядер процессора) растет объем работы, требуемой для удовлетворения запросов на получение или освобождение блокировок, поскольку удлиняются списки запросов блокировок. Дополнительные обходы списков блокировок нужны для выявления синхронизационных тупиков (а все это увеличивает размер критических участков). Возникающие последствия губительны для производительности системы.

На рис. 7 показана разбивка по времени работы менеджера блокировок Shore-NT при возрастании коэффициента загрузки процессора. Как видно, при слабой загрузке системы 85% времени менеджер блокировок выполняет полезную работу. Однако по мере роста загруженности процессора растут расходы на обслуживание конкуренции потоков управления. При стопроцентной загрузке процессора 85% времени работы менеджера блокировок уходит на выполнение операций над защелками.

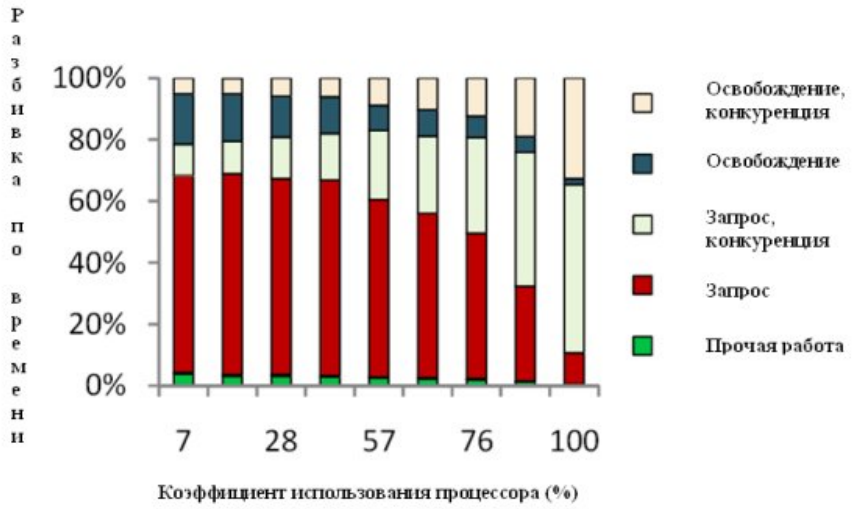

Рис. 7. Разбивка по времени работы менеджера блокировок Shore-MT при выполнении тестового набора TPC-B.

3.2.2. Архитектура DORA

Основные идеи DORA (Data-ORiented Architecture) состоят в следующем:

- потоки управления связываются не с транзакциями, а с отдельными частями базы данных;
- система распределяет работу каждой транзакции по потокам управления в соответствии с тем, к каким данным обращается транзакция;
- при обработке запросов система по мере возможности избегает взаимодействий с централизованным менеджером блокировок.

Потоки управления (*исполнители*), в которых выполняются операции транзакций, связываются с данными за счет установки для каждой таблицы базы данных *правила маршрутизации*. Правило маршрутизации разделяет соответствующую таблицу на наборы записей (фактически, разделы) так, что каждая запись относится к одному и только одному набору. Каждый набор записей приписывается одному исполнителю, и одному исполнителю может быть приписано несколько наборов записей одной таблицы. Физический доступ к данным производится через общесистемный буферный пул, и правила маршрутизации не вызывают какого-либо физического разделения или перемещения данных. Правила маршрутизации поддерживаются во время работы системы менеджером ресурсов DORA и периодически им обновляются для балансировки нагрузки. Столбцы таблицы, используемые в

соответствующем правиле маршрутизации, называются *полями маршрутизации*.

Для распределения работы каждой транзакции по исполнителям в DORA для каждой транзакции образуется *граф потока транзакции*, в котором фиксируются зависимости действий транзакции и их связь с наборами записей. Действие – это часть транзакции, заключающаяся в обращении к одной записи или небольшому набору записей одной таблицы. С каждым действием связывается *идентификатор действия*, который может содержать значения полей маршрутизации или быть пустым. Последовательные действия с одним идентификатором можно слить.

На рис. 8 показан граф потока транзакции `Payment` из тестового набора TPC-C. В соответствии со спецификацией TPC-C [37], транзакция `Payment` изменяет отстаток на счете *клиента* (изменяет некоторую строку таблицы `Customer`), отражает данные о платеже в статистике продаж соответствующих *склада* и *округа* (изменяет по одной строке в таблицах `Warehouse` и `District`) и сохраняет данные о платеже в журнале *истории* (вставляет одну строку в таблицу `History`). Первичными ключами таблиц `Warehouse`, `District` и `Customer` являются `W_ID`, `(D_W_ID, D_ID)` и `(C_W_ID, C_D_ID, C_ID)` соответственно, и если полем маршрутизации для всех трех таблиц является идентификатор склада, то этот атрибут и становится идентификатором действий в группе операций в верхней части рис. 8. Поскольку пары действий чтения и изменения строки для каждой из таблиц `Warehouse`, `District` и `Customer` обладают одинаковыми идентификаторами, их можно слить.

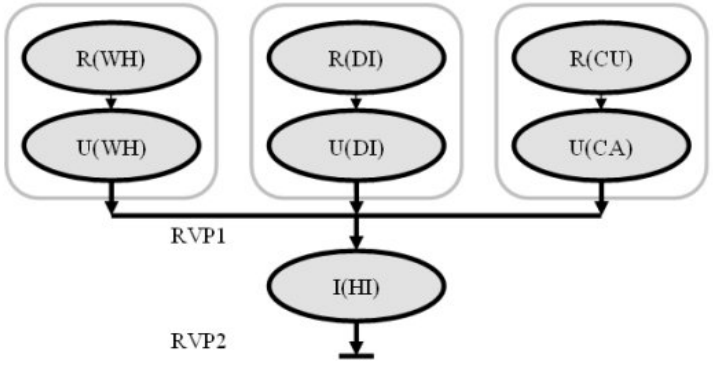

Рис. 8. Граф потока транзакции `Payment` *из тестового набора TPC-C.*

Пары действий над таблицами складов, округов и клиентов, вообще говоря, можно выполнять параллельно, поскольку между ними нет зависимостей по данным. Последнее действие транзакции `Payment` (вставку кортежа в таблицу `History`) нельзя выполнять раньше, чем завершатся все три объединенные предыдущие действия. Для управления распределенными транзакциями и передачи данных между действиями, зависимыми по данным, в DORA используются разделяемые между потоками управления объекты, называемые *точками рандеву (rendezvous point)* (*RVP*). Если между действиями транзакции имеются зависимости по данным, между ними помещается RVP.

RVP разделяют транзакции на фазы: система не может одновременно выполнять действия, относящиеся к разным фазам. В каждом объекте-RVP имеется счетчик, изначально содержащий число действий, которые должны "доложить" о своем завершении. Каждый исполнитель, завершая выполнение некоторого действия, уменьшает на единицу значение счетчика соответствующей RVP. Когда значение счетчика становится нулевым, начинается новая фаза транзакции: исполнитель, обнуливший счетчик RVP, ставит действия следующей фазы в очереди соответствующим исполнителям. Исполнитель, обнуливший счетчик последней RVP в графе потока транзакции, запрашивает ее фиксацию. Аварийно завершить транзакцию и инициировать ее откат может любой исполнитель.

