Федеральное государственное бюджетное образовательное учреждение
высшего образования «Казанский национальный исследовательский
технический университет им. А.Н. Туполева – КАИ» (КНИТУ – КАИ)

На правах рукописи

Классен Роман Константинович

КОНСЕРВАТИВНЫЕ СУБД КЛАССА BIGDATA С РЕГУЛЯРНЫМ ПЛАНОМ ОБРАБОТКИ ЗАПРОСОВ НА КЛАСТЕРНОЙ ПЛАТФОРМЕ

Специальность 05.13.11 –
«математическое и программное обеспечение вычислительных машин,
комплексов и компьютерных сетей»

Диссертация на соискание ученой степени
кандидата технических наук

Научный руководитель:
Райхлин Вадим Абрамович,
доктор физ.-мат. наук, профессор

Казань – 2019

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Актуальность исследований. Объемы собираемых и хранимых данных в современных информационных системах достигают огромных объемов, исчисляемых десятками, сотнями GB и TB, а порой и PB. Объемы БД в десятки и сотни GB все чаще встречаются у относительно небольших предприятий. Терабайтные и петабайтные объемы присущи территориально распределенным системам: социальные сети, поисковые системы, государственные информационные системы и др. Например, размер БД с сообщениями пользователей социальной сети «ВКонтакте» на 2017 год составлял ≈ 324 TB [1], а поисковые системы давно преодолели петабайтный порог (50 PB для «Яндекс» в 2016 году [2]). Кроме того, объемы хранимых данных по результатам разного рода испытаний и наблюдении также могут быть весьма значительными: от десятков GB до единиц PB [3].

Своевременная обработка накопленных данных требует применения высокопроизводительных вычислительных средств (HPC) и специализированного программного обеспечения – СУБД консервативного типа с эпизодическим обновлением данных. Имеется множество коммерческих и свободных разработок параллельных СУБД на платформе вычислительных кластеров (Microsoft SQL Server, Oracle Database, PostgreSQL, MySQL Cluster и др.). Но все они ориентированы, в основном, на OLTP нагрузку [4], которая характеризуется выполнением множества сравнительно простых операций типа *select* и *insert* над динамически изменяемыми базами данных. Причина в том, что серьезной проблемой для высокопроизводительных реляционных СУБД всегда было и остается повышение скорости обработки сложных запросов [5, 6]. Для консервативных СУБД свойственна OLAP нагрузка [7], которая характеризуется высоким удельным весом сложных запросов типа «селекция – проекция – соединение», оперирующих множеством таблиц с большим числом операций соединения.

Разработки в этом направлении ведутся. В коммерческом секторе уже сейчас последние версии СУБД от Microsoft SQL Server 2016 и 2017 могут быть применены для аналитической обработки в рамках одного узла [8]. Производительность Microsoft SQL Server 2016 на одном сервере Lenovo x3950X6 [9] (8 вычислительных

модулей, 144 ядра) с внешней флеш-памятью ~120 TB и объемом оперативной памяти 12 TB, при совокупной стоимости системы $2 634 342 (стоимость Lenovo x3950X6 ~$1,5 млн., стоимость Windows Server 2016 и SQL Server 2016 Enterprise Edition ~$1 млн.), показывает отличные результаты [10] на тесте TPC-H [11]. У Oracle имеется собственная программно-аппаратная платформа Exadata [12] обладающая примерно такой же стоимостью аппаратуры (~$1,5 млн.) на одну стойку в средней комплектации: 2 сервера БД (192 CPU Cores / 3 TB RAM / 12 TB Flash Storage на каждый сервер) и 14 серверов хранилища (20 CPU Cores / 3 TB RAM / 24 TB Flash Storage / 120 TB HDD Storage на каждый сервер). Что более выгодно, чем Lenovo x3950X6, но программное обеспечение для Exadata в стоимость не входит и приобретается отдельно. Поэтому, если установить на эту платформу СУБД Oracle Database с расширением для OLAP обработки, то к стоимости системы прибавится стоимость ПО ~$9 млн. За столь высокой стоимостью коммерческих систем кроются гарантии производителя в надежности и высокой производительности его системы.

Альтернативой дорогим коммерческим СУБД являются свободные СУБД с открытым исходным кодом, ориентированные на работу с БД повышенного объема (PostgreSQL XL, MySQL Cluster, Spark и др.). Их главное преимущество – доступность и независимость разработки. Но они существенно уступают коммерческим по надежности и, в меньшей мере, по производительности. Кроме того, никто не гарантирует их корректную работу, никто не несет ответственности в случае отказа в работе. Настройка и эксплуатирование таких систем полностью ложится на системных администраторов, которые вынуждены принимать меры по повышению надежности подконтрольных им систем своими силами.

Многие свободные СУБД поддерживаются корпорациями, которые предоставляют платные услуги по их настройке и обслуживанию. Ярким примером такой СУБД является MySQL, которая хоть и является СУБД с открытым исходным кодом, но ее поддержкой занимается Oracle. Стоимость такой поддержки в среднем равна $10 000 на один сервер в год, что гораздо дешевле стоимости оригинальной Oracle Database в разы.

Свободные СУБД позволяют экономить большое количество средств на ПО. Но для достижения приемлемой производительности на БД повышенного объема, они требуют использования самых мощных вычислительных узлов, что не всегда

возможно. Уменьшить стоимость аппаратной платформы представляется возможным заменой одного мощного узла группой маломощных узлов с целью распределения данных и работ по ним. Такой подход применен в СУБД PostgreSQL XL, но ее производительность и надежность оставляют желать лучшего [13].

Разрабатываемые в диссертации прототипы СУБД предназначены для изыскания возможностей реализации экономичных параллельных/распределенных консервативных СУБД повышенных объемов, которые смогут эффективно обрабатывать поток запросов к БД объемом в десятки и сотни GB на сравнительно недорогих кластерных платформах с применением средств MySQL и GPU-акселераторов на исполнительном уровне. Использование готовой СУБД обусловлено тем, что создание полнофункциональной новой СУБД может занять годы при работе большой команды высококвалифицированных разработчиков. Поэтому, чтобы не оказаться в догоняющей позиции, новые отечественные СУБД следует создавать на базе готовых СУБД с открытым кодом и свободной лицензией, поддерживаемых международным сообществом. PostgreSQL является более совершенной СУБД, чем MySQL, и активно позиционируется на территории России [14]. Но MySQL позволяет использовать различные «движки» и имеет систему расширений [15]. Эти особенности упрощают и ускоряют разработку.

Разработанный в диссертации исследовательский прототип параллельной консервативной СУБД Clusterix-N ориентирован на ускорение (в сравнении с СУБД Clusterix [16]) обработки запросов к БД повышенных объемов (не умещающихся в оперативной памяти одного узла). Повышение объема баз данных требует их хеширования по узлам кластера, что обуславливает необходимость использования регулярного плана обработки запросов. Суть предлагаемого подхода состоит в соответствующей организации пакетной передачи данных и использовании GPU-ускорителей.

Альтернативой Clusterix-N при БД объемом, не превышающим размера оперативной памяти одного узла (единицы и десятки GB) является второй разработанный в диссертации исследовательский прототип консервативной СУБД PerformSys. Он ориентирован на обработку большого потока запросов к БД малого объема с полной загрузкой процессорных ядер. Работа прототипа организуется согласно стратегии «ядро на запрос +1» [17]. Система реализована на платформе вы-

числительных кластеров, использует в качестве инструментальной СУБД MySQL и эксплуатирует ряд ее возможностей: выделение отдельного потока под запрос, хранение БД в оперативной памяти. Увеличение объема БД существенно снижает производительность. Для сохранения производительности с ростом объема данных необходимо заменять узлы кластера на более мощные, что не всегда возможно.

В настоящее время правительством РФ взят курс на импортозамещение, в связи с чем серьезное внимание уделяется созданию на базе открытых систем (PostgreSQL, MySQL и др.) отечественных СУБД общего и специального назначения. Поэтому разработка и исследование принципов построения и программной организации сравнительно недорогих отечественных СУБД повышенных объемов является актуальной задачей.

Цели исследований. Основной целью является повышение эффективности (по критерию «производительность/стоимость») экономичных систем консервативных баз данных повышенных объемов при обработке по регулярному плану (используемому в СУБД Clusterix) непрерывных потоков сложных запросов типа «селекция – проекция – соединение» до уровня, сравнимого с технологией Spark, полагаемой в настоящее время наиболее перспективной. Развитие технологии PerformSys, перспективной при умеренных объемах баз данных, служит дополнительной целью.

Задачи исследований. В соответствии с поставленной целью основной задачей исследований является анализ возможностей реализации экономичных консервативных СУБД повышенных объемов – СУБД Clusterix-N, сравнимых по эффективности с системой Spark при обработке потока запросов к БД объемом в сотни GB и более на сравнительно недорогих кластерных платформах с использованием регулярного плана обработки запросов, применением средств MySQL и GPU-акселераторов на исполнительном уровне. Дополнительная задача – архитектурно-алгоритмическая и программная разработка альтернативной системы (PerformSys) при умеренных объемах баз данных и анализ перспектив дальнейших исследований по развитию рассматриваемых IT-технологий.

Решение этих задач связывается с проведением следующих исследований, отраженных в главах 2-5 работы соответственно:

– Изыскание специальной настройки инструментальной СУБД MySQL на максимальную загрузку процессорных ядер при работе с консервативными СУБД; разработка способа претрансляции запросов к регулярному плану; интерпретация методологии конструктивного моделирования систем применительно к решению задачи синтеза консервативных СУБД; выбор и разработка начального состояния этого итеративного процесса.

– Итеративная разработка и исследование иерархической модели синтеза СУБД Clusterix-N согласно методологии конструктивного моделирования систем вплоть до получения состояния этой модели, удовлетворяющего поставленной цели.

– Разработка предложений по архитектуре PerformSys и сравнительное исследование ее эффективности на платформах SUN- и GPU-кластеров при различных объемах БД.

– Анализ перспектив перехода от системы Clusterix-N к системе Clusterix-G с работой со сжатыми БД и более широким использованием GPU-ускорителей. Выяснение перспектив балансировки нагрузки в территориально распределенных консервативных СУБД с целью повышения их пропускной способности.

Научная новизна:

1. Разработан способ претрансляции запросов к регулярному плану, основанный на таком дроблении исходного запроса на SQL-фрагменты, который, в отличие от использованного для Clusterix метода, позволяет использовать его с различными инструментальным СУБД.

2. Предложена интерпретация методологии конструктивного моделирования систем применительно к задаче моделирования процесса синтеза консервативных СУБД класса BigData, которая, в отличие от ранее использованной для Clusterix-подобных СУБД, позволяет добиться большей эффективности системы.

3. Предложен и реализован метод параллельной обработки селективных запросов в СУБД Clusterix-N на уровне IO, основанный на поблочной выборке из СУБД MySQL, что, в отличие от ранее реализованного метода в Clusterix-подобных системах, позволяет использовать все процессорные ядра всех узлов IO для обработки одного селективного запроса с полной загрузкой процессорных ядер.

4. Для СУБД Clusterix-N предложены и реализованы методы сосредоточенной и распределенной динамической сегментации промежуточных/временных отно-

шений с применением GPU-ускорителей, основанные 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 [16] и Clusterix-M [18], позволяет существенно ускорить операции хеширования, загрузить все процессорные ядра всех узлов уровня JOIN и более эффективно использовать сеть.

5. Предложен и реализован в СУБД PerformSys метод распределения потока запросов по процессорным ядрам кластерной платформы с их полной загрузкой, основанный на стратегии «запрос на ядро +1», что, в отличие от реализации в MySQL Router [19], позволяет передавать в узлы ровно столько запросов, сколько они могут эффективно обработать.

6. Выявлена возможность дальнейшего повышения эффективности Clusterix-подобных систем (переход от Clusterix-N к архитектуре Clusterix-G), основанного на работе со сжатыми БД, что, в отличие от применения GPU для выполнения SQL запросов в [20, 21], позволяет увеличить объемы хранимых данных и ускорить их передачу по сети.

7. Предложены методы межрегиональной балансировки нагрузки для территориальной распределенных консервативных СУБД, основанные на подсчете веса очередей и времени активности каждого региона, что, в отличие от общепринятых методов балансировки нагрузки [22], позволяет более равномерно распределять нагрузку по регионам и увеличить эффективность эксплуатации территориально распределенных СУБД.

Научная значимость полученных результатов:

1. Проведенное в диссертации конструктивное моделирование консервативных СУБД с регулярным планом обработки запросов позволило существенно пополнить состав развитых ранее элементов содержательной теории таких СУБД анализом возможностей полной загрузки процессорных ядер, использования гибридных технологий и блочной динамичной сегментации отношений.

2. Разработка в диссертации четырех версий СУБД Clusterix-N и получение в последней итерации моделирования результатов, сравнимых с технологией MapReduce, поможет актуализации широкомасштабных научных исследований по дальнейшему развитию теории консервативных СУБД класса BigData с регулярным планом обработки запросов.

3. Позитивные результаты выполненного в диссертации кросс-моделирования межрегиональной балансировки нагрузки территориально распределенных СУБД консервативного типа показывают целесообразность развертывания серьезных научных исследований в этом направлении.

Практическая значимость. Помещенные в открытый доступ исследовательские прототипы систем PerformSys[1] и Clusterix-N[2] могут быть использованы как действующая платформа для создания экономично-эффективных аналитических систем в организациях с ограниченными финансовыми возможностями и изучения вопросов параллельной/распределенной обработки данных в учебном процессе ВУЗов.

