Особенности эффективной обработки SQL-запросов к базам данных консервативного типа

Р.К. Классен

Казанский национальный исследовательский технический университет им. А.Н. Туполева – КАИ, г. Казань, Россия

Аннотация. Рассматриваются вопросы обработки SQL-запросов с высоким удельным весом операций join к базам данных консервативного типа (с эпизодическим обновлением данных в специально выделяемое время) повышенных объемов на платформе GPU-кластера. Модифицируется архитектура ранее разработанной параллельной СУБД Clusterix-N (N – от New). Предлагаются методы организации динамической сегментации промежуточных/временных отношений на выделенном узле с GPU-ускорителями и полной загрузки процессорных ядер узлов уровней JOIN и IO. Проводится сравнение производительности этой СУБД с оригинальной СУБД PerformSys на ограниченном тесте TPC-H (без операций записи) с $V_{БД}$=60 GB и $V_{БД}$=120 GB.

Ключевые слова: базы данных консервативного типа, повышенные объемы данных, модификация архитектуры параллельной СУБД, динамическая сегментация промежуточных/временных отношений, применение графических ускорителей, полная загрузка процессорных ядер, сравнительные оценки производительности.

DOI 10.14357/207186321804011

Введение

Развитие информационных технологий требует обработки все большего объема информации. Создано множество инструментов и СУБД для работы с большими данными [1]. Примерами таких СУБД могут служить: MS SQL Server, Oracle Database, SciDB [2], VoltDB [3], PostgreSQL XL [4], Clusterix [5, 6] и др. В большинстве своем – это закрытые коммерческие продукты высокой стоимости. Открытые системы существенно уступают коммерческим по надежности и, в меньшей мере, по производительности. Коммерческие системы требуют серьезных вычислительных мощностей для обеспечения приемлемой производительности. Далеко не каждая организация может позволить себе их приобретение. Поэтому создание отечественной СУБД, способной работать на сравнительно недорогих вычислительных кластерах, эффективно использовать их ресурсы и обрабатывать большие массивы данных, является актуальной задачей.

В [7] предложена архитектура параллельной СУБД консервативного типа Clusterix-N с ориентиром на относительно недорогие гибридные технологии. Она работоспособна на небольших (единицы ГБ) объемах БД и не применима к обработке БД повышенных объемов (десятки и сотни ГБ), поскольку такая обработка требовала чрезмерных объемов оперативной памяти узлов JOIN. Поэтому уже на $V_{БД}$=60 ГБ возникали отказы в работе. Вот почему первоначальная архитектура Clusterix-N требует модификации специально для работы с БД повышенных объемов.

Ранее в Clusterix-N была исключена динамическая сегментация (хеширование) промежуточных/временных отношений. Такой отказ был свя-

зан с высокой трудоемкостью этих операций и чрезмерной загрузкой сети при их выполнении. Вместе с тем, работа с БД повышенных объемов требует распределения работ по ядрам и узлам, чему способствует такая сегментации.

Задачей работы является исследование возможного пути такого распределения, свободного от указанных недостатков.

Суть рассматриваемого подхода к решению этой задачи состоит в соответствующей организации пакетной передачи данных и использовании GPU-ускорителей. Предлагается выделить отдельный узел в кластере, который возьмет на себя функции динамической сегментации промежуточных/временных отношений и управления параллельной обработкой на узлах JOIN.

1. Модификация архитектуры СУБД Clusterix-N

В результате внесенных изменений Clusterix-N теперь состоит из 5 модулей: MGM, IO, JOIN, HASH, SORT. Каждый модуль может быть размещен на выделенных узлах или делить один узел с другими модулями. Все коммуникации между модулями осуществляются по сети. Новая архитектура Clusterix-N представлена на Рис. 1. Главным модулем в системе является MGM. Он размещается на управляющем узле (Mgm) и содержит 14 транслированных к регулярному плану запросов без операций записи из состава теста TPC-H [8]. Трансляция в разработанном прототипе осуществляется вручную. Полученные в результате трансляции загрузочные модули «*select–project*»- и «*join*»-процедур поступают соответственно в узлы IO и JOIN, где передаются в MySQL, который их выполняет. MGM включает процессоры: УПР (процессор управления обработкой запросов, который представлен на Рис. 1 в виде блоков очереди запросов и конвейера обработки запросов), ROUTER (процессор балансировки нагрузки, представленный на Рис. 1 как связка конвейера обработки запросов с модулем сетевого взаимодействия и подсистемой мониторинга), BUF (процессор буфера промежуточных отношений). Процессор УПР выполняет управление обработкой запроса. Он получает транслированный запрос, помещает его в очередь и устанавливает статус в ожидание. Как только УПР обнаруживает, что все узлы IO способны принять очередной подзапрос *select*, он отправляет его в эти узлы. Операция отправки запроса повторяется до тех пор, пока не будут отосланы все подзапросы на выборку для каждого из отношений. Результат работы узлов IO поступает в BUF MGM в виде блоков данных. Блоки данных представляют собой набор по N кортежей результата в каждом блоке. Они накапливаются в буфере MGM по отношениям. Когда все блоки одного из отношений получены, они отправляется в узел HASH, где подвергаются хешированию и перераспределению. Хешированные данные отправляются в узлы JOIN с индексацией по ядрам. После окончания передачи узлы JOIN загружают данные в MySQL и запускают подзапросы *join* в количестве свободных процессорных ядер.