Итак, к одному исполнителю направляются все действия, которые должны выполняться над одним и тем же набором записей. Исполнитель отвечает за изоляцию и упорядоченность конфликтующих действий. В каждом исполнителе поддерживаются три структуры данных: очередь поступающих действий, очередь завершенных действий и локальная таблица блокировок. Действия выполняются в том порядке, в котором они поступают во входную очередь. Для устранения конфликтов между действиями в исполнителях используются локальные таблицы блокировок. В качестве объектов блокировки применяются идентификаторы действий, и блокировки могут устанавливаться только в совместном и монопольном режимах. Каждое действие, получившее требуемую локальную блокировку, продолжает выполнение без обращения к ценрализованному менеджеру блокировок.

Локальные блокировки, полученные каждым действием транзакции, удерживаются в соответствующих исполнителях до фиксации или аварийного завершения этой транзакции. В заключительной RVP транзакция сначала дожидается ответа от основного менеджера управления данными о заключительной фиксации или аварийном завершении транзакции. После этого все действия этой транзакции ставятся в очереди завершенных действий своих исполнителей (откат транзакции в DORA производит основная система). При обработке этих действий исполнители освобождают соответствующие локальные блокировки и удовлетворяют запросы блокировок ожидающих действий.

Одна из проблем DORA состоит в том, что оказывается невозможно обойтись локальными блокировками идентификаторов действий при выполнении операций удаления и вставки строк таблиц. Действительно, возможна ситуация, когда транзакция *T*1 удаляет запись в некоторой странице, а транзакция *T*2, действие которой выполняется в другом исполнителе, вставляет на ее место свою запись до завершения транзакции *T*1. Тогда, если транзакция *T*1 завершится аварийным образом, ее нельзя будет откатить, поскольку слот удаленной записи уже занят.

Для устранения возможности такого конфликта перед выполнением операций удаления и вставки записей исполнители блокируют соответствующие им идентификаторы (т.е. номер страницы и номер слота внутри страницы). Утверждается, что потребность в таких блокировках возникает сравнительно редко, но беда в том, что эта проблема совсем не техническая – это обратная сторона того, что данные разделяются между исполнителями на логическом уровне. Получается, что без централизованной синхронизации вообще нельзя вставлять в таблицы новые строки, поскольку эти операции могут выполняться параллельно разными исполнителями, и должна обеспечиваться координация распределения памяти. В DORA имеются и другие проблемы, в частности, проблема выявления тупиковых ситуаций с участием локальных и централизованных блокировок, но я не буду на них останавливаться, поскольку, на мой взгляд, эти вопросы в [15] проработаны недостаточно глубоко.

DORA реализована поверх Shore-NT с минимальными изменениями базовой системы. В частности, были блокированы средства Shore-NT собственного управления транзакциями. Транзакции программировались в виде заранее компилируемых хранимых процедур. При тестировании системы на различных тестовых наборах было установлено, что выигрыш в производительности DORA по сравнению с базовой системой достигает 4,8 раз. При неполной загрузке системы за счет внутреннего параллелизма транзакций удается добиться сокращения времени ответа на 60%.

3.2.3. DORA, H-Store и компромиссы

Одной из наиболее серьезных проблем транзакционных параллельных СУБД без использования общих ресурсов является балансировка нагрузки. Чтобы сбалансировать нагрузку, требуется изменить разделение и/или репликацию базы данных. Фактически, нужно переслать часть записей одной или нескольких таблиц из одного раздела в другой. Пересылка данных вызывает ощутимые накладные расходы (даже если вся база данных поддерживается в основной памяти), и в каждом из изменяющихся разделов необходимо должным образом изменить существующие индексы. Во время выполнения этих операций трудно продолжать поддерживать выполнение запросов, адресуемых к данным изменяемых разделов.

В DORA физическая пересылка данных не требуется. Не требуется и массовое преобразование индексов, поскольку индексы поддерживаются для таблицы целиком. Менеджер ресурсов, выявив потребность в расширении раздела одного исполнителя (*E*1) за счет сокращения раздела другого исполнителя (*E*2), изменяет правило маршрутизации для данной таблицы и формирует служебную транзакцию из двух действий, разделенных RVP. Первое действие ставится в очередь входных действий *E*2 и приводит к тому, что *E*2 дожидается, пока не закончатся все транзакции, в обработке которых он участвует, принимая при этом для обработки новые действия, посылаемые в соответствии с новым правилом маршрутизации. После завершения всех действий, направленных ему в соответствии со старым правилом маршрутизации, *E*2 ставит в очередь действие *E*1, позволяющее этому исполнителю обрабатывать те действия, которые направляются ему в соответствии с новым правилом маршрутизации. (*Должен признаться, что эта процедура придумана мной, поскольку в [15] она описана невразумительно. – С.К.*)

Вторая проблема, с которой приходится сталкиваться системам, не использующих общих ресурсов, возникает при обработке запросов, условие выборки которых не позволяет выявить один или несколько (немного) разделов, содержащих требуемые данные. В таком случае соответствующее действие должно направляться всем исполнителям, лишь немногим из которых удастся найти то, что требуется в запросе. В DORA для выполнения таких запросов можно воспользоваться общим для таблицы "вторичным" индексом, который может помочь направить действия требуемым исполнителям. (*На самом деле, с этими "общими" индексами в перспективе не все понятно. Пока DORA живет поверх Shore-NT, поддержка работы с индексами происходит "в другом мире". Но если пытаться продумать полную архитектуру с назначением потоков управления логическим разделам данных, то в ней должно найтись место и работе с индексами, а как это делать – непонятно. – С.К.*)

Несмотря на наличие многих неясностей, основная идея DORA – при наличии в системе физически общих ресурсов производить разделение данных не на физическом, а на логическом уровне – кажется мне очень привлекательной. Прототип DORA рассчитан на традиционную работу с дисками, в нем поддерживаются общая система буферов и журнализация (на уровне Shore-NT) и поэтому:

- можно спокойно делить данные на логическом уровне с точностью до кортежа (все буферные страницы доступны всем потокам управления);
- в нем удается обойтись без двухфазного протокола фиксации транзакций;
- и он плохо согласуется с общими идеями H-Store.

Однако мне кажется, что возможен компромисс между архитектурами H-Store и DORA *внутри* одного многоядерного компьютера для поддержки баз данных в основной памяти без журнализации изменений. Для этого нужно добиться полного отсутствия потребности в централизованных блокировках. Пусть, например, каждая таблица хранится в основной памяти на основе некоторого B-дерева и кластеризуется в соответствии с его ключом (возможно, составным). Все потоки управления работают с общей виртуальной памятью, накрывающей все базу данных целиком, но каждый исполнитель получает для выполнения фрагменты транзакций, которым требуются только данные поддеревьев, приписанных к этому исполнителю. Другими словами, "верхушка" B-дерева каждой таблицы используется координатором транзакций в качестве "правила маршрутизации", а соответствующие поддеревья (определяемые диапазонами значений ключей) используются в исполнителях для доступа к кортежам. Если еще приписать каждому координатору некоторое число свободных страниц основной памяти (для обеспечения возможности автономного расширения таблиц), то при выполнении фрагментов транзакций им никогда не потребуется централизованная синхронизация.