Основные положения, выносимые на защиту:

1. Специальная настройка инструментальной СУБД MySQL на максимальную загрузку процессорных ядер при работе с консервативными СУБД; способ претрансляции запросов к регулярному плану; интерпретация методологии конструктивного моделирования систем применительно к задаче моделирования процесса синтеза консервативных СУБД. *(отвечает пунктам 1 и 8 паспорта специальности 05.13.11).*

2. Оригинальные программные разработки с экспериментальной проверкой, которые требуется выполнять многократно в процессе итеративного исследования – начальное состояние модели и итерации 1 – 4 в процессе разработки модели синтеза СУБД Clusterix-N *(отвечает пунктам 1, 4 и 8 паспорта специальности 05.13.11)*.

3. Методы, алгоритмы и программы СУБД «PerformSys». Результаты экспериментального исследования на платформах SUN- и GPU-кластеров при сравнительно небольших и повышенных объемах БД *(отвечает пунктам 4 и 8 паспорта специальности 05.13.11)*.

4. Результаты предварительной теоретической оценки перспектив перехода от системы Clusterix-N к системе Clusterix-G с работой со сжатыми БД и более широким использованием GPU-ускорителей *(отвечает пунктам 1 и 8 паспорта специальности 05.13.11)*.

---

[1] https://github.com/rozh1/PerformSys
[2] https://bitbucket.org/rozh/clusterixn/

5. Перспективы балансировки нагрузки в территориально распределенных консервативных СУБД с целью повышения их пропускной способности. *(отвечает пунктам 1 и 9 паспорта специальности 05.13.11)*.

Объект исследования. Параллельные и распределенные СУБД консервативного типа на кластерной платформе с графическими ускорителями и без них, основанные на свободных СУБД общего назначения.

Предмет исследования. Модели, алгоритмы и программы параллельных/распределенных СУБД консервативного типа повышенных объемов.

Обоснованность и достоверность результатов диссертации. Обоснованием может служить использование в процессе проведенных исследований методологии конструктивного моделирования систем, развитых элементов теории баз данных, параллельных и распределенных вычислений, объектно-ориентированного программирования. Достоверность подтверждена экспериментально на натурных моделях, разработанных с применением оригинальных программных инструментальных средств.

Соответствие работы специальности ВАК 05.13.11. Работа отвечает следующим пунктам паспорта специальности 05.13.11 (в квадратных скобках указаны соответствующие позиции диссертации):

п.1. *Модели, методы и алгоритмы проектирования* и анализа *программ и программных систем*, *их* эквивалентных преобразований, верификации и *тестирования*. [Модели, методы и алгоритмы проектирования программ и программных систем управления консервативными базами данных. Тестирование разработанных СУБД].

п.4. *Системы управления базами данных* и знаний. [Разработаны исследовательские версии СУБД Clusterix-N и PerformSys].

п.8. *Модели и методы создания программ и программных систем для параллельной и распределенной обработки данных*, языки и инструментальные средства параллельного программирования. [Математическая модель процесса синтеза Clusterix-подобных систем, методы создания программ и программных систем параллельных СУБД на платформе вычислительных кластеров с графическими ускорителями].

п.9. *Модели, методы, алгоритмы* и программная инфраструктура *для организации глобально распределенной обработки данных*. [Модели, методы, алгоритмы межрегиональной балансировки нагрузки в территориально распределенных СУБД].

Апробация работы. Основные результаты работы неоднократно докладывались и обсуждались на Международной молодежной научной конференции «Туполевские чтения» (Казань, 2013, 2015), Международной научно-методической конференции «Информатика: проблемы, методология, технологии» (Воронеж, 2016), Международной конференции и молодежной школы «Информационные технологии и нанотехнологии» (Самара, 2016), Республиканском научном семинаре АН РТ «Методы моделирования» (Казань, 2015-2018), Международной научной конференции «Системы компьютерной математики и их приложения» (Смоленск, 2018), обсуждение на кафедре системного анализа и информационных технологий КФУ (Казань, 2019).

Публикации. Основные результаты по теме диссертации изложены в 14 печатных изданиях, 9 из которых изданы в журналах, рекомендованных ВАК (включая одну SCOPUS-статью), 1 – в РИНЦ-журнале, 4 – в материалах конференций, индексированных в РИНЦ. По выполненным разработкам получено одно свидетельство о государственной регистрации программы для ЭВМ.

Объем и структура работы. Диссертация состоит из введения, пяти глав и заключения. Полный объем диссертации составляет 195 страниц, включая 54 рисунка, 19 таблиц и 6 приложений. Объем основной части диссертации (без приложений) составляет 162 страницы. Список литературы содержит 130 наименований.

ГЛАВА 1. ОБЗОР СОСТОЯНИЯ РАЗРАБОТОК ПАРАЛЛЕЛЬНЫХ СУБД

В этой главе дается анализ положения дел в области параллельных СУБД вообще и аналитических СУБД, в частности. При этом рассматриваются как зарубежные, так и отечественные системы.

1.1. Предварительное рассмотрение

За последние несколько десятков лет информационные технологии шагнули далеко вперед. Компьютеры стали компактными и распространились практически повсеместно. Если ранее для работы с большим массивом информации требовался суперкомпьютер, который мог занять отдельную комнату или даже весь этаж, то сегодня этот же объем может обработать обычный ПК. Например, самый мощный суперкомпьютер в списке TOP500 [23] за июнь 2000 года ASCI Red имел производительность в Linpack равную 2.38 TFlop/s [24], объем оперативной памяти составлял 594 GB [25] и занимал отельный этаж. В настоящее время обычный персональный компьютер с характеристиками CPU Core i5-4670K 3.8 GHz (4 ядра) / RAM 24 GB / Video GTX 770 (1536 ядер CUDA) обладает производительностью более 3 TFlop/s (3.2 TFlop/s – видеокарта [26], 172 GFlop/s – процессор). В сравнении с таким «домашним» компьютером, ASCI Red обладал гораздо бóльшим общим объемом оперативной памяти и количеством процессорных ядер. Но их быстродействие было существенно меньше современного ПК в силу существенно более низких частот и архитектурных особенностей, а также более низкой скорости передачи данных как между узлами, так и внутри узла.

Проводить сравнение ASCI Red с современным «домашним» ПК не совсем корректно. Более верно сравнивать одну суперсистему с другой суперсистемой, т.е. сравнивать ASCI Red с более совершенной вычислительной системой из того же списка TOP500. На ноябрь 2017 года самой производительной системой из списка TOP500 является суперкомпьютер Sunway TaihuLight [27]. Он обладает производительностью в Linpack – 93 014.6 TFlop/s, т.е. более чем в 39 000 раз большей, чем лучший суперкомпьютер 2000 года. Такое увеличение производительности показывает на сколько быстро шло развитие вычислительной техники за последние 17 лет.

Суперкомпьютеры, хоть и обладают внушительной вычислительной мощностью, но чрезвычайно дорогóй как для покупки, так и в обслуживании / эксплуатации. Для большинства организаций будет достаточно всего одного мощного сервера для решения их задач. Но в случае обработки сложных моделей, большого объема данных или выполнения иных ресурсоемких задач в разумное время требуется суперкомпьютер или кластер из нескольких достаточно мощных серверов. Далеко не каждая организация может позволить себе приобретение суперкомпьютера. Наиболее очевидный выход для таких организация – арендовать вычислительные мощности у организаций, занимающихся работой с высокопроизводительными системами, или использовать облачные сервисы, например, AWS [28]. Хорошо, если большие вычислительные мощности требуется всего несколько часов в день или неделю. Но что если они нужны постоянно, требуется повышенная безопасность результатов вычислений или специфичное программное обеспечение? Тогда для организации выгоднее приобрести небольшой кластер (на 5-10 или более узлов). Количество узлов и их оснащение выбирается из соображений стоимости системы и получения результатов за приемлемое время.

Вычислительные кластеры сейчас востребованы для выполнения множества задач в различных областях [29], среди которых область параллельных СУБД занимает особое место, т.к. они могут выступать в качестве подсистем в других системах. Например, могут служить хранилищем для исторических данных метеонаблюдений и их аналитической обработки с последующим применением в моделировании атмосферы, мирового океана, предсказании погоды. Иными словами, использоваться для аналитической обработки больших массивов данных.

Быстрый рост вычислительной мощности позволил реализовать алгоритмы и решить задачи, которые ранее считались не выполнимыми. Удешевление средств хранения данных дало возможность накапливать огромные массивы данных, сохранив при этом сложность их обработки. В результате начало активно развиваться направление «Big Data»:

> *«Больши́е да́нные (англ. big data, [ˈbɪg ˈdeɪtə]) — обозначение структурированных и неструктурированных данных огромных объ-*

> *ёмов и значительного многообразия, эффективно обрабатываемых горизонтально масштабируемыми (англ. scale-out) программными инструментами, появившимися в конце 2000-х годов и альтернативных традиционным системам управления базами данных и решениям класса Business Intelligence*
>
> *В широком смысле о «больших данных» говорят как о социально-экономическом феномене, связанном с появлением технологических возможностей анализировать огромные массивы данных, в некоторых проблемных областях — весь мировой объём данных, и вытекающих из этого трансформационных последствий» [30]*

На сегодняшний день создано множество инструментов и СУБД для работы с большими данными [31]. Примерами СУБД могут служить: MS SQL Server, Oracle Database, SciDB [32], VoltDB [33], PostgreSQL XL [34], Clusterix [16] и т.д. Большинство систем BigData – это закрытые коммерческие продукты с очень высокой стоимостью. Например, СУБД MS SQL Server 2017 для 12 ядерного, двухпроцессорного сервера обойдется как минимум в 85 000$. Такая высока стоимость для конечного потребителя связана с политикой лицензирования СУБД – «на ядро». На каждое ядро в физическом или виртуальном процессоре требуется своя лицензия. Стоимость лицензии да 2 ядра составляет 14 256$ [35]. При этом клиентские лицензии для пользователей не требуются. Открытые системы существенно уступают коммерческим по надежности и, в гораздо меньшей мере, по производительности. Большинство систем, ориентированных в первую очередь на обработку больших массивов данных, построены для работы на платформах вычислительных кластеров и требуют наличия серьезных вычислительных мощностей для обеспечения приемлемой производительности.

Типовая архитектура реляционной СУБД по Стоунбрейкеру и Хеллерстейну [36] включает в себя 5 главных компонентов (рис. 1.1):

1. Менеджер коммуникации с клиентом, включающий протоколы обмена информацией для локальных и удаленных клиентов.

2. Менеджер управления процессами, выполняющий функции диспетчера и планировщика обработки.

3. Менеджер управления транзакциями – управление доступом, блокировкой, журналированием и буфером данных.

4. Общие компоненты и утилиты: пакетные утилиты, службы репликации и загрузки, утилиты для администрирования и мониторинга, менеджеры каталогов и памяти.

5. Процессор реляционных запросов.

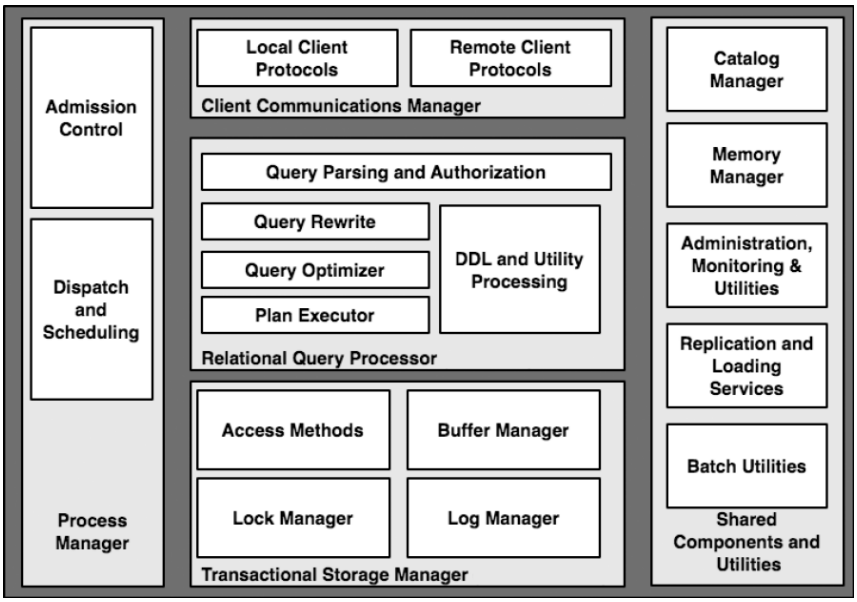


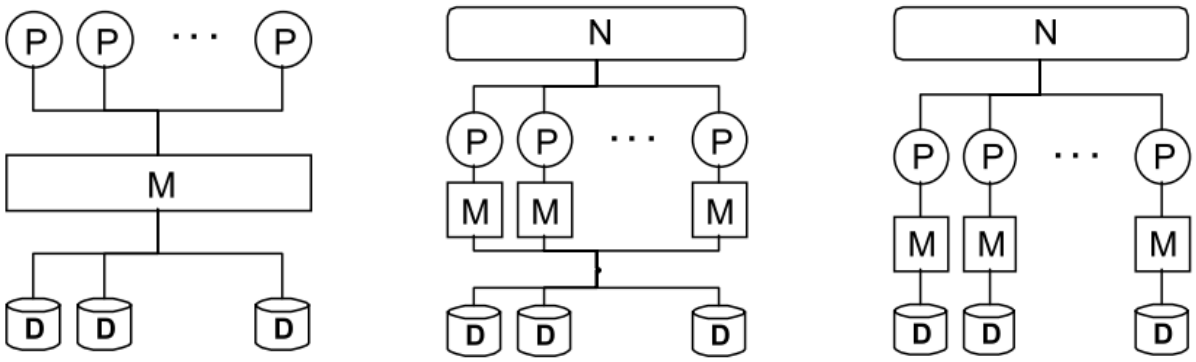
Рис. 1.1. Типичная архитектура реляционной СУБД

Показанная архитектура пригодна для одноузловых последовательных СУБД и активно ими используется. Архитектура параллельных СУБД существенно отличается от одноузловых/последовательных: добавляется коммуникация по сети между модулями, управление процессами выполняется на множестве узлов, утилиты используются различными модулями по мере необходимости, но в целом компоненты все те же, представленные в несколько иной (расширенной) интерпретации.