Результат подзапроса передается в узел HASH, где хешируется и передается в узлы JOIN для следующего подзапроса *join*. Операция повторяется до завершения всех подзапросов *join*. Как только все они завершены, итоговый результат передается в BUF MGM, а оттуда отправляется для дальнейшей обработки в SORT. Выбор узла или группы узлов для отправки очередного подзапроса осуществляет процессор ROUTER. Он анализирует объем свободной ОП на свободных узлах и определяет наиболее подходящий. На занятые узлы отправка не осуществляется. Основное отличие нового модуля MGM от представленного в [7] кроется в переходе на стратегию «группа узлов на запрос». Так, все узлы IO одновременно работают над одним подзапросом. В то же время узлы JOIN (и, соответственно, HASH) могут работать над другим.

2. Алгоритмы, разработанные для модулей IO и HASH

Как и прежде, каждый узел модуля IO оснащен СУБД MySQL. БД равномерно распределяется между этими узлами. Но, в отличие от предложенного в [7] алгоритма работы, здесь обработка подзапроса *select* оптимизируется для многоядерных узлов, реализуя метод параллельной обработки селективных запросов по алгоритму параллельной обработки селективных запросов.

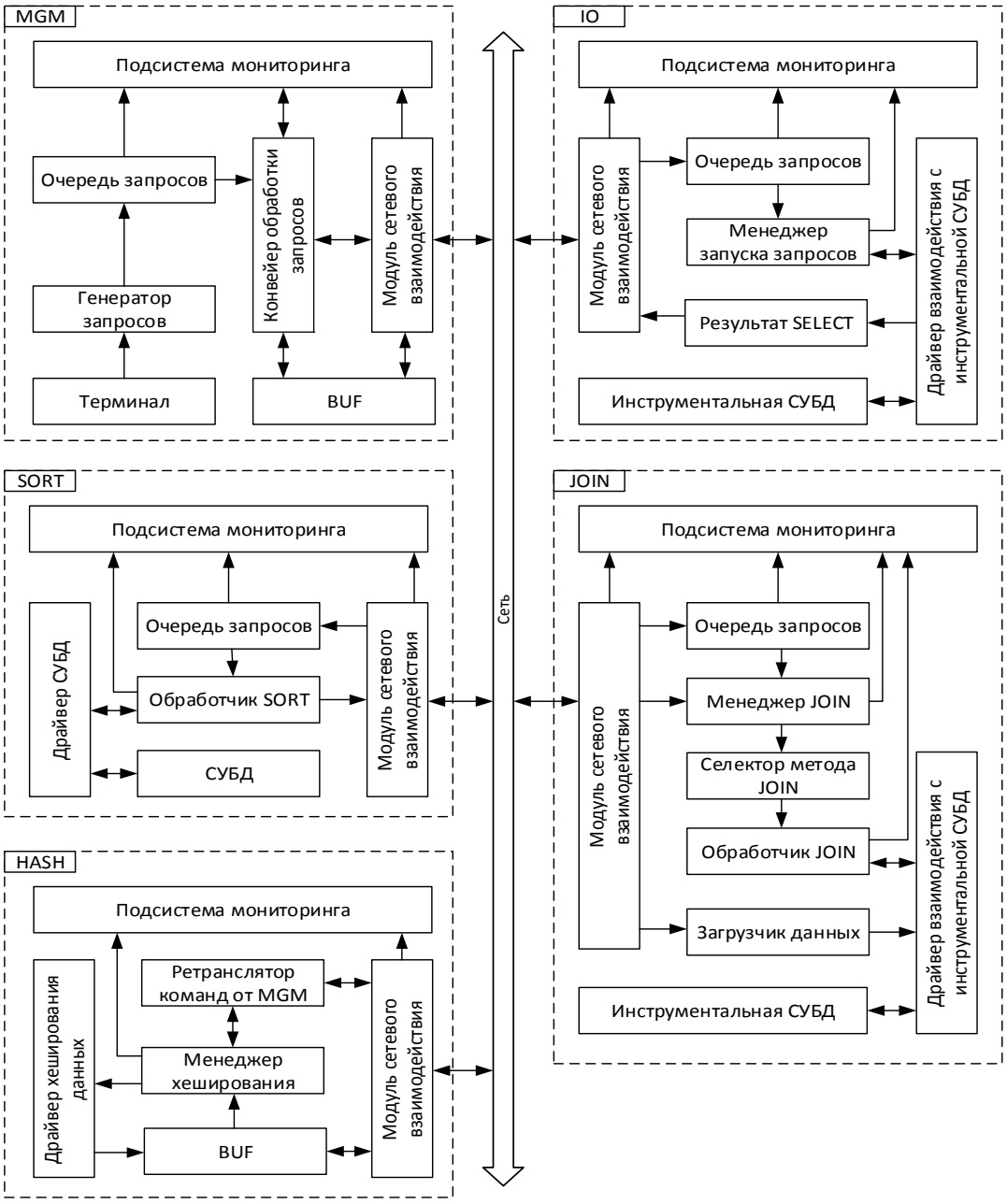

Рис. 1. Новая архитектура Clusterix-N

Алгоритм параллельной обработки селективных запросов. Обозначим: N – количество строк в блоке данных; P – номер строки результата, начиная с которой будет формироваться новый блок данных; B – количество выполненных модифицированных запросов; n – количество процессорных ядер в одном узле; Q – оригинальный подзапрос *select*; Q' – модифицированный подзапрос *select*; D – готовый блок данных; $R \in \overline{\{0, N \cdot n\}}$ – общее количество строк, полученных в результате выполнения всех запросов на всех ядрах. Под блоком данных будем понимать байтовый массив, в который записано множество строк результата выполненного запроса. Разработанный алгоритм имеет вид:

1. *N:*= значение, установленное в настройках.