Балансировать нагрузку в этом случае можно будет путем перераспределения поддеревьев между исполнителями, работающими с "соседними" поддеревьями. Фактически, можно легко потребовать от любого исполнителя отдать некоторую "левую" часть своего поддерева некоторой таблицы своему соседу "слева" или "правую" часть своего поддерева некоторой таблицы своему соседу "справа". По-моему, здесь сработает почти та же процедура, которая используется для балансировки нагрузки в DORA. И не потребуется никакого физического копирования данных и/или перестройки индексов, поскольку реально все данные остаются на месте.

Реплицировать, по всей видимости, придется целиком всю базу данных, поддерживаемую многоядерным компьютером. Если считать, что репликация производится для обеспечения долговечности данных и восстановления системы после отказов узлов, то трудно представить себе такой отказ, при котором перестали бы работать несколько ядер процессора, а остальные сохранили работоспособность.

Для управления прохождением транзакций в такой гибридной архитектуре можно было бы использовать любой вариант, обсуждаемый разработчиками H-Store. В зависимости от этого понадобится или не понадобится двухфазная фиксация транзакций.

Так что мне представляется, что будущее поколение ACID-транзакционных систем будет опираться на две основные параллельные архитектуры одноузловую и многоузловую. Одноузловая архитектура предполагает наличие мощного многоядерного компьютера и использование энергонезависимой внешней памяти, в которой, в частности, должен поддерживаться журнал повторного выполнения транзакций. И в этом

направлении хороший фундамент закладывает DORA. Но нужно учитывать, что для достижения высокой производительности в такой архитектуре потребуется мощная параллельная система ввода-вывода, стоимость которой, вполне вероятно, будет определять стоимость системы в целом (В [15] эксперименты выполнялись с использованием файловой системы в основной памяти именно из-за отсутствия у авторов такой дорогостоящей дисковой подсистемы.)

Многоузловая архитектура строится на основе достаточно большого числа, вообще говоря, недорогих компьютеров, связанных сетью. Базы данных хранятся только в основной памяти, для обеспечения отказоустойчивости (и, следовательно, долговечности данных) используется репликация. В этом направлении хорошей основой является H-Store (и VoltDB), хотя мне кажется, что следовало бы учесть отмеченные выше возможности использования физически общих ресурсах в многоядерных узлах подобных систем.

В заключение этого раздела замечу, что, к сожалению, эти две архитектуры являются взаимоисключающими. Трудно представить компанию, которая хорошо потратилась на приобретение мощного многоядерного сервера с дорогой дисковой подсистемой, а потом решается отказаться от использования дисков и перейти к использованию многоузловой архитектуры. Трудно представить себе и ситуацию, когда сначала был выбран подход с использованием дешевых кластерных архитектур почти без дисков, а потом вдруг покупается отдельный дорогой сервер, и компания переходит к использованию одноузловой архитектуры. Так что, скорее всего, будет продолжать существовать и направление shared disks, свойственное, например, Oracle, поскольку оно позволяет достаточно эффективно использовать кластеры, построенные с использованием мощных дисковых серверов, которые до этого использовались автономно.

4. Рационализация согласованности

Обсуждение этого направления лучше всего начать с работы [38], в которой впервые был предложен новый подход к оптимизации систем управления базами данных (немного в другом контексте этот подход обсуждался в [39]). Отмечается наличие в мире баз данных нескольких новых тенденций.

- Первой тенденцией является изменение приоритетов в требованиях приложений к системам управления данными. Для многих Web-приложений строгая согласованность данных в соответствии с парадигмой ACID не требуется. Зато им требуется возможность масштабирования приложения до миллионов пользователей, ни один из которых не должен блокироваться другими пользователями. Все запросы этих пользователей (и простые, и сложные) должны обрабатываться за гарантированное время в пределах секунды.

- Вторая тенденция состоит в возрастании сложности приложений, насыщенных данными, и в использовании для их организации распределенной сервис-ориентированной архитектуры.
- Наконец, имеются технологические тенденции, такие как "облачные" вычисления (cloud computing) и крупные центры данных, основанные на использовании дешевых аппаратных средств. Эти тенденции влияют на архитектуру программного обеспечения.

Если в "прошлой" жизни в системе управления данными *при заданном наборе аппаратных ресурсов и полноценной поддержке ACID-транзакций требовалось минимизировать время ответов на запросы и максимизировать пропускную способность системы*, то в новых условиях *при заданных требованиях к производительности приложений (пиковая пропускная способность, максимально допустимое время ответа) требуется минимизировать требуемые аппаратные ресурсы и максимизировать согласованные данные*. Сводка различий в формулировке проблемы оптимизации приведена в табл. 2.

Основным показателем системы баз данных, нуждающемся в оптимизации, является *денежная оценка затрат*. По мнению авторов, технология баз данных может удовлетворить любые требования к производительности и пропускной способности приложения, вопрос лишь в том, сколько для этого потребуется машин, т.е. сколько за это придется платить. При использовании "облачных" инфраструктур показатель расходов на аппаратные ресурсы становится непрерывным: чем больше их потребляется, тем больше приходится платить.

Характеристика	*Традиционные базы данных*	*Новые базы данных*
Денежные затраты	фиксированные	минимизируются
Производительность	оптимизируется	фиксированная
Масштабируемость (число машин)	максимизируется	фиксированная
Предсказуемость стоимости и производительности	-	фиксированная
Согласованность (в процентах)	фиксированная	максимизируется
Гибкость (число вариантов)	-	максимизируется

Табл. 2. Сопоставление традиционной и новой проблем оптимизации

В большинстве случаев сегодня *производительность* является заданным ограничением приложения, а не целью оптимизации. В интерактивных приложениях для удовлетворения потребностей пользователей достаточно *время ответа* в несколько миллисекунд. Что касается *пропускной способности*, то реальным требованием является поддержка заданной

пиковой рабочей нагрузки, причем проблемой является не *возможность* этой поддержки (возможно практически все), а ее *стоимость*. Кроме того, из-за сложности современных приложений многие проблемы их производительности вообще не связаны с управлением данными.

В настоящее время принято считать, что любую проблему приложений баз данных можно решить путем соответствующего выбора аппаратных средств (т.е. за счет определенных денежных затрат). *Масштабируемость* означает, что расходы на поддержку приложений возрастают линейно по мере роста бизнеса, и этот рост не ограничивается возможностями системы. Неограниченная масштабируемость приложений необходима для любой развивающейся компании. *Мне кажется, что требование к маштабируемости нужно формулировать точнее: при росте числа аппаратных ресурсов должно соблюдаться линейное возрастание пропускной способности и стоимости с сохранением (или уменьшением) времени ответа. – С.К.*

Во многих случаях *предсказуемость* производительности и стоимости поддержки приложений оказывается не менее важной, чем их масштабируемость. Критичные для компании показатели приложений, в том числе, пиковая пропускная способность, время ответа и стоимость, должны гарантироваться.