Классификация архитектур параллельных систем баз данных по Стонбрейкеру [37] ранее являлась наиболее распространенной. Схематичное изображение классификации представлено на рис. 1.2. Здесь: P – процессор, M –оперативная память, D – диск, N – сеть передачи данных.

В соответствии с этой классификацией параллельные СУБД делятся на следующие базовые классы в зависимости от разделения аппаратных ресурсов:

– *SE (Shared-Everything)* – архитектура с разделяемыми памятью и дисками.
– *SD (Shared-Disks)* – архитектура с разделяемыми дисками.



а б в

Рис. 1.2. Классификация Стоунбрейкера (а – SE, б – SD, в – SN)

– *SN (Shared-Nothing)* – архитектура без совместного использования ресурсов.

«*SE АРХИТЕКТУРА* представляет системы баз данных, в которых все диски напрямую доступны всем процессорам с одинаковым временем доступа и все процессоры разделяют общую оперативную память (рис. 1.2а). Межпроцессорные коммуникации в *SE* системах осуществляются через общую оперативную память. Доступ к дискам в *SE* системах обычно осуществляется через общий буферный пул. При этом следует отметить, что каждый процессор в *SE* системе имеет собственную кэш-память.

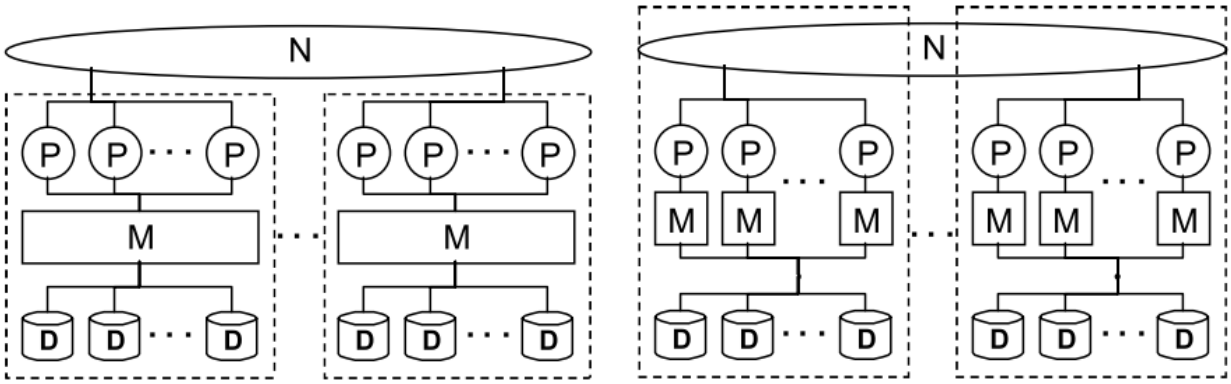
Базовой аппаратной платформой для реализации систем с *SE* архитектурой обычно служит SMP, хотя потенциально *SE* системы можно строить на платформах с архитектурой NUMA и даже MPP с виртуально общей, физически распределенной памятью.

SD АРХИТЕКТУРА (рис. 1.2б) представляет системы баз данных, в которых любой процессор имеет доступ к любому диску, однако каждый процессор имеет свою приватную оперативную память. Процессоры в таких системах соединены посредством некоторой высокоскоростной сети, позволяющей осуществлять передачу данных.

SN АРХИТЕКТУРА (рис. 1.2в) характеризуется наличием у каждого процессора собственной оперативной памяти и собственного диска. Как и в *SD* системах, процессорные узлы соединены некоторой высокоскоростной сетью, позволяющей организовывать обмен сообщениями между процессорами.» [38]

Копеланд и Келлер в работе [39] предложили расширение классификации Стоунбрейкера путем введения двух дополнительных классов архитектур параллельных машин баз данных (рис. 1.3):

– *CE (Clustered-Everything)* – архитектура с *SE* кластерами, объединенными по принципу SN;
– *CD (Clustered-Disk)* – архитектура с *SD*-кластерами, объединенными по принципу *SN*. Граница SD-кластеров на рис. 1.3 распространена на общую (*глобальную*) соединительную сеть, так как в них может присутствовать собственная (*локальная*) соединительная сеть.



а б

Рис. 1.3. Расширение классификации Стоунбрейкера (а – CE, б – CD)

Эти архитектуры также получили название иерархических. На рис. 1.3 изображены двухуровневые иерархии. Однако классификационный подход Копеланда и Келлера легко может быть распространен на архитектуры с тремя и более уровнями иерархии. В качестве примера можно привести трехуровневую иерархическую архитектуру CD 2 (Clustered-Disk with 2-processor modules). Данная архитектура была использована при проектировании системы «ОМЕГА» [40].

«Эрхард Рам в работе [41] предложил рассматривать гибридные архитектуры. Гибридные архитектуры нельзя отнести ни к одному из вышеописанных классов. В качестве примера гибридной архитектуры можно привести архитектуру C_D^N. Она строится как набор однотипных *SD*-кластеров, объединенных по принципу *SN*. Отличительной особенностью данной системной архитектуры является то, что на верхних уровнях системной иерархии *SD*-кластеры рассматриваются как *SN*-системы (рис. 1.4). Это выражается в том, что каждому процессорному узлу логически назначается отдельный диск. Такой подход позволяет избежать проблем, связанных с реализацией глобальной таблицы блокировок и поддержкой когерентности кэшей, характерных для *SD*-систем, и одновременно использовать преимущества *SD*-архитектуры в плане возможности балансировки загрузки.» [42]

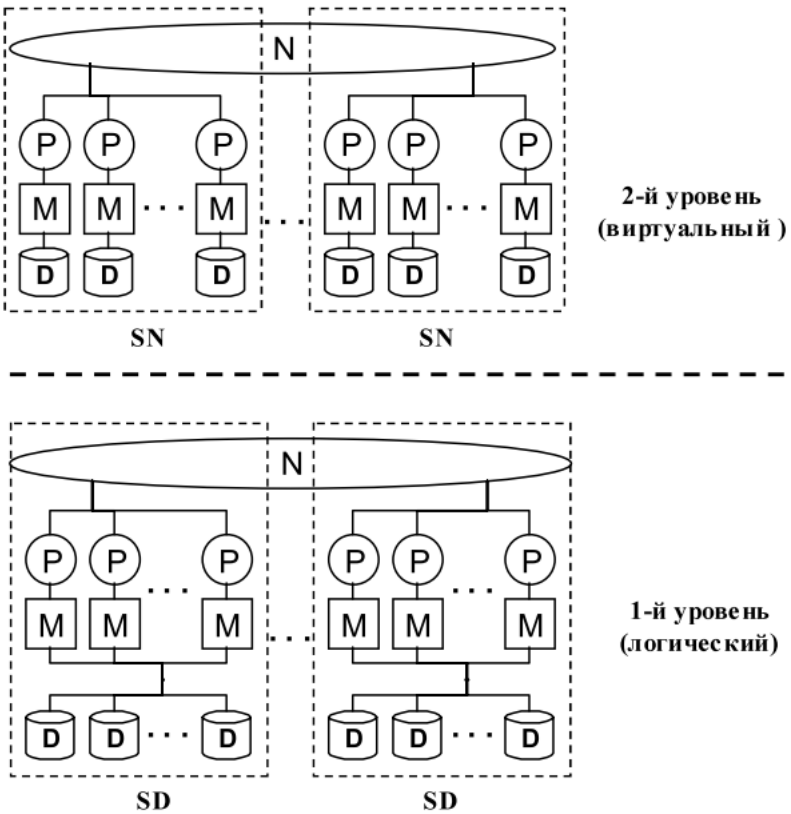


Рис. 1.4. Гибридная архитектура C_D^N

1.2. Обзор популярных СУБД

Чтобы проиллюстрировать положение дел на сегодняшний день в мире аналитических СУБД рассмотрим несколько их представителей: MS SQL Server, Oracle Database, MySQL, PostgreSQL, Postgres-XL, ОМЕГА [40], PostgresPro [43] Clusterix. Первые 4 СУБД выбраны из списка [44], где представлено большое количество различных СУБД с оценкой их популярности. Postgres-XL и Clusterix выбраны в силу их ориентации на работу с вычислительными кластерами.

Все представленные СУБД можно разделить на коммерческие и бесплатные, параллельные и последовательные. К коммерческим относятся СУБД, которые нельзя использовать без приобретения лицензии: MS SQL Server, Oracle Database. К бесплатным относятся СУБД, которые можно использовать в любых проектах на безвозмездной основе, даже если возможна их покупка или платная поддержка: MySQL, PostgreSQL, Postgres-XL, Clusterix. Последовательные СУБД – это СУБД которые могут выполнять один запрос только в одном потоке на 1 процессорном ядре. К таковым относятся MySQL 5 и PostgreSQL 9. Остальные рассматриваемые СУБД обладают параллелизмом обработки запросов либо в рамках одного узла с

балансировкой нагрузки между узлами (MS SQL Server [8]), либо в рамках кластера с использованием всех узлов для обработки одного запроса (Oracle Database, Postgres-XL, Clusterix). Аналогично им более новые версии свободных СУБД (например, PostgreSQL 10 [45]) обзаводятся поддержкой параллельной обработки одного запроса на нескольких процессорных ядрах.

Microsoft (MS) SQL Server может выполнять запросы и операции индексирования в параллельном режиме (несколько процессорных ядер на запрос) для компьютеров с более чем одним CPU [8]. Степень параллельности выполнения запросов (Degree of Parallelism, DOP) определяется автоматически или установкой специально конфигурационной переменной. DOP определяет количество потоков, выделяющихся для обработки того или иного запроса. Например, в случае DOP = 4 для запроса №4 из теста TPC-H [11] получим план, состоящий из 8 потоков выборки данных из таблиц и 4 потока для выполнения операций *join*. Полный план выполнения этого запроса показан на рис. 1.5.

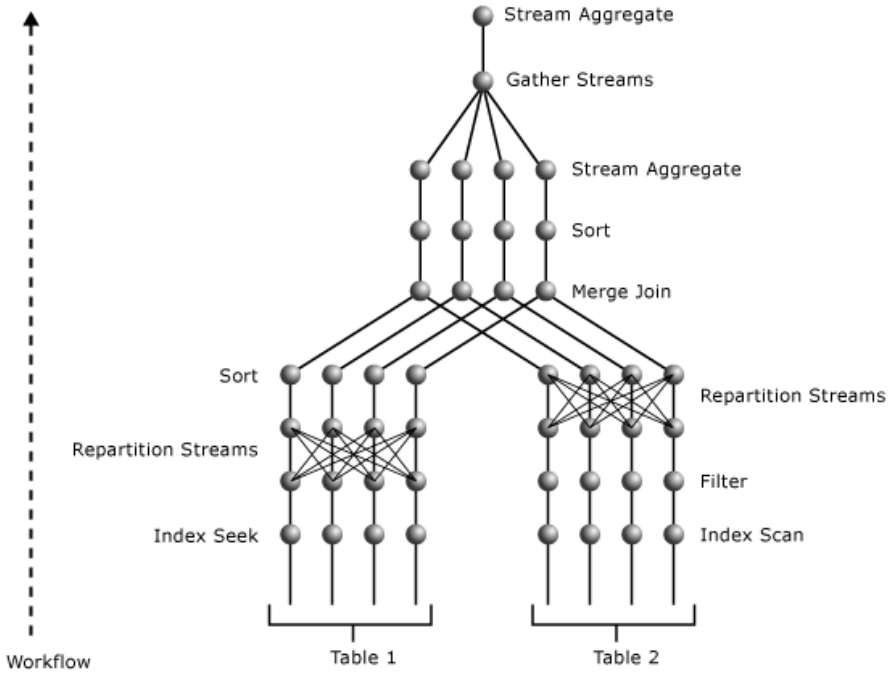


Рис. 1.5. План оптимизатора запросов с DOP=4 и операцией *join* для двух таблиц

Основным недостатком SQL Server является невозможность распределения БД на нескольких узлах в целях параллельной обработки.

Oracle Database, аналогично MS SQL Server, способен выполнять запросы параллельно в рамках одного узла. Но эта СУБД способна выполнять и один запрос на группе узлов [46]. Параллельное выполнение SQL в базе данных Oracle основано на

принципе координатора (Query Coordinator – QC) и процессов параллельного выполнения (Parallel Execution – PX). QC инициализирует параллельные операторы SQL, а серверы PX – выполняют параллельную работу от имени QC. QC распределяет работу между серверами PX, выполняет операции, которые не могут быть распараллелены, и готовит результат для пользователя. Например, для параллельного запроса с операцией `SUM()` в конце, требуется окончательное сложение всех индивидуальных под-итогов, рассчитанных каждым сервером PX, которое выполняется QC.