2. Подготовить множество модифицированных запросов в количестве процессорных ядер:

$Q'_i := Q + «LIMIT\ P, N»$, $P := N \cdot (B + i)$,

где i – номер ядра $i = \overline{0, n-1}$.

3. Запустить Q' в работу по схеме «запрос на ядро»: на каждом ядре запустить соответствующий Q'_i.

4. Результат обработки Q'_i разместить в блоках данных D_i для всех i.

5. Подсчитать суммарное количество полученных строк и записать в R.

6. $B := B + n$.

7. Передать полученные блоки D_i в модуль сетевого взаимодействия.

8. Если $R = N \cdot n$, то перейти к шагу 2, иначе завершение работы, т.к. получены все результаты.

Такая реализация параллельной обработки позволяет полностью загрузить процессорные ядра многоядерного узла при достаточном объеме оперативной памяти (вся БД должна умещаться в оперативной памяти). Но эффективность работы оставляет желать лучшего. Так, при запуске селективного запроса на 12 ядрах он будет выполнен всего в 5 раз быстрее выполнения того же запроса на одном ядре. Это объясняется тем, что MySQL при выборке данных с условием по неключевому полю выполняет полное сканирование отношений и не может заранее определить смещение в файле данных для LIMIT *P,N* (ключевое слово LIMIT используется для ограничения количества строк, возвращаемых в результате запроса, где *P* смещение, а *N* – количество строк). В результате полное сканирование происходит параллельно на 12 ядрах, а ускорение получается за счет параллельного сбора результатов и отсутствия дополнительной работы по разбиению результата на блоки.

Полученные блоки данных асинхронно передаются по сети в BUF MGM. Размер блоков с большим N ($N = 10^6$ и более) исчисляется десятками и сотнями мегабайт. Большой размер блока позволяет использовать потоковую передачу, что положительно сказывается на использовании сети. Асинхронная природа модуля сетевого взаимодействия позволяет готовить новый блок для передачи по время отправки предыдущего. Такая модель позволяет передавать блоки друг за другом с минимальным временем простоя сети и с эффективным использованием вычислительных ресурсов узла IO.

Модуль HASH является прокси-сервером между модулем MGM и модулем JOIN. Он получает задания от процессора УПР и данные из BUF MGM и выполняет процедуру динамической сегментации с хешированием данных на GPU [9]. Результат хеширования помещается в буфер отправки по ядрам (для каждого ядра в узлах JOIN модуль HASH формирует буфер в своей памяти). Отправка данных происходит по готовности операции хеширования. Полученные из BUF MGM данные хешируются и передаются «на лету», т.е. без записи результата в локальный буфер.

Модуль HASH организует динамическую сегментацию промежуточных/временных отношений на выделенном узле с GPU-ускорителями путем хеширования на GPU и распределения данных по всем процессорным ядрам уровня JOIN. Хеширование выполняется с использованием алгоритма деления [10]. Значение хеш-функции – остаток от деления суммы ключевых полей строки на количество процессорных ядер JOIN.

Алгоритм хеширования на GPU. Обозначим $H = \{H_i\}$, $i \in \{\overline{0, N-1}\}$ – множество результатов хеширования всех i-строк в блоке данных; $Ss = \{Ss_i\}$, $i \in \{\overline{0, N-1}\}$ – множество начальных позиций всех i-строк в блоке данных; $Se = \{Se_i\}$, $i \in \{\overline{0, N-1}\}$ – множество конечных позиций всех i-строк в блоке; m – количество доступных ядер JOIN; $Hbuf = \{Hbuf_k\}$, $k \in \{\overline{0, m-1}\}$ – множество буферов результата динамической сегментации промежуточных/временных отношений; $C = \{C_k\}$, $k \in \{\overline{0, m-1}\}$ – множество объемов буферов динамической сегментации; $L = \{L_k\}$, $k \in \{\overline{0, m-1}\}$ – множество указателей конечных позиций $Hbuf$.

1. Скопировать полученный блок данных D в память GPU.

2. Выполнить разметку строк в блоке данных на GPU:

2.1. найти начальные позиции для всех строк в блоке данных и записать их в *Ss*;

2.2. найти конечные позиции для всех строк в блоке данных и записать из в *Se*;

2.3. количество строк в блоке записать в *N*.

3. Вычислить хеш-функцию для каждой строки блока данных *D* на GPU. Результаты вычисления хеш-функций занести в *H*.

4. Вычислить объем буфера для каждого ядра JOIN: для каждого $k = H_i$ вычислить сумму разностей $Se_i - Ss_i$ и записать ее в C_k, где $k = \overline{0, m-1}$ – номер ядра, $i = \overline{0, N-1}$ – номер строки в исходном блоке данных.

5. Выделить память для каждого $Hbuf_k$ размером C_k, где $k = \overline{0, m-1}$ – номер ядра.

6. Произвести копирование строк из *D* в *Hbuf* следующим образом: для каждого H_i, где $i = \overline{0, N-1}$ выполнить:

6.1. string := строка из *D* с позиции Ss_i до позиции Se_i;

6.2. $Hbuf_{H_i} := Hbuf_{H_i} + string$;

6.3. $L_{H_i} := L_{H_i} + Ss_i - Se_i$.

7. Удалить исходный блок данных.

Содержимое буферов *Hbuf* преобразуются в блоки данных и передаются в узлы JOIN. В процессе передачи хешированных блоков данных по ядрам адрес назначения (узел и ядро) определяется однозначно. Каждый узел JOIN содержит одинаковое количество процессорных ядер и уникальный идентификатор. Узел HASH сортирует узлы JOIN по уникальному идентификатору и проводит сквозную нумерацию ядер. Тем самым каждое ядро получает уникальный номер, который соответствует одному из буферов.