Что касается *согласованности*, на авторов [38] сильное влияние оказали идеи [24] и [40] (замечу, что в то время оба автора этих публикаций работали в Amazon.com). Утверждается, что ACID-транзакции плохо сочетаются с сервис-ориентированной архитектурой, которая диктует автономию сервисов, участвующих в транзакции. Кроме того, по утверждениям авторитетных практиков из Amazon.com, в современной Internet-практике ACID-транзакции требуются нечасто. Наконец, более приоритетным требованием является обеспечение стопроцентной доступности данных по чтению и записи всех пользователей. Оказывается лучше разрабатывать систему, которая умеет обращаться с несогласованными данными и помогает устранять их несогласованность, чем систему, которая предотвращает несогласованность в любых ситуациях. Одним словом, в подходе [38] согласованность данных является целью оптимизации системы, а не ее фиксированным ограничением.

Замечу, что в этих рассуждениях о согласованности одновременно имеются и здравый смысл (очевидно, что поддержка ACID-транзакций в распределенной среде стоит немалых расходов, и если целью оптимизации является минимизация расходов, то по поводу транзакций нужны какие-то компромиссы), и изрядная путаница (во многом напоминающая путаницу, с которой мы разбирались в разд. 2; один раз авторы [38], хотя и очень стеснительно, ссылаются и на теорему CAP). И мне кажется, что частично причины этой путаницы указывают сами авторы, обосновывая в своей более ранней работе [41], что "при разработке крупномасштабных распределенных систем определения уровней согласованности в духе предложений [42], а не

определений, содержащихся в стандарте языка SQL и реализуемых в коммерческих системах баз данных текущего поколения".

Но если обратиться напрямую к [42], то видно, что сами Таненбаум (Andrew S. Tanenbaum) и Стеен (Maarten van Steen) используют термин *согласованность (consistensy)* как в смысле, традиционном для области баз данных ("если у системы до начала транзакции имелись некие инварианты, которые она постоянно должна хранить, они будут сохраняться и после ее завершения" – разд. 5.6 "Распределенные транзакции"), так и в смысле, затрагивающем лишь репликацию данных ("Модель согласованности (consistency model), по существу представляет собой контракт между процессами и хранилищем данных. Он гласит, что если процессы согласны соблюдать некоторые правила, хранилище соглашается работать правильно." – Разд. 6.2 "Модели непротиворечивости, ориентированные на данные"). Легко (еще раз!) видеть, что понятие согласованности второго вида не имеет прямого отношения к транзакциям вообще и к ACID-транзакциям в частности. Оба понятия согласованности, безусловно, полезны и имеют право на жизнь, но предпочитать одно другому – это все равно, что предпочитать *красное сладкому*.

Возвращаясь к перечню характеристик систем баз данных из табл. 2, *гибкость* – это возможность настройки программной системы к индивидуальным требованиям пользователей. Гибкость системы измеряется числом ее внедренных вариантов. В традиционных системах OLTP требование гибкости отсутствует. При отсутствии гибкости проще добиться высоких значений показателей производительности и масштабируемости.

Следуя новым приоритетам характеристик систем управления данными, авторы [38] предлагают новую архитектуру, которую мы кратко опишем в следующем подразделе.

4.1. Архитектура, удовлетворяющая новым требованиям

На рис. 9 схематически показана традиционная трехзвенная архитектура организации приложений баз данных. Запросы инициируются пользователями на уровне представлений. В настоящее время для этого обычно используются Web-браузеры. На среднем уровне поддерживается логика приложений; обычно на этом же уровне работают Web-серверы. Управление базами данных целиком сосредоточено на низшем уровне, и для этого используются СУБД.

Трехзвенная архитектура ориентирована на удовлетворение требований традиционных систем баз данных (средний столбец табл. 2). Согласованность данных поддерживается в нижнем звене сервером базы данных. Производительность обеспечивается за счет применения на всех трех уровней методов, разработанных в сообществе баз данных на протяжении десятилетий – кэширование, индексация, разделение данных и т.д. На верхних двух уровнях архитектура масштабируется почти неограниченно.

Рис. 9. Традиционная трехзвенная архитектура организации приложений баз данных

Трехзвенная архитектура ориентирована на удовлетворение требований традиционных систем баз данных (средний столбец табл. 2). Согласованность данных поддерживается в нижнем звене сервером базы данных. Производительность обеспечивается за счет применения на всех трех уровней методов, разработанных в сообществе баз данных на протяжении десятилетий – кэширование, индексация, разделение данных и т.д. На верхних двух уровнях архитектура масштабируется почти неограниченно.

Классическая трехзвенная архитектура (при традиционной реализации СУБД) не отвечает требованию предсказуемости. Когда с базой данных одновременно работает много клиентов, практически невозможно понять, что происходит в СУБД (*например, когда начинается активное замещение страниц в буферном пуле – С.К.*). Возможности масштабирования на нижнем уровне архитектуры ограничены.

Гибкость в традиционной архитектуре тоже не поддерживается должным образом. Одной из причин является использование разных технологий на каждом из трех уровней: SQL на нижнем уровне, объектно-ориентированные методы на уровне приложений и XML/HTML и скриптовые языки на уровне представлений. Настройку нужно вести на всех трех уровнях с использованием разных технологий, что затруднительно и ненадежно. (*Коротко можно сказать, что традиционной архитектуре присуща известная проблема* потери соответствия (impedance mismatch) – *С.К.*)

Двумя основными принципами разработки приложений баз данных в классической трехзвенной архитектуре являются *контроль* и *передача запросов (query shipping)*. Первый принцип означает, что на нижнем уровне СУБД контролирует все аппаратные ресурсы и весь доступ к данным. Передача запросов предполагает выталкивание на нижний уровень как можно больше функций приложений за счет использования хранимых процедур, типов данных определяемых пользователями и т.д. С одной стороны, эти принципы позволяют добиться согласованности данных и высокой производительности системы. С другой стороны, по мнению авторов [38], оба эти принципа стимулируются бизнес-моделью производителей СУБД и превращают системы управления данными в монолитных монстров, размеры и сложность которых непрерывно растут. Это наносит вред предсказуемости, гибкости и масштабируемости. Кроме того, они приводят к тому, что приложения баз данных становятся дорогостоящими, поскольку для поддержки таких СУБД требуется дорогая аппаратура. В предлагаемой авторами архитектуре используются противоположные принципы разработки.