Серверы PX выбираются из глобально пула доступных PX-серверов и назначаются для заданной операции. Поскольку серверы PX получают необходимые данные с подключенных сетевых хранилищ, любой PX может быть выбран для любой операции (*select, join*). Они могут выбираться автоматически системой или же назначены вручную системным администратором. Например, на рис. 1.6 показана схема распределения данных при выполнении параллельного *join*: 2 сервера PX выделены для выполнения операция *scan* (*select*) и 2 сервера – *join*, результат агрегируется в QC.

Рис. 1.6. Распределение данных по PX серверам для параллельного *join*

Главное преимущество Oracle Database – обильный функционал этой СУБД, позволяющей ей работать как на относительно маломощных серверах, так и на много-узловых вычислительных кластерах. Но это преимущество порождает и главный недостаток – высокую стоимость.

Как уже было отмечено выше, построение высокоэффективной и надежной системы обработки данных на основе СУБД MS SQL Server 2017 оказывается довольно дорогим. Но, СУБД Oracle Database 18c обладает бóльшими возможностями

и, как следствие, гораздо большей стоимостью. Так, при лицензировании «на ядро» (аналогично MS SQL Server) стоимость СУБД для 12 ядерного двухпроцессорного сервера обойдется как минимум в 285 000$, т.е. 47 500$ [47] за каждые 2 ядра процессора. Поэтому СУБД от Microsoft хоть и менее совершенная, но значительно более привлекательная по цене для организаций.

MySQL и PostgreSQL. В качестве альтернативы дорогих коммерческих продуктов выступает программное обеспечение с открытым исходным кодом. У такого ПО есть ряд других преимуществ перед коммерческим ПО:

- Цена – открытое ПО распространяется на безвозмездной основе, но с прилагаемым лицензионным соглашением, в котором описывается условия распространения и использования данного программного продукта.
- Безопасность – открытый исходный код минимизирует возможность установки в ПО различных вредоносных закладок разработчиками.
- Возможность изменения и дополнения исходного кода для конкретных задач.
- Кроссплатформенность – большинство открытых проектов могут быть скомпилированы для различных платформ и ОС.

Яркими представителями открытых СУБД являются MySQL и PostgreSQL. Именно они наиболее часто используются при разработке веб-приложений и в проектах с открытым исходным кодом. Обе СУБД достаточно хороши в работе с небольшими БД и в обработке простых селективных запросов. Какая-либо параллельная обработка отсутствует (один запрос исполняется в одном потоке). Балансировка нагрузки по серверам осуществляется внешними сервисами и службами (для MySQL – NDB Cluster [48] и MySQL-router [19], для PostgreSQL – Pgpool-II [49] и Slony [50]), а доступ к данным осуществляется либо через общий распределенный файловый сервер, либо к локальному хранилищу, синхронизированному с главным узлом.

В сравнении с MySQL, PostgreSQL является более совершенной СУБД [51], т.к. в большей степени поддерживает стандарт ISO/IEC 9075 (SQL-2008) [52], полностью выполняет требования ACID [53]. Но MySQL обладает большей гибкостью в плане модификаций, т.к. имеет систему подключаемых модулей [15]. Функционал MySQL может быть расширен за счет добавления новых операторов SQL, добавления пользовательских функций или добавления нового движка хранения дан-

ных и обработки запросов без необходимости перекомпиляции СУБД в целом. Данный факт делает эту СУБД более привлекательной внесения изменений, чем PostgreSQL, и ускоряет разработку.

Общий недостаток обоих СУБД – отсутствие возможности обработки одного запроса на множестве узлов. В тоже время PostgreSQL, в отличии от MySQL, может выполнять один запрос на нескольких процессорных ядрах одного узла.

PostgreSQL XL. PostgreSQL не имеет функционала параллельной обработки на нескольких узлах, но существует ее модификация – PostgreSQL XL, которая реализует как внутриузловой, так и межузловой параллелизм для одного запроса. Внутриузловой параллелизм реализуется выполнением отдельных операторов запроса на множестве процессорных ядер. Межузловой параллелизм использует распределение данных БД по узлам и позволяет выполнять отдельные операции на всех узлах параллельно (*select, join*) с перераспределением данных по узлам во время обработки запроса.

Типовая конфигурация Postgres-XL [54] показана на рис. 1.7.

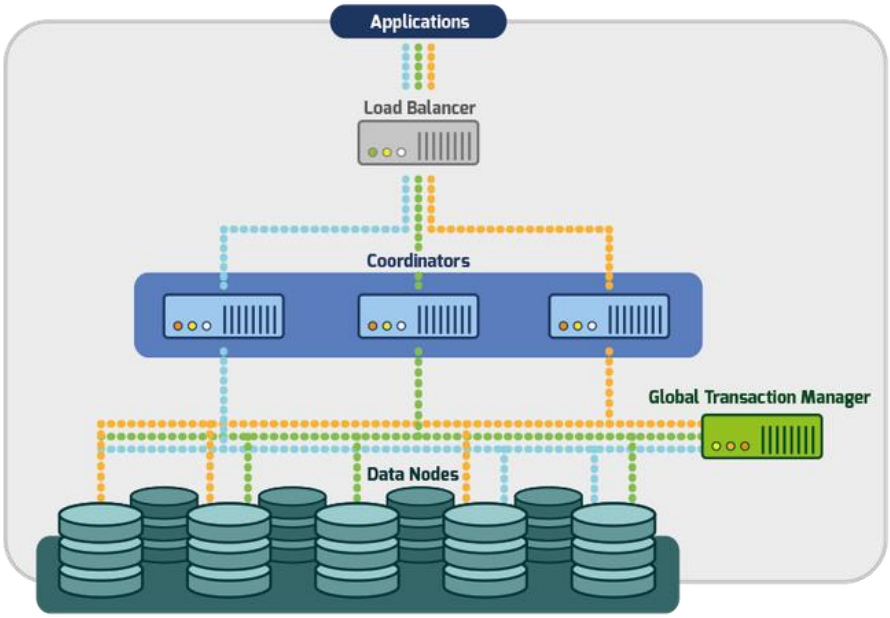


Рис. 1.7. Типовая конфигурация PostgreSQL XL

Здесь [55]:

- Load Balancer – балансировщик нагрузки для входящих соединений
- Coordinator – модуль управления процессом выполнения запроса.
  - Производит обработку входящего запроса, его разбор.
  - Создает план выполнения запроса.
- Выполняет итоговые операции после выполнения запроса (операции *sort* и передачи результата пользователю)

– Global Transaction Manager – отлеживает состояние системы, поддерживает консистентность данных между запросами, хранит информацию о текущих транзакциях.
– Data Node – хранит данные и выполняет основную часть работы: *select-project-join*.

Экспериментальные исследования производительности Postgres-XL [13] на тесте TPC-H показали 10 кратное ускорение выполнения половины запросов по сравнению с PostgreSQL. Эксперимент проводился на платформе AWS, где был построен кластер из 16 узлов i2.xlarge. Узлы i2.xlarge [56] характеризуются наличием 4-х процессорных ядер с частотой 2.3 GHz, 30 GB RAM, 800 GB SSD, сетью с пропускной способностью до 10 Gigabit/s. Узлы Coordinator и Global Transaction Manager разворачивались на отдельных узлах с 16 процессорными ядрами. Размер тестовой базы данных составлял 3TB. БД распределялась по всем 16 узлам Data Node.

На рис. 1.8 представлены результаты проведенного эксперимента. Нетрудно видеть, что набольшее ускорение получили запросы с наименьшим количеством сетевых передач. Например, запрос Q1 ускорен линейно (в ~16 раз на 16 узлах), запрос Q9 был ускорен чуть более чем в 3 раза. Запрос Q9 характеризуется боль-

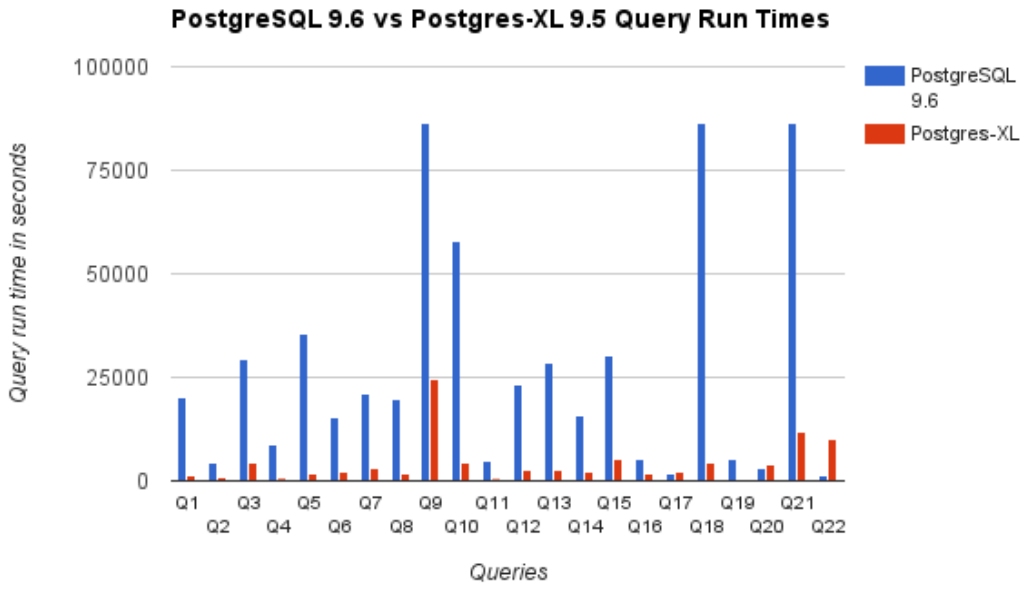


Рис. 1.8. Время обработки запросов теста TPC-H на PostgreSQL и Postgres-XL

шим удельным весом операций *join* над большими объемами данных и требует множества сетевых передач. Еще один медленный запрос Q22 производит тройное сканирование БД в подзапросах, что вызывает чрезмерный трафик в сети. В результате запрос Q22 эффективнее исполняется на одном сервере PostgreSQL, чем на

кластере с Postgres-XL. Можно заключить, что Postgres-XL эффективно справляется с OLAP нагрузкой, но план выполнения запроса не всегда оптимален. Повышение производительности требует дальнейшей разработки.

1.3. Обзор отечественных разработок

Из отечественных разработок в области параллельных СУБД можно выделить параллельную СУБД для мультипроцессорных вычислительных систем с кластерной архитектурой «ОМЕГА», Postgres Pro, PargreSQL [57], параллельную СУБД консервативного типа Clusterix, а также ее модификацию Clusterix-M [18].

СУБД «ОМЕГА» разрабатывалась для многопроцессорной вычислительной системы с массивно-параллельной архитектурой [58]. Такие системы хорошо зарекомендовал себя при решении большого спектра задач и могут быть использованы в качестве сверхмощного параллельного сервера баз данных.

Структура СУБД «ОМЕГА» включает в себя аппаратно-зависимый и аппаратно-независимый уровни. Ядро СУБД, реализующее виртуальную машину баз данных, представляет собой аппаратно-независимый уровень. Аппаратно-зависимый уровень делится на технологическое и системное окружение. Технологическое окружение содержит модули, обеспечивающие возможность эффективного профилирования и отладки системы. Операционное окружение содержит подсистемы, обеспечивающие межкластерный и внутрикластерный обмен сообщениями, операции обмена с дисками и поддержку легковесных процессов.

Архитектура СУБД «ОМЕГА» [59] позволяет выполнять гибкую конфигурацию в зависимости от аппаратного окружения. СУБД разделяется на кластеры, внутри которых данные распределены между узлами. Между кластерами данные реплицируются и в каждом из кластеров находится полная копия. Выполнение запроса может быть произведено как в рамках одного кластера, так и в рамках всех кластеров. Данный факт положительно влияет на возможности масштабирования этой СУБД.

Но данная СУБД в настоящее время не развивается и не существует ее версии для современных вычислительных серверов – в этом ее главный недостаток.

Postgres Pro – Российская СУБД, разработанная компанией Postgres Professional на основе свободно-распространяемой СУБД PostgreSQL. Postgres Pro

входит в реестр российского ПО [60]. Данная СУБД существенно расширяет функционал PostgreSQL [61]:

- Улучшения производительности на многоядерных системах
- Усовершенствования полнотекстового поиска
- Покрывающие индексы. Поддержка конструкции INCLUDING в CREATE INDEX.
- Переносимость: поддержка библиотеки libicu на всех платформах, что обеспечивает однозначную обработку порядка сортировки и прочих операций с символами UNICODE. На ряде платформ эта библиотека улучшает производительность сортировки, и, что немаловажно, позволяет в Postgres Pro пользоваться abbreviated keys, которые были отключены в основной версии PostgreSQL.
- Модуль pg_trgm поддерживает не только нечеткое сравнение строк, но и нечеткий поиск подстроки и т.д.

Полный список возможностей PostgreSQL и PostgresPro приведен в [62].

PargreSQL является успешной попыткой внедрения параллелизма в открытую СУБД PostgreSQL, путем модификации ее исходного кода. В работе [57] приводятся методы внедрения фрагментного параллелизма в открытые СУБД и их реализация применительно к PostgreSQL. Суть модификаций заключалась в введении специального оператора обмена и реализации модуля тиражирования запросов.

Оператор обмена служит для получения корректных результатов в ходе обработки запроса в параллельной СУБД и обеспечения обмена кортежами между вычислительными узлами. Оператор вставляется в план запроса на этапе построения параллельного плана запроса и инкапсулирует в себе передачу кортежей между экземплярами СУБД на различных узлах вычислительной системы. Причем вставляется он только в те места плана, где требуется обеспечить распределение кортежей между вычислительными узлами по некоторому правилу.