К недостаткам алгоритма можно отнести зависимость размера данных от объема памяти GPU и требование двойного объема оперативной памяти по сравнению с объемом блока данных. Так, чтобы обработать блок данных на GPU из *N*=1000000 строк со средней длиной строки *Len* = 1 KB и двумя ключевыми полями, понадобится *N*·*Len*~1000MB для исходного блока данных плюс объем массивов *H*, *Se* и *Ss*: *N*·*4*·*3*~12 MB, где 4 – размер одного элемента массива типа *int* в байтах, 3 – количество массивов. Итого на GPU потребуется ~1012MB для обработки. В то же время host система потребляет 2 объема блока данных (~2012MB в тех же условиях, что и GPU), т.к. выполняет копирование *D* в *Hbuf* и удаляет *D* только после завершения копирования.

3. Новая реализация модулей JOIN и SORT

Модуль JOIN, как и IO, оснащен СУБД MySQL. Он тоже претерпел ряд изменений. Теперь в зависимости от конфигурации системы модуль принимает задание и данные не только от MGM, но и от модуля HASH, производит загрузку данных в MySQL и запускает операцию *join*. Загрузка данных контролируется специальным загрузчиком и производится командой LOAD FILE после полного получения всех данных для загружаемого отношения. Но в связи с реализацией распределённой обработки теперь для каждого ядра предусматривается свой набор отношений, что допускает параллельную загрузку данных и параллельное выполнение операции *join*. Введена очередь запросов, поскольку данные для их обработки могут быть получены заранее.

Администратор выполняет настройку перед началом работы системы. Запросы в очереди могут выполняться одним из методов: параллельный, интегрированный или последовательный *join*. Параллельный – запускается на всех свободных ядрах вычислительного узла и эффективно работает с большим массивом данных. Последовательный – выполняет *join* один за другим на одном ядре и позволяет обрабатывать сразу несколько запросов в небольшой БД. Интегрированный – выполняет сразу все необходимые операция *join* и требует значительного объема памяти в узле. Запуском запросов из очереди и контролем их выполнения занимается менеджер JOIN, который загружает метод *join* и передает ему управление. Результат *join*-обработки передается в модуль HASH сразу после его выполнения. Все промежуточные отношения удаляются по завершении обработки

подзапроса. Поскольку наше рассмотрение ориентировано на БД повышенных объемов, в статье рассматривается только метод параллельного *join*.

Как и ранее [7], модуль SORT использует свой MySQL. Аналогично модулю JOIN, он получает задание и данные от УПР MGM и BUF MGM, производит загрузку данных в MySQL, запускает операцию *sort*. Результат работы модуля SORT размещается в новом отношении, которое передается по готовности в BUF MGM и от него – пользователю. Работа модуля организуется по стратегии «запрос на ядро».

4. Сравнительные оценки быстродействия

В попытке достижения полной загрузки процессорных ядер ранее была реализована система PerfomSys [11]. Основная идея этой реализации заключалась в применении стратегии «запрос на ядро» при соответствующей настройке MySQL. Такая стратегия позволила полностью загрузить все процессорные ядра вычислительного кластера при условии достаточного количества запросов в системе и получить существенное повышение быстродействия в сравнении с Clusterix при объемах БД в несколько GB [12]. Проверку успешности модификации архитектуры Clusterix-N в сравнении с PerfomSys выполним с помощью натурного эксперимента при объемах БД в десятки GB.

Эксперимент организован на платформе GPU-кластера КНИТУ-КАИ. Параметры узлов: 2 six-core E5-2640CPU/2,5GHz/DDR3 128GB; 2 448-core GPU Tesla C-2075/1,15GHz/GDDR5 6GB (на Mgm GPU отсутствуют). Дисковая подсистема узла – RAID 10 из 4 WD1000 DHTZ/1TB. Операционная система – Windows Server 2012 R2. Интерконнект между узлами – GigabitEthernet с 24-портовым коммутатором SSE G24-TG4.

Представительский тест (ПТ) был организован из 6 перестановок TPC-H Throughput Test без операций записи. На множестве этих перестановок сформирован поток из 84 запросов. Тестовые данные были сгенерированы утилитой dbgen из состава TPC-H. Размер сгенерированных БД составил 60 GB и 120 GB. После загрузки БД в MySQL и индексации их размер составил ~100 GB и ~200 GB соответственно.

Выполнение потока запросов на PerformSys. В случае PerfomSys узлы кластера распределены следующим образом (Рис. 2): 6 узлов исполнительных и 1 узел управляющий. Управляющий узел выполняет функции роутеризации, ведения очереди запросов, балансировку нагрузки. Модуль симуляции клиентов запускается на этом же узле. Каждый исполнительный узел включает в себя собственно модуль сервера PerformSys и СУБД MySQL с полной копией БД на каждом узле.

PerformSys использует стратегию «ядро на запрос». Поэтому клиентский модуль эмулирует работу 72 клиентов, т.к. в системе могут параллельно обрабатываться 72 запроса (6 узлов по12 ядер в узле дают 72 ядра). Как только очередной запрос выполнен, на его место отправляется новый, вплоть до исчерпания очереди на управляющем узле.