Этот абзац кажется мне очень важным, поскольку, по моему мнению, отказ от указанных принципов в действительности вызывает очень важные последствия. Отказ от принципа *контроля* приводит к тому, что СУБД перестает быть *системой*, а превращается в набор утилит. Если у СУБД отсутствует контроль над обрабатываемыми ею данными, она, естественно, не сможет поддерживать ACID-транзакции. Как мы видели в предыдущем разделе, можно делать транзакционные СУБД массивно-параллельными (фактически, использовать для их разработки методы распределенных систем), оставляя их системами с точки зрения внешнего использования.

Если отнять у СУБД контроль над аппаратными средствами и данными, то для построения распределенных систем обработки данных на всех уровнях придется применять общие методы построения распределенных систем. Наверное, это хорошо с точки зрения повсеместного использования сервис-ориентированной архитектуры, но задачи, которые хорошо решаются традиционными СУБД (в том числе, обеспечение ACID-транзакций) становятся практически неразрешимыми. Кстати, такая организация напоминает мне архитектуру файл-серверных СУБД типа Informix SE (см. например, [43]), в которой данные контролировались файловым сервером, а запросы обрабатывались на клиентских рабочих станциях. В этой архитектуре

поддерживались ACID-транзакции, но очень высокой ценой – путем блокировки файлов целиком. И не просто так впоследствие компании-производители SQL-ориентированных СУБД перешли к использованию серверов баз данных, в которых данные контролируются СУБД.

Кстати, и отказ от второго принципа возвращает нас ко времени Informix SE, где для выполнения запросов на рабочие станции из файлового сервера передавались блоки данных. Вроде бы, вполне естественно было перейти на архитектуру, в которой из клиента в сервер баз данных передавались SQL-запросы, а возвращались только те данные, которые действительно нужны приложению. И механизмы хранимых процедур и определяемых пользователями типов данных во многом появились не из-за корысти производителей СУБД, а для того, чтобы еще сократить объемы данных, которыми приходится обмениваться клиенту и серверу. И как видно из разд. 3, все это совсем не обязательно наносит вред предсказуемости и масштабируемости (насчет гибкости ничего сказать не могу, поскольку отсутствуют точные критерии).

Взамен традиционной архитектуре с рис. 9 в [38] предлагается новая архитектура, показанная на рис. 10. Эта архитектура реализована компанией 28msec в продукте Sausalito [44]. В архитектуре Sausalito основные функции СУБД – обработка запросов и управление транзакциями – переносятся на прикладной уровень. (*В действительности, не очень понятно, что же такое* транзакция *в этой системе. Подробнее об этом см. следующий подраздел. – С.К.*) На нижнем уровне поддерживается только распределенное хранение данных за счет использования службы Amazon S3 [45] (хотя теоретически можно использовать и другие аналогичные средства). Согласованность данных поддерживается не на уровне хранения, а на прикладном уровне. Отсутствует какой-либо объект, контролирующий весь доступ к данным, и согласованность данных обеспечивается в духе [42] за счет применения во всех серверах приложений общих протоколов. (*Как уже отмечалось выше, в [42] эти протоколы направлены на поддержку согласованности* реплик *данных, так что здесь имеется в виду именно этот вид согласованности. – С.К.*)

На всех уровнях можно использовать дешевые аппаратные средства, и каждый уровень может масштабироваться до тысяч машин. В любой момент времени допускается отказ любого узла. На нижнем уровне отказоустойчивость достигается за счет репликации и предоставления гарантий только *согласованности в конечном счете (eventual consistency)*. На верхних уровнях все узлы работают без сохранения состояния, поэтому при отказе какого-либо узла в самом худшем случае могут потеряться некоторые активные транзакции.

Рис. 10. Архитектура Sausalito

В Sausalito все данные и заранее откомпилированный код приложения сохраняются в среде Amazon S3 в виде BLOB'ов. При обработке каждого HTTP-запроса (поступающего, например, по инициативе пользователя из Web-браузера) служба Amazon EC2 [46] с учетом балансировки нагрузки выбирает доступный сервер EC2. В этот сервер из S3 загружается код приложения, который затем интерпретируется подсистемой поддержки времени выполнения Sausalito с доступом, при необходимости, к объектам базы данных, хранимым в S3. Особенностью Sausalito является то, что для реализации логики приложений и доступа к базе данных в системе используется язык XQuery [47], расширенный средствами обновления данных и написания скриптов. Авторы [38] мотивируют этот выбор тем, что XQuery хорошо согласуется со стандартами Web, обладает развитыми средствами

запросов, и возможностей этого языка достаточно для создания развитых Web-приложений. Однако мне кажется, что важную роль сыграла и личная близость Даниелы Флореску (Daniela Florescu) и Дональда Коссманна к процессам стандартизации и реализации этого языка. Не уверен, что (единственная) возможность использования XQuery в качестве языка разработки приложений приводит в восторг потенциальных пользователей Sausalito.

Как отмечается в [38], у системы с архитектурой с рис. 10 мало шансов сравняться с классическими системами баз данных в отношении производительности и согласованности данных (*я уже говорил выше, что это обратная сторона отказа от принципов контроля над данными и пересылки запросов – С.К.*). Однако эта архитектура хорошо соответствует требованиям правого столбца табл. 2. Архитектура экономически эффективно реализуется с использованием дешевых аппаратных средств, масштабируемость на всех уровнях обеспечивается автоматически. Гибкость достигается за счет упрощения платформы и использования на всех уровнях единой модели программирования и данных (XML/XQuery). Для многих рабочих нагрузок обеспечивается предсказуемость расходов и производительности.

В [38] изложены основные идеи "облачной" системы управления данными, хорошо согласующейся с сервис-ориентированной архитектурой, гибкой и масштабируемой на всех уровнях. На появление новой архитектуры систем баз данных повлияли сценарии интерактивных (транзакционных) Web-приложений – онлайновых магазинов и т.д. Однако, как отмечалось выше, идеология поддержки транзакций в [38] выглядит очень туманной. Некоторую ясность привносит более свежая работа [9], которой посвящен следующий подраздел.

4.2. Разным данным разная согласованность: новая парадигма транзакций?

В [9], как и в [38], речь идет о возможности реализации средств управления данными поверх "облачных" сред хранения данных. Основная идея исследования состоит в том, что не все данные требуется обрабатывать на одном и том же уровне согласованности (*в этой статье делается попытка "примирить" понятия согласованности из областей баз данных и распределенных систем, хотя, на мой взгляд, это не всегда у авторов получается – С.К.*). Например, в онлайновом магазине данные о кредитных картах клиентов и состоянии их счетов должны поддерживаться на более высоком уровне согласованности, чем данные о предпочтениях покупателей.

Поскольку существующие "облачные" платформы хранения данных предоставляют только базовые гарантии согласованности данных (согласованность в конечном счете), обеспечение дополнительных уровней согласованности значительно повышает стоимость выполнения операций. С другой стороны, стоимость несогласованности данных также можно измерить,

оценив относительное число операций, которые выполняются некорректно из-за отсутствия согласованности, и переведя эту оценку в денежные расходы, которые должна понести компания, допустившая такое некорректное выполнение операций (например, для компенсации ущерба от ошибочно выполненных заказов).