Модуль тиражирования запросов обеспечивает передачу запросов всем подключенным узлам. Каждый узел выполняет запрос над собственным множеством данных. За корректностью обработки следит подсистема параллелизатора, которая обеспечивает создание параллельного плана запроса путем вставки операторов обмена в нужные места последовательного плана запроса. Обмен данными перед выполнением операции необходим, т.к. в противном случае на узлах может не оказаться необходимых данных и запрос в целом окажется выполненным неверно.

В экспериментах на исследование ускорения СУБД PargreSQL исполняла запрос, предполагающий выполнение операции естественного соединения двух отношений по общему атрибуту. Соединяемые отношения имели размеры 300 млн. и 7.5 млн. кортежей соответственно, распределяемые равномерно по узлам кластера.

В экспериментах, исследовавших расширяемость, СУБД PargreSQL исполняла запрос, предполагающий выполнение операции естественного соединения двух отношений по общему атрибуту. Кортежи отношений равномерно распределены по узлам кластера. Размеры соединяемых отношений увеличивались пропорционально увеличению количества используемых узлов кластера с множителем 12 млн. и 0.3 млн. кортежей соответственно (т.е. при использовании 128 узлов осуществлялось соединение отношений из 1 536 млн. и 38.4 млн. кортежей соответственно).

Результаты обоих экспериментов показали, что ускорение и расширяемость разработанной СУБД на множестве узлов от 1 до 128 близки к линейной.

Хоть данная СУБД и преодолевает главный недостаток PostgreSQL, но обладает существенным недостатком: она требует полной копии БД на каждом узле.

Clusterix и Clusterix-M. В этих СУБД применяется регулярный план обработки [63] (рис. 1.9) по схеме:

СЕЛЕКЦИЯ (σ) – ПРОЕКЦИЯ (π) – СОЕДИНЕНИЕ (σ_θ (R x S)).

Здесь < x > – декартово произведение, селектирование в операции соединения ведется по θ-соответствию кортежей отношений R и S. По условию соединение всегда естественное и проводится по полям первичного ключа. При использовании стратегии «*множество узлов кластера – на один запрос*» база данных оказывается

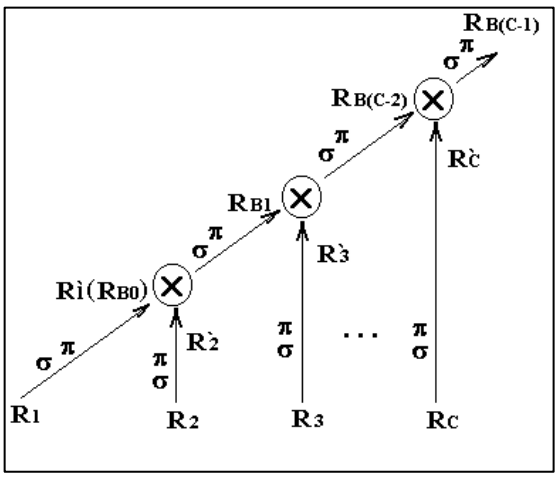


Рис. 1.9. Регулярный план

распределенной по узлам. Получение любого промежуточного R_i' и любого временного $R_{B(i-2)}$ отношений происходит параллельно. При этом теоретически воз можно совмещение обоих процессов, если за время съема с дисков и предварительной обработки (селекции с проекцией) исходного отношения R_i успевает сформироваться временное отношение $R_{B(i-2)}$.

Обработка запросов в Clusterix – 2-уровневая. На нижнем уровне выполняются операции селекции (σ) и проекции (π) над исходными отношениями R_i базы данных. Результатом обработки этого уровня является промежуточное отношение R_i'. На верхнем уровне реализуется операция соединения $R_{Bi-1} = \pi(\sigma_\theta(R_i' x R_{Bi-2}))$, где R_{Bi-1} и R_{Bi-2} – временные отношения как результаты соединений в i- и в (i-1)-шагах соответственно. Фильтрация на нижнем уровне значительно снижает объемы данных, передаваемых на верхний уровень.

Отношения БД распределены по дискам на процессорах нижнего уровня (IO_r) с применением хеш-функции к первичному ключу для каждого кортежа отношения. Требованию равномерного распределения кортежей хорошо удовлетворяет выбор в качестве нее функции деления по модулю [64]. Основание деления – число процессоров нижнего уровня. Остаток от деления r однозначно определяет номер страницы (процессора IO_r), куда будет распределен кортеж. Процессоры верхнего уровня обработки запроса называются процессорами JOIN.

Кроме исполнительных процессоров (IO_r и $JOIN_j$), есть еще два процессора. Процессор УПР реализует функции мониторинга и управления остальными процессорами системы. Для реализации функций объединения результатов обработки с уровня JOIN введен процессор SORT. Он дополнительно выполняет функции агрегации и сортировки результата предшествующей операции соединения. В *Clusterix* реализован вариант совместной работы процессоров УПР и SORT на Host ЭВМ. Работа на уровне файловой системы, системных буферов, алгоритмов доступа к данным, работы с индексами и т.п. реализуется с помощью СУБД MySQL.

Система реализует потоково-конвейерный механизм обработки запросов. Конвейерность достигается за счет совмещения работ на нижнем, верхнем уровнях обработки и на уровне SORT. Для обеспечения надежной работы системы на всех уровнях используется барьерная синхронизация. Применение динамической сег-

ментации промежуточных и временных отношений позволяет реализовать параллельное выполнение операции соединения на множестве процессоров $JOIN_j$. Вместе с тем, это приводит к параболической зависимости времени T обработки представительского теста ПТ от числа узлов кластера h из-за увеличения нагрузки по интерконнекту с ростом h.

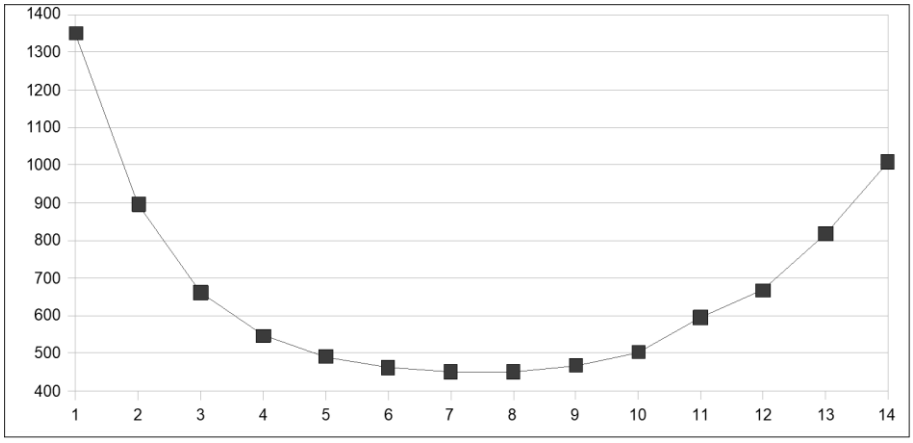


Рис. 1.10. Монокластер, «линейка», ПТ

На рис. 1.10 показан результат эксперимента [18] на натурной модели СУБД Clustrerix, полученный для кластерной платформы: вычислительный кластер фирмы SUN из 22 узлов 2 Quad-core Intel Xeon E5450CPU/1,87GHz/32GB. Интерконнект между узлами – Gigabit Ethernet. В качестве ПТ использовался ограниченный тест TPC-H из 14 запросов, не содержащих операций записи. Граница масштабируемости по числу узлов h_G отвечает точке минимума графика T = f(h). Для рис. 1.10 значение h_G= 7. Наличие двух процессоров в узле позволяет существенно улучшить масштабируемость «линейки» и повысить ее максимальное быстродействие на границе масштабируемости. Это достигается установкой на каждый SMP-узел вычислительного кластера двух серверов MySQL. Один из них связан с процессами IO, другой – с процессами Join (конфигурация «совмещенная симметрия»).

Переход к мультикластеризации (СУБД *Clusterix-M* [18]) означает изменение идеологии обработки запросов в кластерной системе: от «множества маломощных узлов на один запрос» к одному мощному узлу на каждый запрос». При этом «мощный узел» реализуется как монокластер – компонент кластера в целом с установкой на него системы *Clusterix*. Число узлов монокластера не превышает грани масштабируемости. Host (главная ЭВМ) включает модуль ROUTER, который распределяет поток запросов, поступающих от N клиентов между n монокластерами-компонен-

тами. Очередной запрос от каждого пользователя поступает на вход сервера СУБД сразу, как будет получен ответ на предыдущий запрос. В любой момент времени суммарное число запросов, находящихся в сервере, $L = N \geq n \cdot r$. Оптимальная длина очереди монокластера $r = 2$.

На рис. 1.11 показаны результаты обработки на прежней платформе конкатенации трех запросов ПТ с объемом базы данных 5GB. Монокластеры-компоненты конфигурировались как «совмещенная симметрия». Варьировалось количество монокластеров n и число их узлов h_m. В случае $h_m = 4$ наблюдался рост масштабируемости мультикластера по сравне нию с однокластерной системой в 2,3 раза; при $h_M = 6$ – более чем в 2,6 раз. Степень роста производительности на пороге масштабируемости – в 2,4 и не менее чем в 3 раза соответственно.

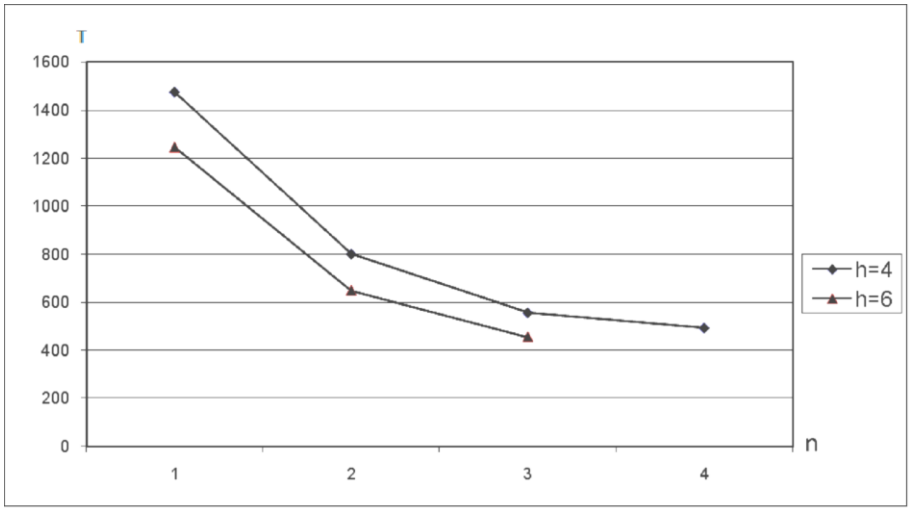


Рис. 1.11. Мультикластер, «совмещенная симметрия», конкатенация 3-х перестановок ПТ

Созданные исследовательские версии систем Clusterix [16] и Clusterix-M [18] были малоэффективны. Но в процессе их исследований были разработаны элементы содержательной теории консервативных СУБД. Основные из них:

1. Для всех Clusterix-подобных систем характерна параболическая зависимость времени обработки представительского теста от n – числа узлов кластера. Минимум параболы определяет границу масштабируемости $n_{гр}$.
2. С ростом объемов БД граница масштабируемости сдвигается вправо.
3. Серьезное влияние на производительность системы оказывает увеличение суммарного времени ожидания t_{wait} барьерной синхронизации с ростом числа узлов кластера. Величина t_{wait} доминирует над другими задержками при $n = n_{гр}$.
4. При выходе за пределы $n_{гр}$ могут наблюдаться сбои в сети передачи данных.

5. Целесообразен переход к мультикластерной конфигурации, когда отдельные монокластеры в составе кластера работают вблизи грани масштабируемости.

1.4. Обзор подхода NoSQL

MapReduce [65] – модель распределенных вычислений, представленная компанией Google, используемая для параллельных вычислений над очень большими, вплоть до нескольких петабайт [66], наборами данных в компьютерных кластерах.

Работа MapReduce состоит из двух шагов (рис. 1.12): Map и Reduce. На шаге Map происходит предварительная обработка входных данных. Для этого один из компьютеров (называемый главным узлом – master node) получает входные данные задачи, разделяет их на части и передает другим компьютерам (рабочим узлам – worker node) для предварительной обработки. На шаге Reduce происходит свертка предварительно обработанных данных. Главный узел получает ответы от рабочих узлов и на их основе формирует результат – решение задачи, которая изначально формулировалась.