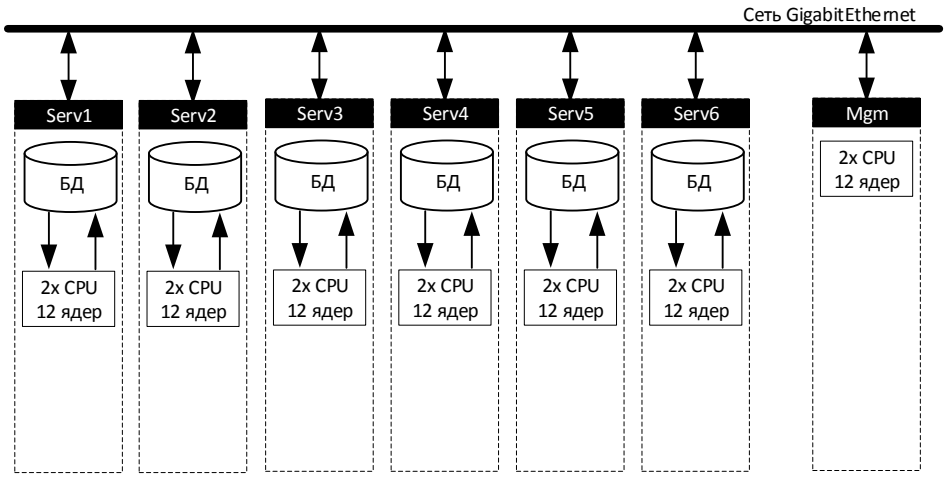

Рис. 2. Конфигурация экспериментального полигона для PerformSys

СУБД MySQL сконфигурирована особым образом:

– отключено кеширование результатов запросов;

– используется хранилище БД InnoDB;

– под буфер данных выделено 80% оперативной памяти;

– после старта СУБД вся БД загружается в оперативную память хоста.

Результаты выполнения потока запросов для PerformSys представлены в Табл. 1.

В случае $V_{БД}$ = 60 GB вся БД размещается в оперативной памяти узла. Такое размещение позволяет избавиться от необходимости чтения данных с диска и существенно повышает производительность системы. Процессорные ядра нагружены полностью. Время обработки потока запросов составило ~1 час, а среднее время ожидания результата клиентами составило 11 минут.

В случае $V_{БД}$ = 120 GB вся БД не может разместиться в оперативной памяти узла. Поэтому производится активная работа с диском. Поскольку диск является узким местом в работе системы, процессорные ядра не нагружаются полностью, и бóльшая часть времени уходит именно на чтение данных с диска. В итоге имеем время обработки потока запросов ~50 часов и существенно увеличившееся среднее время ожидания ответа (700 минут, против 11).

В процессе проведения эксперимента с $V_{БД}$ = 120 GB выяснилось, что MySQL не может принимать непрерывно поступающие запросы (они отбрасываются через 15 секунд после их передачи в MySQL и не исполняются) во время

Табл. 1. Результаты обработки потока запросов PerformSys

Параметры эксперимента	Время работы, час	Среднее время ожидания ответа, мин
$V_{БД}$ = 60 GB	0.93	11.05
$V_{БД}$ = 120 GB	50.25	699.36

работы с диском. Вероятно, причина кроется в хранилище InnoDB, которое для каждого нового запроса создает временный файл на диске. В этом случае имеется как минимум 2 решения: ввести ожидание готовности MySQL и переместить директорию временных файлов на более быстрый носитель (например, использовать SSD или RAM-диск). Но поскольку вся оперативная память используется для выполнения запросов, то остается один вариант: ввести ожидание готовности. Именно этот вариант использован в эксперименте (Табл. 1).

Готовность MySQL определяется следующим образом. В СУБД передается очередной запрос. Если при его выполнении возникла ошибка недоступности MySQL, то он перезапускается заново через 15 секунд. Если запрос не был выполнен за 12 часов и возникла очередная ошибка его выполнения, то он отбрасывается.

Выполнение потока запросов на Clusterix-N. Конфигурация экспериментального полигона показана на Рис. 3. Она включает 2 узла IO, 3 узла JOIN, 1 узел HASH и 1 узел MGM. Такая ассиметричная конфигурация с превалированием числа узлов JOIN выбрана из-за большого объема работ на этом уровне. Двух узлов IO в данном

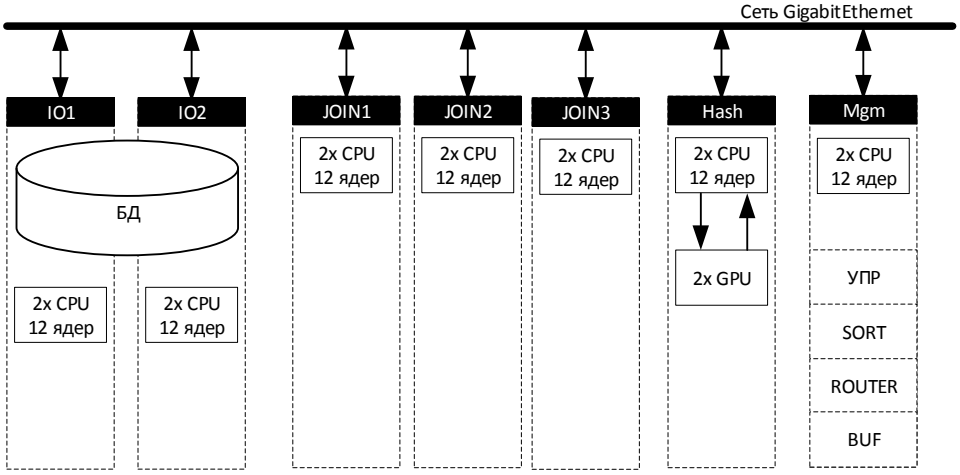

Рис. 3. Конфигурация экспериментального полигона для Clusterix-N

случае вполне достаточно для его постоянной загрузки. БД распределена между узлами IO.