Нахождение баланса между стоимостью выполнения операций, согласованностью данных и их доступностью является нетривиальной задачей. Этот баланс зависит от нескольких факторов, включая семантику приложений. Предлагается использовать динамическую стратегию поддержки согласованности данных – снижать уровень требований к согласованности, когда это не приводит к слишком большим убыткам, и повышать их, если убытки могут оказаться слишком большими. Авторы [9] называют этот подход *рационализацией согласованности (consistency rationing)* по аналогии с понятием *рационализации управления запасами (inventory rationing)*, когда запасы отслеживаются с разной точностью в зависимости от их наличного объема.

При применении подхода рационализации согласованности все управляемые данные разделяются на три категории A, B и C, и для каждой категории применяются свои методы обработки в зависимости от обеспечиваемого уровня согласованности. К *категории A* относятся данные, нарушение согласованности которых привело бы к крупным убыткам. Несогласованность данных *категории C* является приемлемой (несогласованность либо вызывает лишь небольшие убытки, либо в действительности не возникает). Наконец, *категория B* включает данные, требования к согласованности которых изменяются во времени. Для данных категории B можно добиться оптимального соотношения расходов на выполнение операций и обеспечиваемого уровня согласованности.

Гарантии согласованности в [9] обеспечиваются для данных, а не для транзакций. Как утверждается, в результате ослабляются только свойства изоляции и согласованности ACID-транзакций, а атомарность и долговечность обеспечиваются в полном объеме. На самом деле, здесь нужно было бы привести определение возникающего понятия *транзакции*, но такое определение в [9] отсутствует, и придется обойтись предыдущим расплывчатым предложением. Для данных возможны два уровня согласованности – *сессионная согласованность (session consistency)* [48] и *сериализуемость (serializability)*.

Сессионная согласованность данных подразумевает, что клиенты подключаются к системе в контексте *сессий* (*что такое "сессия", ни в [9], ни в [48] не определяется – С.К.*). Во время сессии система гарантирует *монотонность по чтению собственных записей (read-your-own-writes monotonicity)* (*согласно [42], система "обладает свойством монотонности по чтению собственных записей, если удовлетворяется то условие, что результат записи процесса в элемент данных* x *всегда виден последующим*

операциям чтения x *того же процесса" – С.К.*). За границы сессии гарантии монотонности не распространяются: в новой сессии того же клиента могут быть не видны записи, произведенные в предыдущей сессии; в сессии одного клиента могут быть не видны записи, выполненные в сессии другого клиента. Сессионная согласованность поддерживается для данных категории C.

Для данных категории A обеспечивается сериализуемость в традиционном транзакционном смысле. Все транзакции, модифицирующие данные категории A, изолируются и оставляют данные согласованными (здесь имеются в виду и согласованность в смысле ACID, и *строгая согласованность* в смысле распределенных систем). Для поддержки сериализуемости требуются значительные накладные расходы, и данные следует относить к категории A, только если их согласованность и актуальность являются обязательными. В [9] для поддержки сериализуемости используется двухфазный протокол синхронизационных блокировок.

Для данных категории B во время работы системы на основе разработанных авторами политик производится переключение между режимами сессионной согласованности и сериализуемости. Если одна транзакция работает с некоторой записью в режиме сериализуемости, а другая обновляет ее в режиме сессионной согласованности, то обеспечивается сессионная согласованность (*т.е., насколько я понимаю, транзакции, работающие с данными категории B в режиме сериализуемости, не изолируются от транзакций, работающих с теми же данными в режиме сессионной согласованности – С.К.*).

Если в одной транзакции одновременно обрабатываются данные категорий A и C, то при выполнении операции чтения данных категории A эта транзакция всегда получает их последнее согласованное состояние и оставляет их согласованными после своей фиксации. При чтении данных категории C транзакция может получить их в несогласованном и/или устаревшем состоянии.

Как считают авторы [9], в ряде случаев можно будет разделить процессы разработки приложений и рационализации данных. В процессе разработки можно предполагать наличие традиционной согласованности данных и следовать обычной модели программирования с явно определяемыми транзакциями. На стадии внедрения приложения можно произвести рационализацию данных в соответствии с оценками расходов на выполнение операций и ущерба от несогласованности данных. Рационализацию мог бы производить специалист не из числа разработчиков приложения.

В [9] подробно описываются разработанные авторами политики переключения режимов согласованности для данных категории B, однако в данной статье я не хочу затрагивать эту тему, поскольку для наших целей она не очень важна. Не буду также говорить о деталях реализации разнообразных протоколов поддержки транзакций и согласованности данных, потому что в

[9] эти вопросы освещены не слишком внятно, а в подробном техническом отчете [49] соответствующие описания занимают слишком много места.

Главный вывод, который, на мой взгляд, следует сделать на основе публикаций, посвященных Sausalito, состоит в том, что разработчики этой системы вполне смогли справиться с поддержкой ACID-транзакций и ослабляют требования к согласованности совсем не на основе ограничений, которые ставит теорема CAP, а с целью вполне разумной оптимизации стоимости службы управления данными.

5. Заключение

Основные цели этой статьи заключались в обсуждении истинной сути теоремы CAP и рассмотрении нескольких новых архитектур транзакционных СУБД, хорошо масштабируемых в динамических кластерных средах. Я попытался показать, что теорема CAP Эрика Брювера не имеет никакого отношения к возможности или невозможности поддержки ACID-транзакций в горизонтально масштабируемых кластерных системах.

Проект H-Store показывает, что в параллельной архитектуре shared-nothing основную проблему представляют распределенные транзакции, накладные расходы на фиксацию которых могут заметно снижать пропускную способность системы. Однако полученные результаты показывают, что для ряда важных типов рабочей нагрузки этих расходов можно избежать.

Другую проблему транзакционных параллельных систем без совместно используемых ресурсов составляет потребность в массовой физической пересылке данных при балансировке нагрузки. И здесь, как мне кажется, может несколько помочь подход проекта DORA, в котором вместо физического разделения данных в многоядерном компьютере используется их логическое разделение за счет наличия общей памяти.

Наконец, подход Sausalito показывает, во-первых, что при отказе от основных принципов разработки СУБД – контроля над данными и передачи запросов – в угоду более точному следованию архитектуре SOA расходы на поддержку ACID-транзакций существенно возрастают. Во-вторых, эта поддержка по-прежнему возможна, и разумна оптимизация системы управления данными, позволяющая снизить расходы на управление транзакциями, если приложениям не требуется качественная согласованность данных.

По всей видимости, исследования и разработки массивно-параллельных транзакционных СУБД в ближайшие годы будут активно продолжаться, и нам предстоит еще увидеть и услышать много интересного.

Литература

[1] Michael Stonebraker, Ugur Cetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone. Proceedings of the 21st International Conference on Data Engineering, 2005, pp. 2-11, http://www.cs.brown.edu/~ugur/fits_all.pdf.

Перевод на русский язык: Майкл Стоунбрейкер, Угур Кетинтемел. "Один размер пригоден для всех": идея, время которой пришло и ушло, 2007, http://citforum.ru/database/articles/one_size_fits_all/.