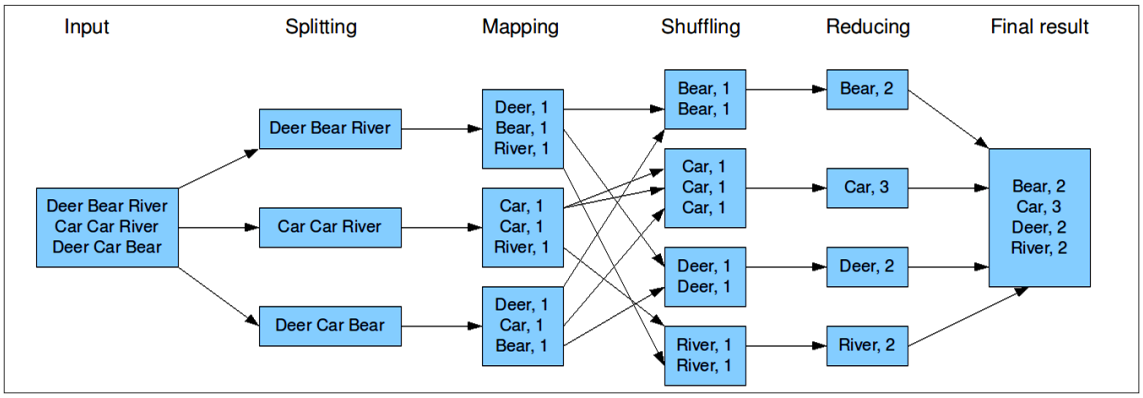


Рис. 1.12. Процесс работы MapReduce

Преимущество MapReduce заключается в том, что он позволяет распределено производить операции предварительной обработки и свертки. Операции предварительной обработки работают независимо друг от друга и могут производиться параллельно. Аналогично, множество рабочих узлов может осуществлять свертку – для этого необходимо только, чтобы все результаты предварительной обработки с одним конкретным значением ключа обрабатывались одним рабочим узлом в один момент времени.

Хотя этот процесс может быть менее эффективным по сравнению с более последовательными алгоритмами, MapReduce может быть применен к большим объ-

емам данных, которые могут обрабатываться большим количеством серверов. Например, MapReduce может быть использован для сортировки данных объемом в несколько ПБ, что займет лишь несколько часов. Параллелизм также дает некоторые возможности восстановления после частичных сбоев серверов: если в рабочем узле, производящем операцию предварительной обработки или свертки, возникает сбой, то его работа может быть передана другому рабочему узлу.

Hadoop [67] – проект фонда Apache Software Foundation, свободно распространяемый набор утилит, библиотек и фреймворк для разработки и выполнения распределенных программ, работающих на кластерах из сотен и тысяч узлов [68]. Используется для реализации поисковых и контекстных механизмов многих высоконагруженных веб-сайтов. Разработан на Java в рамках вычислительной парадигмы MapReduce, согласно которой приложение разделяется на большое количество одинаковых элементарных заданий, выполнимых на узлах кластера и естественным образом сводимых в конечный результат.

Проект состоит из четырех модулей:

1. Hadoop Common – связующее программное обеспечение – набор инфраструктурных программных библиотек и утилит, используемых для других модулей и родственных проектов.
2. HDFS – распределенная файловая система.
3. YARN – балансировщик нагрузки – система для планирования заданий и управления кластером.
4. Hadoop MapReduce – платформа программирования и выполнения распределённых MapReduce-вычислений.

Архитектура Hadoop представлена на рис. 1.13.

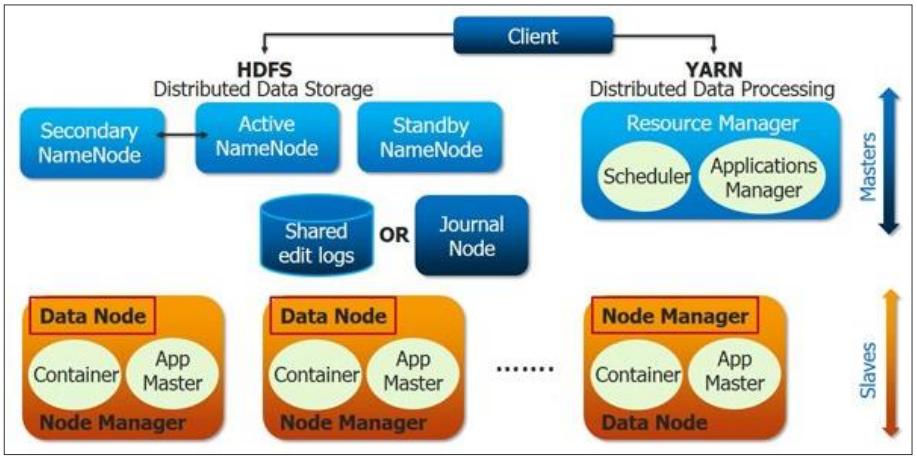


Рис. 1.13. Архитектура Hadoop

Apache Spark [69, 70]– фреймворк с открытым исходным кодом для реализации распределенной обработки неструктурированных и слабоструктурированных данных, входящий в экосистему проектов Hadoop. В отличие от классического обработчика из ядра Hadoop, реализующего двухуровневую концепцию MapReduce с дисковым хранилищем, Spark использует специализированные примитивы для рекуррентной обработки в оперативной памяти, благодаря чему позволяет получать значительный выигрыш в скорости работы для некоторых классов задач.

Проект предоставляет программные интерфейсы для языков Java, Scala, Python, R. Изначально написан на Scala, впоследствии добавлена существенная часть кода на Java для предоставления возможности написания программ непосредственно на Java. Состоит из ядра и нескольких расширений, таких как Spark SQL (позволяет выполнять SQL-запросы над данными), Spark Streaming (надстройка для обработки потоковых данных), Spark MLlib (набор библиотек машинного обучения), GraphX (предназначено для распределённой обработки графов). Может работать как в среде кластера Hadoop под управлением YARN, так и без компонентов ядра Hadoop.

Архитектура Apache Spark показана на рис. 1.14.

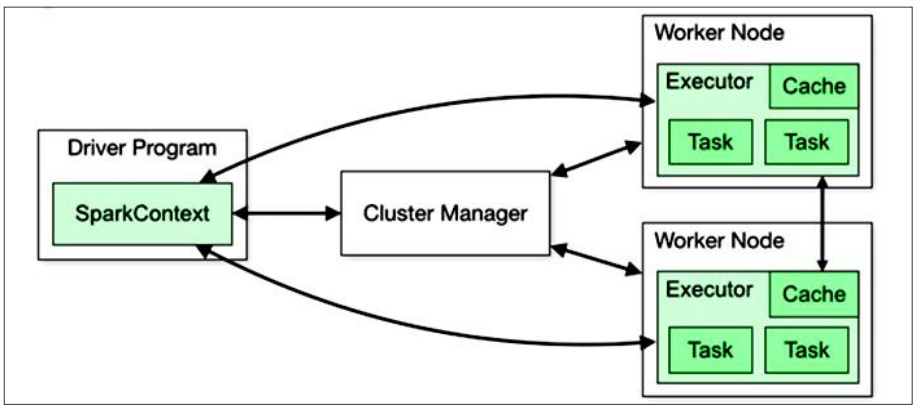


Рис. 1.14. Архитектура Spark

Hadoop и Spark являются разработками с открытым исходным кодом, поэтому их использование не требует закупки лицензий. Требования к аппаратному обеспечению, предъявляемые данными системами весьма скромны (минимум 8GB RAM, 1 диск, 2 ядра процессора на узел). Интерконнект, обеспечивающий достаточную пропускную способность, может быть реализован на основе GigabitEthernet. Все это делает Hadoop и Spark идеальными системами для обработки больших массивов данных на большом количестве относительно маломощных узлов.

Тем не менее в связке Hadoop + Spark все же имеется ряд недостатков. Во-первых, требуется запись результата на диск. Во-вторых, во время обработки производится полное чтение всех данных.

1.5. Выводы по главе 1

В настоящее время существует множество параллельных СУБД. В них реализованы различные подходы к обработке запросов. Одни способны задействовать все процессорные ядра в рамках одного узла, другие могут выполнять один запрос на группе узлов для выполнения одного запроса. Все они различаются архитектурой, планом выполнения запросов, алгоритмом работы. Каждая СУБД имеет свои плюсы и недостатки. Общее сравнение рассмотренных СУБД представлено в таблице 1.1.

Таблица 1.1. Сравнение популярных СУБД

Параметр	**MS SQL Server 2017**	**Oracle Database 18c**	**MySQL 5.7**	**Postgre SQL 9.6**	**Postgre SQL XL 9.5**	**Postgres Pro**	**ОМЕГА**	**Clusterix / Clusterix-M**	**Hadoop / Spark**
Стоимость на «ядро»	7128$	23750$	–	–	–	–	–	–	–
Внутриузловой параллелизм[3]	есть	есть	нет	нет	есть	есть	есть	есть	есть
Межузловой параллелизм[4]	нет	есть	нет	нет	есть	нет	есть	есть	есть
Распределение данных	нет	есть	нет	нет	есть	нет	есть	есть	есть
Балансировка нагрузки	есть	есть	нет	нет	есть	есть	есть	есть	есть
Параллельная обработка множества запросов	есть	есть	есть	есть	есть	есть	нет	нет	нет
Сложность администрирования	низ.	низ.	сред.	сред.	выс.	сред.	выс.	выс.	сред.
Качество документации	выс.	выс.	выс.	выс.	сред.	выс.	низ.	низ.	сред.
Сертификат ФСТЭК	нет	нет	нет	нет	нет	есть	нет	нет	нет

Коммерческие СУБД обладают большой стоимостью, но их производитель дает определенные гарантии по возможностям и надежности своих программных продуктов. Этот фактор чрезвычайно важен в промышленных системах. Производители коммерческих систем несут полную ответственность за их корректную работу в соответствии с предоставленным описанием. В стоимость таких СУБД включается получение обновлений, помощи в настройке и эксплуатации на протяжении года, доступ к качественной и подробной документации.

[3] Выполнение одного запроса на нескольких ядрах / процессорах одного узла
[4] Выполнение одного запроса на множестве узлов кластера

И все же популярные открытые СУБД поддерживаются большим числом разработчиков по всему миру. Их применение значительно удешевляет построение кластерных систем, т.к. не требует покупки лицензий на множество узлов. Но они не гарантируют безотказную работу, корректность алгоритмов и целостность данных. Их разработчики не несут ответственности за сбои в ПО. Оперативная поддержка и доступ к подробной документации по таким СУБД нередко требует дополнительной оплаты. Все это создает трудности использования открытого ПО в критически важных системах.

Что касается исследовательских проектов (PostgreSQL XL, PargreSQL, Clusterix и др.), то они только проверяют различные подходы к решению задачи и не предназначены для использования в промышленных системах.

ГЛАВА 2. ПОСТАНОВКА ОСНОВНОЙ ЗАДАЧИ

Как отмечено во введении, основной задачей данной работы является анализ возможностей реализации экономичных консервативных СУБД повышенных объемов, сравнимых по эффективности (по критерию производительность/стоимость) с системой Spark при обработке потока запросов к БД объемом в сотни GB и более на сравнительно недорогих кластерных платформах с использованием регулярного плана обработки запросов, применением средств MySQL и GPU-акселераторов на исполнительном уровне.

2.1. Принятые ограничения

Согласно требованию экономичности платформой исследуемых СУБД являются вычислительные кластеры, собранные из зарубежных комплектующих отечественными фирмами. SMP-узлы кластера – 2-процессорные, оснащенные инструментальной СУБД MySQL и операционной системой семейства Linux или Windows. Процессоры в узлах – серийные разработки с числом процессорных ядер не более 8. Допускается подключение к узлам по шине PCI-e GPU-ускорителей. Сеть связи между узлами – GigabitEthernet (10GigabitEthernet/Infiniband – по возможности). Дисковая подсистема – SATA (SAS – по возможности). Объем оперативной памяти в узле – не более 512 GB. Рассматриваемые СУБД являются многопользовательскими системами с пакетной обработкой запросов.

Соответственно все исследовательские эксперименты были проведены на платформе GPU-кластера, состоящего из 7 узлов: 6 исполнительных и один управляющий (MGM). 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 объемом 2 TB. Операционная система на исполнительных узлах – Windows Server 2012 R2, на управляющем – CentOS 7 Linux. Интерконнект между узлами – GigabitEthernet с 24-портовым коммутатором SSE G24-TG4. Объемы БД – 60 и 120 GB. Представительский тест (ПТ) – конкатенация 6 перестановок TPC-H Throughput Test без операций записи. В качестве инструментальной СУБД использовалась MySQL версии 5.7 [71].

В эксперименте со Spark БД была представлена в виде структурированных текстовых файлов и равномерно распределена по 6 исполнительным узлам. Доступ к данным был реализован с помощью Hadoop (HDFS). Балансировка нагрузки производилась модулем YARN, также входящим в состав Hadoop. Обработка запросов выполнялась Spark в конфигурации «worker на ядро» (итого 6x12=72 worker'a на кластер). Запросы запускались в работу без каких-либо изменений и оптимизаций. За работу с SQL-запросами отвечало расширение Spark spark-sql, которое выполняло разбор и оптимизацию исходного запроса. Результаты экспериментов таковы: при $V_{БД}$ = 60 GB – 3,3 часа, при $V_{БД}$ = 120 GB – 4,5 часа.

2.2. Предлагаемый метод претрансляции запросов

Прежде, чем выполнить запрос, его необходимо претранслировать к регулярному плану [63] (рис. 1.9): выделить все необходимые для работы поля всех отношений, составить подзапросы для получения промежуточных отношений R’i, составить подзапросы выполнения операций соединения с указанием их порядка и участвующих отношений (R’i, результат соединения или пустое отношение), составить подзапрос *sort* с указанием исходного отношения как результата последнего *join*.

Претранслятор запросов в Clusterix-N программно не реализован. Для целей исследований вполне достаточна его «ручная» реализация для ограниченного множества тестовых запросов. Внутренний формат запроса в Clusterix-N отвечает регулярному плану. Он содержит отдельные коллекции для всех подзапросов (*select-project*, *join*, *sort*) и включает всю необходимую информацию для работы с ними:

– Сам подзапрос – подзапрос с «заглушками» для подстановки имен используемых отношений.
– Схема результата подзапроса – список полей, их параметров, а также перечень индексных полей, которые требуются для создания результирующего отношения в БД.
– Идентификатор запроса – уникальный идентификатор, позволяющий связывать результирующие отношения одних подзапросов с исходными отношениями для других.