СУБД MySQL на узлах IO сконфигурирована аналогично PerformSys. После запуска данные загружаются в оперативную память узлов. На уровне JOIN для исключения использования диска используется хранилище MEMORY, увеличены объемы буферов ключей и сортировки. На уровне SORT в целях экономии оперативной памяти используется хранилище MyISAM в конфигурации по умолчанию.

Поскольку Clusterix-N использует множество узлов для работы на каждом уровне, а уровней обработки 3, то для обеспечения эффективности системы требуется, чтобы в ней велась одновременная обработка по крайней мере 3-х запросов. Как было показано в [13], объем данных, необходимый для обработки всех 14 запросов, составляет 1,4 $V_{БД}$, а объем данных для самого «большого» запроса составляет $0.43V_{БД}$. С учетом объема доступной оперативной памяти узла MGM (128 GB) и необходимости выделения части памяти для модуля SORT, длина очереди для Clusterix-N на GPU-кластере КНИТУ-КАИ была принята равной 7.

Результаты в Табл. 2 показывают превосходство Clusterix-N над PerformSys в случае $V_{БД}$ = 120 GB и полную несостоятельность в случае $V_{БД}$ = 60 GB. При этом в случае $V_{БД}$ = 60 GB среднее время ожидания существенно выше, чем у PerformSys (45 мин против 11), но при увеличении объема БД до 120 GB картина меняется противоположным образом: среднее время ожидания для Clusterix-N – 83 минуты, а для PerformSys – 700 минут.

На Рис. 4 представлены временные диаграммы выполнения потока запросов на Clusterix-N для $V_{БД}$ = 120 GB. Как следует из рисунка, двух узлов IO вполне достаточно для обеспечения полной загрузки уровня JOIN. Длина очереди, равная 7, оказывается достаточной, т.к. на визуализации видны небольшие периоды ожидания, связанные с заполнением памяти в узле MGM. В тоже время модуль SORT, расположенный на узле MGM, практически не загружен. Объяснить это можно существенным уменьшением объема данных после выполнения всех операций JOIN. Сеть особенно активно используется узлами MGM и HASH.

Табл. 2. Результаты обработки потока запросов Clusterix-N

Параметры эксперимента	Время работы, час	Среднее время ожидания ответа, мин
$V_{БД}$ = 60 GB	9.2	45.24
$V_{БД}$ = 120 GB	19.67	83.18

Заключение

Высказанные в [13] сомнения в эффективности динамической сегментации промежуточных/временных отношений при повышенных объемах БД были обусловлены тем, что при использовании регулярного плана обработки запросов и сохранении стратегии, принятой в Clusterix, даже при наличии GPU-акселераторов в узлах она сохраняет перегрузки по интерконнекту и, как итог, приводит к непредсказуемым последствиям [14]. Проведенное рассмотрение показывает, что выделение отдельного узла с GPU-ускорителем под динамическую сегментацию и применение блочной организации сетевых передач позволяет решить эту проблему при модификации архитектуры Clusterix-N. Тем не менее, для консервативных БД малых объемов нет необходимости распределять БД по узлам, выполнять динамическую сегментацию и применять GPU-ускорители. В этом случае преимущество остается за PerformSys.

Исходные коды систем PerformSys и модифицированной версии Clusterix-N помещены в открытый доступ [15, 16].

В результате выполненной модификации архитектуры Clusterix-N:

1. предложен метод организации динамической сегментации промежуточных/временных отношений на выделенном узле с GPU-ускорителями, основанный на хешировании с ускорением на GPU и распределении данных по всем процессорным ядрам всех узлов JOIN, что, в отличие от реализации этой процедуры в СУБД Clusterix, позволяет существенно ускорить операции хеширования, загрузить все процессорные ядра всех узлов уровня JOIN и более эффективно использовать сеть;