[2] Jeffrey Dean, Sanjay Ghemawat. MapReduce: Simplifed Data Processing on Large Clusters, Proceedings of the Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004, pp. 137-150, http://labs.google.com/papers/mapreduce-osdi04.pdf.

[3] Michael Stonebraker, David J. DeWitt. MapReduce: A major step backwards, Database Column, January 17, 2008, http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards/.

[4] Michael Stonebraker, David J. DeWitt. MapReduce II, Database Column, January 25, 2008, http://databasecolumn.vertica.com/database-innovation/mapreduce-ii/.

[5] С.Д. Кузнецов. MapReduce: внутри, снаружи или сбоку от параллельных СУБД?, Труды Института системного программирования, т. 19, М., ИСП РАН, 2010, стр. 35-40, http://citforum.ru/database/articles/dw_appliance_and_mr/.

[6] Pat Helland, Dave Campbell. Building on Quicksand. Proceedings of the Fourth Biennial Conference on Innovative Data Systems Research (CIDR 2009), January 4-7, 2009, Asilomar, Pacific Grove, CA USA, http://www-db.cs.wisc.edu/cidr/cidr2009/Paper_133.pdf.
Перевод на русский язык: Пэт Хелланд, Дейв Кэмпбел. Дом на песке, 2010, http://citforum.ru/database/articles/quicksand/.

[7] Eric Brewer, Towards Robust Distributed Systems, Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 2000, p. 7, http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf.

[8] Daniel Abadi, Alexander Thomson. The problems with ACID, and how to fix them without NoSQL. DBMS Musings, August 31, 2010, http://dbmsmusings.blogspot.com/2010/08/problems-with-acid-and-how-to-fix-them.html.
Перевод на русский язык: Дэниел Абади и Александер Томсон. Проблемы с ACID, и как их устранить, не прибегая к использованию NoSQL, 2010, http://citforum.ru/gazeta/164/.

[9] Tim Kraska, Martin Hentschel, Gustavo Alonso, Donald Kossmann. Consistency Rationing in the Cloud: Pay only when it matters. Proceedings of the 35th VLDB Conference, August 24-28, 2009, Lyon, France, pp. 253-264, http://www.dbis.ethz.ch/research/publications/ConsistencyRationing.pdf.
Перевод на русский язык: Тим Краска, Мартин Хеншель, Густаво Алонсо, Дональд Коссман. Рационализация согласованности в "облаках": не платите за то, что вам не требуется, 2010, http://citforum.ru/database/articles/kossmann_vldb_2009/.

[10] Домашняя страница проекта H-store, 2010, http://hstore.cs.brown.edu/.

[11] Официальный сайт компании VoltDB, 2010, http://voltdb.com/.

[12] Michael Stonebraker. Errors in Database Systems, Eventual Consistency, and the CAP Theorem. BLOG@CACM, April 5, 2010, http://cacm.acm.org/blogs/blog-cacm/83396-errors-in-database-systems-eventual-consistency-and-the-cap-theorem/fulltext.
Перевод на русский язык: Майкл Стоунбрейкер. Ошибки в системах баз данных, согласованность "в конечном счете" и теорема CAP, 2010, http://citforum.ru/gazeta/154/.

[13] Michael Stonebraker. Clarifications on the CAP Theorem and Data-Related Errors. VoltDB.com, October 21, 2010, http://voltdb.com/blog/clarifications-cap-theorem-and-data-related-errors.

Перевод на русский язык: Майкл Стоунбрейкер. Уточнения по поводу теоремы CAP и ошибок, связанных с данными, 2010, http://citforum.ru/gazeta/169/.

[14] Carlo Curino, Evan Jones, Yang Zhang, Sam Madden. Schism: a Workload-Driven Approach to Database Replication and Partitioning. 36th International Conference on Very Large Data Bases, September 13-17, 2010, Singapore. Proceedings of the VLDB Endowment, Vol. 3, No. 1, 2010, pp. 48-57, http://db.csail.mit.edu/pubs/schism-vldb2010.pdf.
Перевод на русский язык: Карло Курино, Эван Джонс, Янг Жанг и Сэм Мэдден. Schism: управляемый рабочей нагрузкой подход к репликации и разделению баз данных, 2010, http://citforum.ru/database/articles/madden_vldb2010/.

[15] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, Anastasia Ailamaki. Data-Oriented Transaction Execution. 36th International Conference on Very Large Data Bases, September 13-17, 2010, Singapore. Proceedings of the VLDB Endowment, Vol. 3, No. 1, 2010, pp. 928-939, http://infoscience.epfl.ch/record/141326/files/pvldb10pandis.pdf.
Перевод на русский язык: Иппократис Пандис, Райан Джонсон, Никос Харадавеллас и Анастасия Айламаки. Выполнение транзакций, ориентированное на данные, 2010, http://citforum.ru/database/articles/ailamaki_vldb2010/.

[16] Theo Haerder, Andreas Reuter. Principles of transaction-oriented database recovery. ACM Computing Surveys, Volume 15, Issue 4, December 1983, pp. 287 – 317, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.88.5548&rep=rep1&type=pdf

[17] Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, Irving Traiger. The recovery manager of the System R database manager. ACM Computing Surveys, Volume 13, Issue 2, June 1981, 223-242, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.94.9008&rep=rep1&type=pdf

[18] Eric Brewer. A certain freedom: thoughts on the CAP theorem. Proceeding of the 29th ACM SIGACT-SIGOPS Symposium on Principles of distributed Computing, 2010, p. 335, http://portal.acm.org/citation.cfm?id=1835701.

[19] Seth Gilbert, Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News, Volume 33 Issue 2, June 2002, pp. 51-59, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.67.6951&rep=rep1&type=pdf

[20] Julian Browne. Brewer's CAP Theorem, January 11, 2009, http://www.julianbrowne.com/article/viewer/brewers-cap-theorem.

[21] Daniel Abadi. Problems with CAP, and Yahoo’s little known NoSQL system. DBMS Musings, April 23, 2010, http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html.

[22] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, Pat Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). Proceedings of the 33rd International Conference on Very Large Data Bases, 2007, pp. 1150-1160, http://cs-www.cs.yale.edu/homes/dna/papers/vldb07hstore.pdf.
Перевод на русский язык: Майкл Стоунбрейкер, Сэмюэль Мэдден, Дэниэль Абади, Ставрос Харизопулос, Набил Хачем, Пат Хеллэнд. Конец архитектурной эпохи, или Наступило время полностью переписывать системы управления данными, 2007, http://citforum.ru/database/articles/end_of_arch_era/.

[23] M. Stonebraker, C. Bear, U. Cetintemel, M. Cherniack, T. Ge, N. Hachem, S. Harizopoulos, J. Lifter, J. Rogers, and S. Zdonik. One Size Fits All?— Part 2: Benchmarking Results. Proceedings of the Third Biennial Conference on Innovative Data Systems Research (CIDR 2007), January 7-10, 2007, Asilomar, Pacific Grove, CA USA, http://nms.csail.mit.edu/~stavros/pubs/osfa.pdf.