– Перечень идентификаторов зависимых подзапросов – ссылка на используемые результаты из других подзапросов.

Идентификаторы запросов и перечни идентификаторов зависимых подзапросов позволяют однозначно определить зависимости между подзапросами и производить обработку строго по регулярному плану.

Процесс претрансляции разберем на примере запроса №3 из состава теста TPC-H, представленного в листинге 2.1.

Листинг 2.1. Запрос №3 из состава теста TPC-H

```
1.  SELECT
2.     L_ORDERKEY,
3.     SUM(L_EXTENDEDPRICE * (1 - L_DISCOUNT)) AS REVENUE,
4.     O_ORDERDATE,
5.     O_SHIPPRIORITY
6.  FROM
7.     CUSTOMER,
8.     ORDERS,
9.     LINEITEM
10. WHERE
11.    C_MKTSEGMENT = 'HOUSEHOLD'
12.    AND C_CUSTKEY = O_CUSTKEY
13.    AND L_ORDERKEY = O_ORDERKEY
14.    AND O_ORDERDATE < DATE '1995-03-31'
15.    AND L_SHIPDATE > DATE '1995-03-31'
16. GROUP BY
17.    L_ORDERKEY,
18.    O_ORDERDATE,
19.    O_SHIPPRIORITY
20. ORDER BY
21.    REVENUE DESC,
22.    O_ORDERDATE;
```

Для начала выделим отношения, которые участвуют в запросе:

CUSTOMER, ORDERS, LINEITEM.

Затем выделим используемые поля в каждом из отношений:

– CUSTOMER: C_CUSTKEY, C_MKTSEGMENT
– ORDERS: O_ORDERDATE, O_SHIPPRIORITY, O_ORDERKEY, O_CUSTKEY
– LINEITEM: L_ORDERKEY, L_EXTENDEDPRICE, L_DISCOUNT, L_SHIPDATE

Определим условия для формирования R’ по выделенным отношениям:

– CUSTOMER: C_MKTSEGMENT = 'HOUSEHOLD'
– ORDERS: O_ORDERDATE < DATE '1995-03-31'
– LINEITEM: L_SHIPDATE > DATE '1995-03-31'

Уберем из списка полей, те поля, которые используются только в условиях. Тогда в запросах *select-project* останутся поля:

- CUSTOMER: C_CUSTKEY
- ORDERS: O_ORDERDATE, O_SHIPPRIORITY, O_ORDERKEY, O_CUSTKEY
- LINEITEM: L_ORDERKEY, L_EXTENDEDPRICE, L_DISCOUNT

Составим запросы R’ для каждого из отношений. Заметим, что вычисление L_EXTENDEDPRICE * (1 - L_DISCOUNT) может быть выполнено на этапе *select-project*. Поэтому выполним его с формированием нового поля L_REVENUE. Полученные запросы отражены в листинге 2.2.

Листинг 2.2. Подзапросы select-project для запроса №3 теста TPCH-H

```
1.    -- CUSTOMER'
2.   SELECT
3.    C_CUSTKEY
4.   FROM
5.    CUSTOMER
6.   WHERE
7.    C_MKTSEGMENT = 'HOUSEHOLD';
8.
9.   -- ORDERS'
10.  SELECT
11.   O_ORDERDATE,
12.   O_SHIPPRIORITY,
13.   O_ORDERKEY,
14.   O_CUSTKEY
15.  FROM
16.   ORDERS
17.  WHERE
18.   O_ORDERDATE < DATE '1995-03-31';
19.
20.  -- LINEITEM'
21.  SELECT
22.   L_ORDERKEY,
23.   L_EXTENDEDPRICE * (1 - L_DISCOUNT) AS L_REVENUE
24.  FROM
25.   LINEITEM
26.  WHERE
27.   L_SHIPDATE > DATE '1995-03-31';
```

Чтобы загрузить полученные отношения в модуль JOIN, нужно знать их схему (перечень полей с указанием их типа и индексов). Сформировать ее можно автоматически, сопоставляя схемы исходных отношений с полями R’. Например, отношению CUSTOMER соответствует схема, показанная в листинге 2.3. На ее основе, с учетом полученного отношения R’, схема примет вид листинга 2.4, где PRIMARY KEY – ключевое поле, INDEX – индексное поле. Из листингов видно, что поле CUSTKEY и его индексация остались без изменений, а остальные поля были удалены.

Листинг 2.3. Схема отношения CUSTOMER из состава теста TPC-H

```
1.   C_CUSTKEY    INTEGER NOT NULL,
2.   C_NAME     VARCHAR(25) NOT NULL,
3.   C_ADDRESS    VARCHAR(40) NOT NULL,
```

```
4.   C_NATIONKEY  INTEGER NOT NULL,
5.   C_PHONE    CHAR(15) NOT NULL,
6.   C_ACCTBAL   DECIMAL(15,2)  NOT NULL,
7.   C_MKTSEGMENT CHAR(10) NOT NULL,
8.   C_COMMENT   VARCHAR(117) NOT NULL;
9.
10.  PRIMARY KEY (C_CUSTKEY),
11.  INDEX (C_NATIONKEY)
```

Листинг 2.4. Схема вновь полученного R’ на основе CUSTOMER

```
1.   C_CUSTKEY   INTEGER NOT NULL;
2.
3.   PRIMARY KEY (C_CUSTKEY);
```

В случае, если в схеме имеется новое поле (его нет в схеме исходного отношения), как в отношении `LINEITEM'`, то его тип будет соответствовать наиболее общему типу в выражении. Это позволяет избежать переполнений в случае, если в выражении используются разные типы данных.

Условия вида `C_CUSTKEY = O_CUSTKEY` являются операциями *join*. Таковых условий в запросе – 2. Соответственно будет выполнено два *join*: над двумя R’ – результатом *join* и оставшимся R’.

Подзапросы *join* будем формировать по порядку, начиная с условия `C_CUSTKEY = O_CUSTKEY`. Первое поле в условии содержится в `CUSTOMER'`, второе – в `ORDERS'`. Соответственно *join* будет выполнен между этими отношениями с указанным условием. Набор полей результата формируется объединением полей отношений, участвующих в *join*, с исключением полей, по которым эта операция выполняется, если они не требуются в результате исходного запроса. Полученный подзапрос *join* и его схема показаны в листинге 2.5, где `{LEFTRELATION}` и `{RIGHTRELATION}` – места подстановки («заглушки») названий отношений во время работы.

Листинг 2.5. Первый подзапрос *join* и его схема

```
1.    -- join 1
2.   SELECT
3.    O_ORDERDATE,
4.    O_SHIPPRIORITY,
5.    O_ORDERKEY
6.   FROM
7.     {LEFTRELATION} AS C,
8.     {RIGHTRELATION} AS O
9.   WHERE
10.    C_CUSTKEY = O_CUSTKEY;
11.
12.  -- схема результата
13.  O_ORDERDATE   DATE NOT NULL,
14.  O_ORDERPRIORITY CHAR(15) NOT NULL,
```

```
15.  O_ORDERKEY    INTEGER NOT NULL;
16.
17.  PRIMARY KEY (O_ORDERKEY);
```

Второй (последний) подзапрос *join* формируется для условия `L_ORDERKEY = O_ORDERKEY` на основе результата предыдущего *join*. По аналогии с предыдущим *join*, получим подзапрос со схемой (листинг 2.6).

Листинг 2.6. Второй подзапрос *join* и его схема

```
1.   -- join 2
2.   SELECT
3.    O_ORDERDATE,
4.    O_SHIPPRIORITY,
5.    L_ORDERKEY,
6.    L_REVENUE
7.   FROM
8.     {LEFTRELATION} AS J01,
9.     {RIGHTRELATION} AS L
10.  WHERE
11.   L_ORDERKEY = O_ORDERKEY;
12.
13.  -- схема результата
14.  O_ORDERDATE   DATE NOT NULL,
15.  O_ORDERPRIORITY CHAR(15) NOT NULL,
16.  L_ORDERKEY  INTEGER NOT NULL,
17.  L_REVENUE DECIMAL(15,2) NOT NULL;
18.
19.  INDEX (L_ORDERKEY);
```

Следующий шаг – составить подзапрос *sort* для получения итогового результата. Для этого необходимо выделить параметры из исходного запроса: функции агрегации, сортировки и группировки. В рассматриваемом запросе используется всего одна функция агрегации `SUM()`, которая применяется к выражению `L_EXTENDEDPRICE * (1 - L_DISCOUNT)`, вычисленному еще на этапе *select-project* и записанному в поле `L_REVENUE`. Сортировка производится по двум полям: сначала – по `REVENUE` с убыванием, а затем – по `O_ORDERDATE`. Поле `REVENUE` вычисляется с помощью применения функции `SUM()` к `L_REVENUE` с группировкой по `L_ORDERKEY`, `O_ORDERDATE` и `O_SHIPPRIORITY`. На основе полученной информации можно получить подзапрос *sort* и его схему (листинг 2.7). Индексы и ключи в схеме результата отсутствуют, т.к. получен результат для исходного запроса, и дальнейших операций над ним не предвидится.

Листинг 2.7. Подзапрос *sort* для запроса №3 теста TPC-H

```
1.   -- sort
2.   SELECT
3.    L_ORDERKEY,
4.    SUM(L_REVENUE) AS REVENUE,
```

```
5.    O_ORDERDATE,
6.    O_SHIPPRIORITY
7.   FROM
8.     {RELATIONNAME} AS J02
9.   GROUP BY
10.   L_ORDERKEY,
11.   O_ORDERDATE,
12.   O_SHIPPRIORITY
13.  ORDER BY
14.   REVENUE DESC,
15.   O_ORDERDATE;
16.
17.  -- схема результата
18.  L_ORDERKEY  INTEGER NOT NULL,
19.  REVENUE DECIMAL(15,2) NOT NULL,
20.  O_ORDERDATE   DATE NOT NULL,
21.  O_ORDERPRIORITY CHAR(15) NOT NULL;
```

Операция претрансляции была применена ко всем запросам без операций записи теста TPC-H и записана в специальном формате, который учитывает связи между отношениями. Так, всем отношениям был назначен уникальный идентификатор, который используется для выбора отношений на передачу и для генерации имен отношений в инструментальной СУБД.

Для рассмотренного запроса №3 соответствующее назначение идентификаторов можно выполнить следующим образом. Пусть результаты *select-project* получат идентификаторы:

- `CUSTOMER' - 9e8215df-6e79-42c7-bc4d-6e5b1de9b5e0`
- `ORDERS' - 99e76918-3fb3-411d-b9e5-e6bf1bce8b21`
- `LINEITEM' - 38a98c1e-21e8-4a0f-8f07-2227e5cf8322`

В этом случае первый *join* будет проведен над отношениями с идентификаторами `9e8215df-6e79-42c7-bc4d-6e5b1de9b5e0` и `99e76918-3fb3-411d-b9e5-e6bf1bce8b21`.

Назначим результату первого *join* идентификатор `7ae758d8-2679-47f2-92d0-4470f39c4cf9`. Тогда второй *join* будет выполнен над отношениями с идентификаторами `7ae758d8-2679-47f2-92d0-4470f39c4cf9` и `38a98c1e-21e8-4a0f-8f07-2227e5cf8322`.

Аналогично для *sort*: если для результата второго *join* назначить идентификатор `823ed8b9-978f-4f6f-9138-6002ce78b705`, он будет использован для идентификации исходного отношения в операции *sort*, результату которой также назначается идентификатор, например, `c368ecde-7dd2-4726-908b-d63bfd2a96d8`.

2.3. Разработанный метод настройки MySQL

Clusterix-N предполагает обработку отношений параллельно по ядрам. Однако СУБД MySQL [71] с настройками по умолчанию в таком режиме работает чрезвычайно медленно. Поэтому требуется ее специальная настройка, которая позволит эффективно обрабатывать запросы к БД консервативного типа [17].

На производительность СУБД влияет множество факторов [72]: производительность «железа» (процессор, память, сеть, дисковая подсистема и т.д.), настройка ОС, ее разрядность (32 или 64 бита), настройка СУБД, оптимизация запросов, наличие индексов, блокировки доступа и др. При условии, что «железо» менять нельзя, из всех этих факторов администратор БД может повлиять только на настройку ОС, настройку СУБД, схему БД (индексы, связи, отношения) и запросы. А в случае работы со стороннем сервисом или тестирования производительности, например, с TPC-H [11] администратор может повлиять только на 2 фактора: настройку ОС и настройку СУБД.

Современные ОС как правило хорошо работают в настройке по умолчанию, поэтому сосредоточимся на настройке СУБД. Узким местом в любой вычислительной системе обычно является внешняя память (сетевые хранилища, дисковая память и др.), поэтому было бы очень хорошо не обращаться к внешней памяти во время работы, как это сделано в In-Memory Database VoltDB [33] или Spark [69]. Следующей проблемой являются блокировки таблиц. Движок в MySQL по умолчанию MyISAM [73] во время выполнения запроса производит блокировку таблиц целиком, что плохо сказывается на параллельной обработке. Другой движок InnoDB [74] выполняет блокировку построчно. Такая организация блокировок хороша для параллельной обработки, но может привести к неожиданному падению производительности во время операций записи [75]. Консервативные БД подразумевает работу только на чтение, поэтому в качестве движка будем использовать InnoDB.