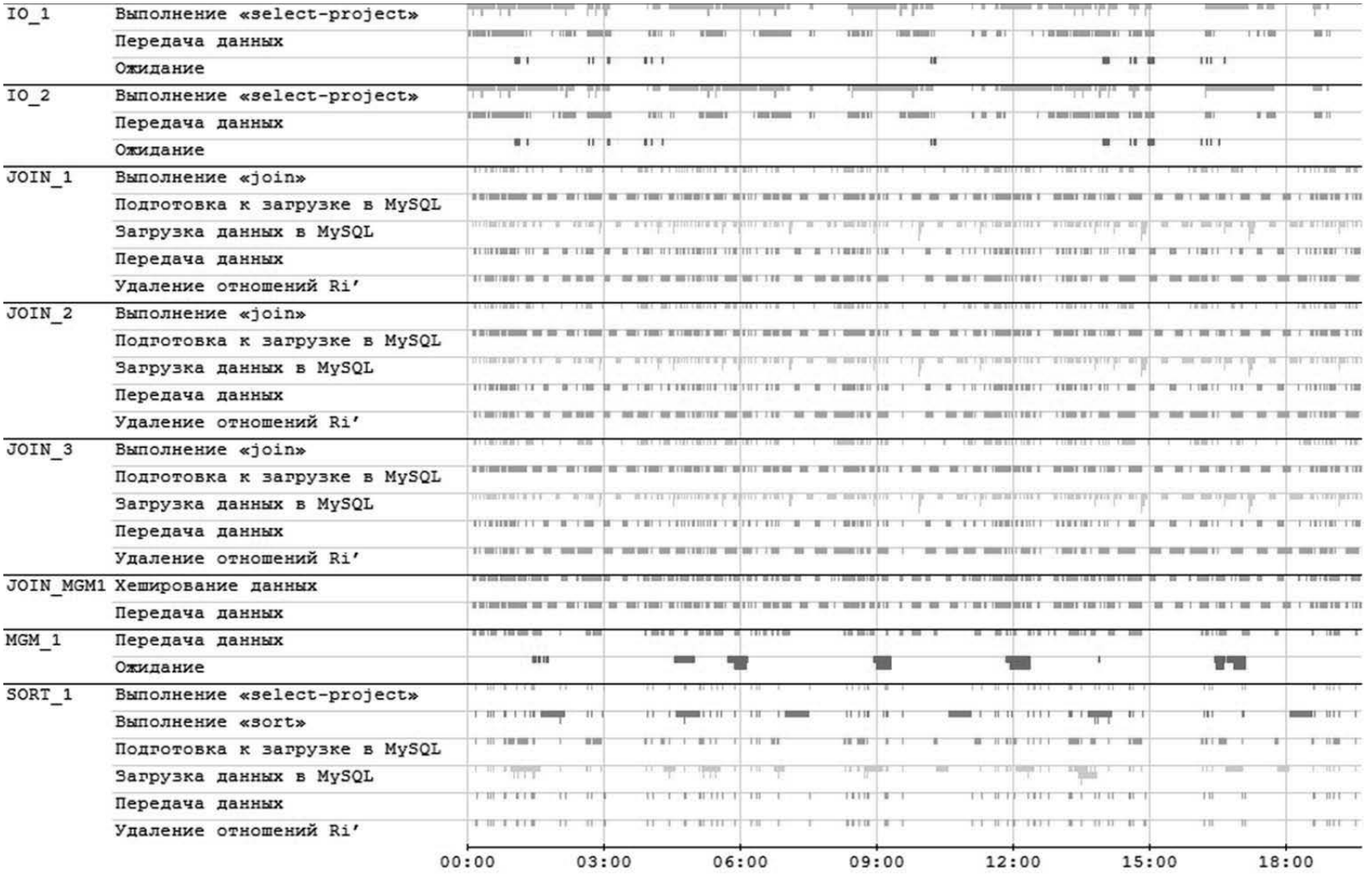

Рис. 4. Визуализация выполнение потока запросов на Clusterix-N для $V_{БД}$ = 120 GB

2. предложен метод параллельной обработки селективных запросов на уровне IO, основанный на поблочной выборке из СУБД MySQL, что, в отличие от ранее реализованного метода в Clusterix-N и PerformSys, позволяет использовать все узлы IO для обработки одного селективного запроса с полной загрузкой процессорных ядер.

Эффективность предложенных методов показана экспериментально путем сравнения двух различных СУБД (PerformSys и Clusterix-N) на двух объемах данных ($V_{БД}$ = 60 GB и $V_{БД}$ = 120 GB). При $V_{БД}$ = 60 GB вся БД помещается в ОП узлов, что исключает использование дисковой подсистемы и позволяет системе PerformSys работать с максимальной скоростью. Однако при $V_{БД}$ = 120 GB эта система начинает активно использовать диск, в то время как в Clusterix-N вся БД по-прежнему умещается в ОП. В итоге для Clusterix-N при такой $V_{БД}$ имеем время обработки ПТ 19.7 часа против 50 часов для PerformSys, а среднее время ожидания ответа составило 83 минуты против 700.

В процессе экспериментального исследования было установлено, что GPU производит операции хеширования вдвое быстрее скорости передачи данных по сети, а узлы JOIN по большей части ожидают сетевых передач. Задействовать оба GPU в узле хеширования и ускорить доставку временных отношений в узлы JOIN можно заменой сети на 10GigabitEthernet. В результате такой замены среднее время ожидания ответа и общее время обработки ПТ должны уменьшиться в разы.

Литература

1. Matt Turck Firing on All Cylinders: The 2017 Big Data Landscape [Web page] // URL: http://mattturck.com/bigdata2017/
2. Paradigm4: Creators of SciDB a computational DBMS URL: https://www.paradigm4.com/
3. In-Memory Database | VoltDB URL: https://www.voltdb.com/
4. Postgres-XL | Open Source Scalable SQL Database Cluster URL: https://www.postgres-xl.org/
5. Абрамов, Е.В. Параллельная СУБД Clusterix. Разработка прототипа и его натурное исследование /Е.В. Абрамов //Вестник КГТУ им. А.Н. Туполева. – 2006. – №2. – С.50-55.
6. Райхлин, В.А. Информационные кластеры как диссипативные системы //В.А. Райхлин, Д.О. Шагеев //Нелинейный мир. – 2009. – Т.7, №5.– С.323-334.
7. В.А. Райхлин, Р.К. Классен Сравнительно недорогие гибридные технологии консервативных СУБД больших объемов // Информационные технологии и вычислительные системы. Москва. – 2018. – Т. 68. №1. – С. 46-59.
8. TPC Benchmark™ H Standard Specification Revision 2.17.1, URL: http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.1.pdf
9. Реализация хеширования на GPU. URL: https://github.com/rozh1/gpuhash
10. Martin, J. Computer database organization. Second Edition — Prentice-Hall, Inc., Eng-lewood Cliffs, New Jersey 07632, 1977.
11. Классен Р.К. Программа региональной балансировки нагрузки к базе данных консервативного типа на кластерной платформе «PerformSys». Свидетельство о государственной регистрации программы для ЭВМ №2017611785 от 09.02.2017.
12. Классен Р.К. Повышение эффективности параллельной СУБД консервативного типа на кластерной платформе с многоядерными узлами // Вестник КГТУ им. А.Н.Туполева, № 1, 2015. С. 112-118.
13. Vadim A. Raikhlin, Roman K. Klassen Can GPU-accelerator significantly increase the effectiveness of conservative DBMS considerable volumes on cluster platforms? //2017 International Siberian Conference on Control and Communications (SIBCON). 2017. P. 1-5. DOI: 10.1109/SIBCON.2017.7998474
14. В.А. Райхлин, Р.Ш. Минязев Анализ процессов в кластерах консервативных баз данных с позиций самоорганизации // Вестник КГТУ им. А.Н.Туполева, № 2, 2015. С. 120-126.
15. PerformSys. URL: https://github.com/rozh1/PerformSys
16. Clusterix-N. URL: https://bitbucket.org/rozh/clusterixn