Перевод на русский язык: Майкл Стоунбрейкер, Чак Беэ, Угур Кетинтемел, Мич Черняк, Тиньян Ге, Набил Хачем, Ставрос Харизопулос, Джон Лифтер, Дженни Роджерс, Стэн Здоник. Пригоден ли один размер для всех? Часть 2: результаты тестовых испытаний, 2007, http://citforum.ru/database/articles/one_size_fits_all_2/.

[24] Pat Helland. Life beyond Distributed Transactions: an Apostate's Opinion. Proceedings of the Third Biennial Conference on Innovative Data Systems Research (CIDR 2007), January 7-10, 2007, Asilomar, Pacific Grove, CA USA, http://web.mit.edu/tibbetts/Public/CIDR_2007_Proceedings/papers/cidr07p15.pdf.

[25] Using VoltDB, V1.2, VoltDB, Inc., June 13, 2010, http://community.voltdb.com/docs/UsingVoltDB/index.

[26] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, Daniel J. Abad. HStore: A HighPerformance, Distributed Main Memory Transaction Processing System. Proceedings of the VLDB Endowment, Volume 1 Issue 2, August 2008, pp. 1496-1499, http://cs-www.cs.yale.edu/homes/dna/papers/hstore-demo.pdf.

[27] Evan P.C. Jones, Daniel J. Abadi, Samuel Madden. Low Overhead Concurrency Control for Partitioned Main Memory Databases. SIGMOD’10, Indianapolis, Indiana, USA, June 6–11, 2010, http://cs-www.cs.yale.edu/homes/dna/papers/hstore-cc.pdf.
Перевод на русский язык: Эван Джонс, Дэниэль Абади и Сэмуэль Мэдден. Управление параллелизмом с низкими накладными расходами для разделенных баз данных в основной памяти, 2010, http://citforum.ru/database/articles/h-store-sigmod2010/.

[28] Daniel Abadi, Alexander Thomson. The Case for Determinism in Database Systems. 36th International Conference on Very Large Data Bases, September 13-17, 2010, Singapore. Proceedings of the VLDB Endowment, Vol. 3, No. 1, 2010, pp. 70-80, http://db.cs.yale.edu/determinism-vldb10.pdf.
Перевод на русский язык: Дэниел Абади и Александер Томсон. Доводы в пользу детерминизма в системах баз данных, 2010, http://citforum.ru/database/articles/abadi_vldb2010/.

[29] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, Michael Stonebraker. OLTP Through the Looking Glass, and What We Found There, Proceedings of the ACM SIGMOD International Conference on Management of Data, Vancouver, BC, Canada, June 2008, pp. 981-992, http://db.cs.yale.edu/hstore/oltpperf-sigmod08.pdf.
Перевод на русский язык: Ставрос Харизопулос, Дэниэль Абади, Сэмюэль Мэдден, Майкл Стоунбрейкер. OLTP в Зазеркалье, 2010, http://citforum.ru/database/articles/oltp_lg/.

[30] Домашняя страница проекта Shore, 2010, http://www.cs.wisc.edu/shore/.

[31] G. Karypis. METIS— Family of Multilevel Partitioning Algorithms, 2010, http://glaros.dtc.umn.edu/gkhome/views/metis.

[32] Домашняя страница проекта WEKA (Waikato Environment for Knowledge Analysis), 2010, http://www.cs.waikato.ac.nz/ml/index.html.

[33] Домашняя страница проекта Shore-NT, 2010, http://www.cs.wisc.edu/shore-mt/.

[34] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: A Scalable Storage Manager for the Multicore Era. Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology (EDBT 2009), 2009, pp. 24-35, http://diaswww.epfl.ch/shore-mt/papers/edbt09johnson.pdf.

[35] Spinlock. Материал из Википедии – свободной энциклопедии, 2010, http://ru.wikipedia.org/wiki/Spinlock.
[36] Сергей Кузнецов. Базы данных. Вводный курс. 13.3.1. Синхронизационные блокировки, 2008, http://citforum.ru/database/advanced_intro/41.shtml#13.3.1.
[37] TPC BENCHMARK C. Standard Specification. Revision 5.11. Transaction Processing Performance Council, 2010, http://www.tpc.org/tpcc/spec/tpcc_current.pdf.
[38] Daniela Florescu, Donald Kossmann. Rethinking Cost and Performance of Database Systems. SIGMOD Record, Vol. 38, No. 1, March 2009, pp. 43-48, http://www.dbis.ethz.ch/research/publications/sigrec08.pdf.
Перевод на русский язык: Даниела Флореску, Дональд Коссман. Переосмысление стоимости и производительности систем баз данных, 2009, http://citforum.ru/database/articles/rethinking/.
[39] С.Д. Кузнецов. Год эпохи перемен в технологии баз данных. Труды Института системного программирования, т. 19, М., ИСП РАН, 2010, стр. 9-34, http://citforum.ru/database/articles/epoch/.
[40] Werner Vogels. Data Access Patterns in the Amazon.com Technology Platform. Proceedings of the 33rd International Conference on Very Large Data Bases, Sep 2007, p. 1, http://www.vldb.org/archives/website/2007/program/videos/p1-vogels.wmv.
[41] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a Database on S3. Proceedings of the ACM SIGMOD International Conference on Management of Data, Vancouver, BC, Canada, June 2008, pp. 251–264, http://www.dbis.ethz.ch/research/publications/sigmod08-s3.pdf.
[42] A. Tanenbaum and M. van Steen. Distributed Systems: Principles and Paradigms. Prentice Hall, Upper Saddle River, NJ, 2002.
Перевод на русский язык: Э. Таненбаум, М. ван Стеен. Распределенные системы. Принципы и парадигмы. СПб.: Питер, 2003, 889 стр.
[43] Н.А. Олифер, В.Г. Олифер, П.Б. Храмцов, В.И. Артемьев, С.Д. Кузнецов. Стратегическое планирование сетей масштаба предприятия. Центр Информационных Технологий, 1997, http://citforum.ru/nets/spsmp/.
[44] Домашняя страница компании 28msec/проекта Sausalito, 2010, http://www.28msec.com/.
[45] Домашняя страница Amazon Simple Storage Service (Amazon S3), 2010, http://aws.amazon.com/s3/.
[46] Домашняя страница Amazon Elastic Compute Cloud (Amazon EC2), 2010, http://aws.amazon.com/ec2/.
[47] XQuery 1.0: An XML Query Language (Second Edition), W3C Recommendation, 14 December 2010, http://www.w3.org/TR/xquery/.
[48] Werner Vogels. Eventually Consistent. ACM Queue, Vol. 6 No. 6, October 2008, pp. 15-19, http://queue.acm.org/detail.cfm?id=1466448.
[49] Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, Tim Kraska. Building a Database in the Cloud. Technical Report, ETH Zurich, 2009, http://www.dbis.ethz.ch/research/publications/dbs3.pdf.