Классен Роман Константинович. Казанский национальный исследовательский технический университет им. А.Н. Туполева, г. Казань. Аспирант. Количество печатных работ: 9. Область научных интересов: высокопроизводительные системы, big data, информационные технологии. E-mail: klassen.rk@gmail.com

Features of efficient processing of SQL-queries to conservative type databases

R.K. Klassen

Kazan National Research Technical University named after A. N. Tupolev - KAI, Kazan. Russia

In article, we are considering the problems of processing SQL queries with a high specific weight of *join* operations to conservative type databases (with occasional updating of data in specially allocated time) with increased data volumes on the GPU-cluster platform. Modified architecture of the previously developed parallel DBMS Clusterix-N (N - from New). We are proposed the methods for organizing dynamic segmentation of intermediate / temp relationships on a dedicated node with GPU accelerators and full load of processor cores of nodes on JOIN and IO levels. The performance of this DBMS is compared with the original DBMS PerformSys on a limited TPC-H test (without write operations) with V_{DB}=60 GB and V_{DB}=120 GB.

Keywords: conservative type databases, increased volumes of data, modification of parallel DBMS architecture, dynamic segmentation of intermediate / temp relations, application of graphic accelerators, full load of processor cores, comparative performance estimates.

DOI 10.14357/207186321804011

References

1. Matt Turck Firing on All Cylinders: The 2017 Big Data Landscape [Web page] // URL: http://mattturck.com/bigdata2017/
2. Paradigm4: Creators of SciDB a computational DBMS URL: https://www.paradigm4.com/
3. In-Memory Database | VoltDB URL: https://www.voltdb.com/
4. Postgres-XL | Open Source Scalable SQL Database Cluster URL: https://www.postgres-xl.org/
5. Abramov E.V. Parallel'naya SUBD Clusterix. Razrabotka prototipa i ego naturnoe issledovanie [Parallel DBMS Clusterix. Development of prototype and its full-scale studits] // Herald of KSTU named after A.N. Tupolev. Kazan. 2006. № 2. P.50-55.
6. Raikhlin, V.A. Informacionnye klastery kak dissipativnye sistemy // V.A. Raikhlin, D.O. Shageev // Nonlinear world. Vol.7. 2009. №5. P.323-334.
7. V.A. Rajhlin, R.K. Klassen Sravnitel'no nedorogie gibridnye tekhnologii konservativnyh SUBD bol'shih ob"emov [Relatively inexpensive hybrid technology of large volumes conservative DBMS] // Journal of Information Technologies and Computing Systems. Moscow. 2018. P. 46-59.
8. TPC Benchmark^{TM} H Standard Specification Revision 2.17.1, URL: http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.1.pdf
9. Implementation of HASH on GPU. URL: https://github.com/rozh1/gpuhash
10. Martin, J. Computer database organization. Second Edition — Prentice-Hall, Inc., Eng-lewood Cliffs, New Jersey 07632, 1977.
11. Klassen R.K. Programma regional'noj balansirovki nagruzki k baze dannyx konser-vativnogo tipa na klasternoj platforme «PerformSys». [The program for regional load balancing to a conservative type database on the cluster platform «PerformSys».] Svidetel'stvo o gosudarstvennoj registracii programmy dlya E'VM №2017611785 ot 09.02.2017. [Certificate of state registration of the computer program No. 2017611785 of 09.02.2017].
12. Klassen R.K. Povyshenie e'ffektivnosti parallel'noj SUBD konservativnogo tipa na klaster-noj platforme s mnogoyadernymi uzlami [Improving efficiency of parallel conservative type DBMS on a cluster platform with multi-core nodes] //Herald of KSTU named after A.N. Tupolev. 2015. No.1. P.112-118.
13. Vadim A. Raikhlin, Roman K. Klassen Can GPU-accelerator significantly increase the effectiveness of conservative DBMS considerable volumes on cluster platforms? //2017 International Siberian Conference on Control and Communications (SIBCON). 2017. P. 1-5. DOI: 10.1109/SIBCON.2017.7998474.
14. Raikhlin V.A., Minyazev R.Sh. Analiz processov v klasterax konservativnyx baz dannyx s pozicij samoorganizacii [Analysis of processes in clusters of conservative databases from self-organization perspective] //Herald of KSTU named after A.N. Tupolev. 2015. No.2. P.120-126.
15. PerformSys. URL: https://github.com/rozh1/PerformSys
16. Clusterix-N. URL: https://bitbucket.org/rozh/clusterixn

Klassen Roman Konstantinovich. Department for Computer Systems, Institute for Computer Technologies and Information Protection, Kazan National Research Technical University named after A. N. Tupolev - KAI, Kazan. Russia. E-mail: klassen.rk@gmail.com