InnoDB помимо блокировки на уровне строк обладает еще одной важной особенностью. Этот движок способен работать с минимальным обращением к внешней памяти. Достигается это установкой параметра `innodb-buffer-pool-size`, который отвечает за размер буфера в оперативной памяти для хранения исходных от-

ношений, значения объема отводимой под буфер RAM (обычно 80% доступной оперативной памяти узла [76]) и загрузкой БД в этот буфер при запуске СУБД.

С учетом возможных путей увеличения производительности сформирована конфигурация MySQL, представленная в листинге Б.1. Описание всех параметров предоставлено в [77]. Приведенная конфигурация MySQL записывается в специальный файл [78].

Настройка MySQL для JOIN. Конфигурация в Б.1 пригодна только для операций чтения данных. В то же время операция *join* требует выполнения загрузки данных промежуточных/временных отношений. Загрузка данных – длительный процесс. Его ускорения можно добиться, исключив работу с диском. Для этого достаточно использовать движок `MEMORY` вместо `InnoDB` и переопределить параметры конфигурации для оптимизации работы *join*, как показано в листинге 2.8, где [78]:

- `tmp-table-size` – максимальный размер временных таблиц;
- `max-heap-table-size` – максимальный размер данных, размещаемых в оперативной памяти;
- `join_buffer_size` – размер буфера для операций *join*;
- `bulk_insert_buffer_size` – размер буфера загрузки и вставки данных существующие отношения;
- `sort_buffer_size` – размер буфера для операций *sort*.

Листинг 2.8. Изменения в настройках MySQL для JOIN

```
1. tmp-table-size           = 5G
2. max-heap-table-size      = 100G
3. join_buffer_size         = 1G
4. bulk_insert_buffer_size  = 512M
5. sort_buffer_size         = 512M
6. innodb-buffer-pool-size  = 1G
```

Настройка MySQL для SORT. Конфигурация выполнена аналогично настройке для JOIN со следующими изменениями: используется движок `MyISAM`, как обеспечивающий лучшую производительность в однопоточном режиме и позволяющий экономить оперативную память узла; параметр `max-heap-table-size = 1G`.

Все предложенные конфигурации MySQL используются соответствующими модулями во всех приведенных экспериментах.

2.4. Принятая методология

За основу исследований принимается методология КМС – конструктивного моделирования систем [79, 80].

Особенности методологии КМС. Эта методология трактует модель системы как конструктивный метод, т.е. как модель процесса ее синтеза – ***S***-модель (***S*** – от Synthesis).

«*В условиях неполноты информации* процесс синтеза целесообразно рассматривать с системных позиций в предположении, что синтезируемый объект моделирует поведение некоторой гипотетической системы. Система есть нечто единое целое, бесконечно познаваемое и бесконечно объясняемое, заданное своим оператором назначения.

Под процессом в кибернетике понимается последовательная смена состояний некоторого объекта. В условиях неопределенности процесс синтеза – это еще и процесс познания неизвестного. Поэтому ***S***-модель рассматривается не как статичное образование, а как динамически развивающаяся (*эволюционирующая*) система, каждому состоянию которой отвечает определенное качество моделирования. Ее развитие прекращается по получении заданного качества.» [79, 80]

Кардинальные вопросы синтеза в условиях неполноты информации:

– ГДЕ (в какой области некоторого пространства) искать нужное решение?
– КАК (какими методами) организовать такой поиск?
– ПОЧЕМУ именно там и так?

«Методологическую основу КМС составляют 3 положения:

1. Считается (как отмечено ранее), что синтезируемый объект моделирует поведение некоторой гипотетической системы, заданной оператором своего назначения.

2. Полагается необходимой декларация постулатов для обоснования, в меру накопленных знаний адекватности найденного метода решаемой задачи.

3. Признается бесконечность систем объяснений и их неизбежная незавершенность, т.е. открытость для дальнейших объяснений [81].» [79, 80]

Математическая модель входит в состав ***S***-модели как релевантное описание – фреймовое, логическое, алгебраическое или др. – подмножеств (областей) всевозможных решений (ответ на вопрос: ГДЕ?), на которых теми или иными методами/алгоритмами (ответ на вопрос: КАК?) ищется решение задачи синтеза с за-

данным качеством. Конечной целью ***S***-моделирования (построения ***S***-модели) является разработка теоретически обоснованного (ответ на вопрос: ПОЧЕМУ?) конструктивного метода, т.е. процедуры синтеза. Эта процедура формируется путем модельных исследований, в ходе которых оптимизируется значения параметров в области, устанавливаются оценки достижимого качества на выделенном подмножестве состояний модели.

«Постулирование свойств подмножества эффективных реализаций системы как основа объяснительной теории и предпосылка разработки конструктивного метода является характерной особенностью ***S***-модели. Постулаты в точных науках – явление не совсем обычное. Но процесс ***S***-моделирования связан с познанием своеобразной «вселенной», с приближением к некоторой «абсолютной истине». В таком процессе правомерно использование общей методологии естественных наук (прежде всего физики), которая базируется на введении постулатов.

Обычно требуемый детерминизм достигается при постановке задачи введении разного рода определений и ограничений. Постулаты делают то же самое, но декларируют это как закономерности, выявленные (что характерно) в самом процессе ***S***-моделирования с использованием имеющегося мирового опыта и эвристики, т.е. нестрого индуктивно. В силу последнего система постулатов как элементов объяснительной теории должна быть открытой для корректив.

Процесс ***S***-моделирования (рис. 2.1) рассматривается как многошаговый итеративный процесс, в котором взаимодополнительно проявляются как объяснительные посылки (теория), так и сам конструктивный метод (модель). В этом процессе и формируются постулаты как элементы содержательной (неформальной) теории. Модель и теория становятся столь трудно разделимы, что их противопоставление теряет смысл. Это характерно для системотехники в целом [82]. Здесь наиболее правомерно интегрированное понятие модели-теории. Процесс ее построения является творческим, т.е. неформальным в своей основе. Эвристика играет решающую роль.» [79, 80]

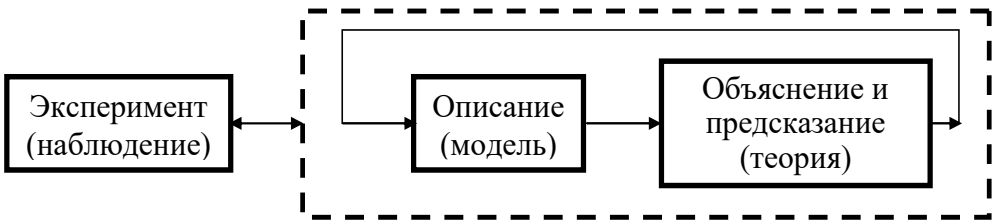


Рис. 2.1. Иллюстрация процесса S-моделирования

Подчеркнем важный момент: система постулатов, как это следует из рис. 2.1, итеративно формируется в самом процессе ***S****-моделирования* (а не предшествует ему) с использованием мирового опыта и результатов натурных и умозрительных экспериментов, проводимых при поиске конструктивного метода. И в то же время постулаты – основа формируемой содержательной теории, дают обоснование найденного метода. Итеративность процесса это объясняет.

«В основе конструктивизма – симбиоз, точнее – взаимодополнительность теории и эксперимента [83]. Но до середины 80-х годов прошлого века господствовала парадигма первенства теории над экспериментом. С этим никогда не были согласны передовые ученые. Они полагали, что натурный, вычислительный либо умозрительный эксперимент – источник открытий. Недостаток информации восполняется открытиями по результатам эксперимента, которые не поддаются строгому доказательству. Надо декларировать эти открытия как конструктивные закономерности и брать их за основу развития перспективных теорий с серьезными приложениями. Близкую точку зрения имели известные конструктивисты прошлого века академики Н.Н. Семенов (творец теории цепных реакций) и П.К. Анохин (творец конструктивной теории функциональных систем [84]).

Введение постулатов – шаг весьма ответственный. Он целесообразен только если разработка конструктивного метода на их основе показывает его перспективность для своего времени, а сам метод не укладывается в рамки существующей теории. Качества оценок адекватности конструктивных методов, развитых с обычных позиций определений-ограничений и с позиций постулатов, существенно различны. Оценка в первом случае достаточно аморфна: *можно и так*. Во втором – категорична в меру накопленных знаний: *надо только так!* Это далеко не одно и то же.» [79, 80]

Большие системы, как правило, иерархические – ***IS***-модели. «Особенность ***IS***-моделей в том, что каждый *i*-уровень иерархии представлен некоторой ***S***-моделью $M^i = \{M^i_j\}$, где M^i_j – состояние *j* модели M^i , *j*=1, 2, … Значение $i \in \{\overline{1,N}\}$, где N – число уровней. Задача ***IS***-моделирования состоит в определении полного k-состояния $M_k =< M^1_{j(1)},..., M^N_{j(N)} >_k$ модели $M = \{M_k\}$, *k*=1, 2, … , из условия эффективного симбиоза представлений $M^i_{j(i)}$ на всех *N* уровнях для заданного качества моделирования. Здесь уже *j(i)* – фиксированный для каждого *i* номер состояния мо-

дели M^i. Эта задача решается поиском в *N*-мерном дискретном пространстве J_N элементов $< j(1),..., j(N) >$ при известном конечном множестве состояний моделей любого уровня.

Использование градиентных методов поиска решающего вектора $< j(1),..., j(N) >_k$ осложнено трудностями генерации программных кросс-систем с изменяющимися параметрами для получения оценок качества в динамике моделирования. Однако необходимое число шагов поиска, а потому и требуемый объем работ в данном случае можно значительно снизить. Действительно, никакой поиск не проводится «на пустом месте». Поэтому начальное полное состояние M_1 модели M заведомо определено тем или иным способом. Идентифицируем это состояние единичным вектором $< 1,...,1 >$. Системная парадигма гармонического развития ***IS***-модели требует вести поиск таким образом, чтобы переход от *j* к *j*+1 происходил одновременно по всем координатам пространства J_N, т.е. искать решение в виде k-вектора <k, …, k>.

При этом совсем не обязательно $M^i_{j+1} \neq M^i_j$ для всех *i*, т.е. некоторые соседние элементы последовательностей $\{M^i_j\}$, *j* = 1, 2, …, могут повторяться. В переходе при данном *j* фактически должны участвовать только те уровни, изменение состояний которых дает заметное улучшение качества моделирования без чрезмерного роста стоимости устройства. Обычно они выявляются умозрительно без особых затруднений. Такой поиск «вширь» завершается достаточно быстро, как только достигнутое качество станет не меньше требуемого.» [79, 80]

Интерпретация КМС применительно к процессу синтеза консервативных СУБД. К числу ***IS***-моделей относятся и СУБД с уровнями иерархии: *select-project, join, sort*, динамическая сегментация отношений, их индексация, сетевой и др. В данном случае оператор назначения гипотетической системы задан условием получения высокой эффективности обработки запросов при минимальной стоимости системы, определенной ранее сформулированными ограничениями. Состояние ***IS***-модели отображает архитектура программной системы как совокупность взаимодействующих программных модулей. Название состояния будем ассоциировать с некоторым характерным признаком, а полную программную разработку будем называть *полным состоянием*.

Процесс ***IS***-моделирования не должен занимать слишком много времени, как это свойственно естественной эволюции. Поэтому в таком процессе должен быть найден минимальный набор состояний (областей) в пространстве всевозможных состояний модели, переходы между которыми образуют *кратчайший путь* получения искомого результата. Такой набор определяется пресловутой *математической моделью* (*внешняя модель*), которая строится эвристически в самом процессе ***IS***-моделирования. Алгоритмическая и программная разработка каждого состояния является предметом *внутреннего моделирования*.

От пространства полных состояний можно перейти к пространству параметров. Под параметром будем понимать среднее время обработки одного запроса ПТ на том или ином уровне. Для заданной платформы существует однозначное отображение пространства полных состояний в пространство параметров (вопрос о взаимной однозначности оставляем открытым), в котором и будем вести рассмотрение. По аналогии с принятым в синергетике [85], для каждого полного состояния будем выделять так называемый «*параметр порядка*», минимизация влияния которого на производительность системы и определит переходы между состояниями-итерациями.

В качестве такового принимается параметр, имеющий максимальное значение для данного полного состояния. Но функционирование всех уровней в большой системе взаимосвязано (принцип единства системы [86, 87]). Поэтому снижение влияния «*параметров порядка*» на производительность системы неизбежно влечет изменение влияния и других уровней. Число итераций обычно сравнительно невелико, если критерий качества конечного решения приемлемо задан (в данном случае – получение эффективности, сравнимой с эффективностью системы Spark). Начальное состояние ***IS***-модели определяется из тех или иных соображений.

2.5. Принятые постулаты

В процессе проведенного ***IS***-*моделирования* сформулирована система постулатов как декларация целесообразных направлений разработок искомых моделей.

ПОСТУЛАТ 1. Решение поставленной задачи должна обеспечить эволюция Clusterix-подобных СУБД от начальной реализации принципов гибридной технологии (см. ниже принятое начальное состояние ***IS***-модели).

ПОСТУЛАТ 2. Поиск очередных состояний (итераций) ***IS***-модели Clusterix-N следует вести на пути замены стратегии «ядро на одно отношение», принятой для ее начального состояния, на стратегию «группа узлов (ядер) на одно отношение». Это необходимо для обеспечения надежной работы эффективной системы при значительных объемах баз данных и требует динамической сегментации отношений, которая может быть, как сосредоточенной, так и распределенной.

ПОСТУЛАТ 3. Внутреннее ***IS***-моделирование Clusterix-подобных систем следует проводить в направлениях, определенных внешней (математической) фреймовой моделью процесса синтеза, показанной на рис. 2.2.

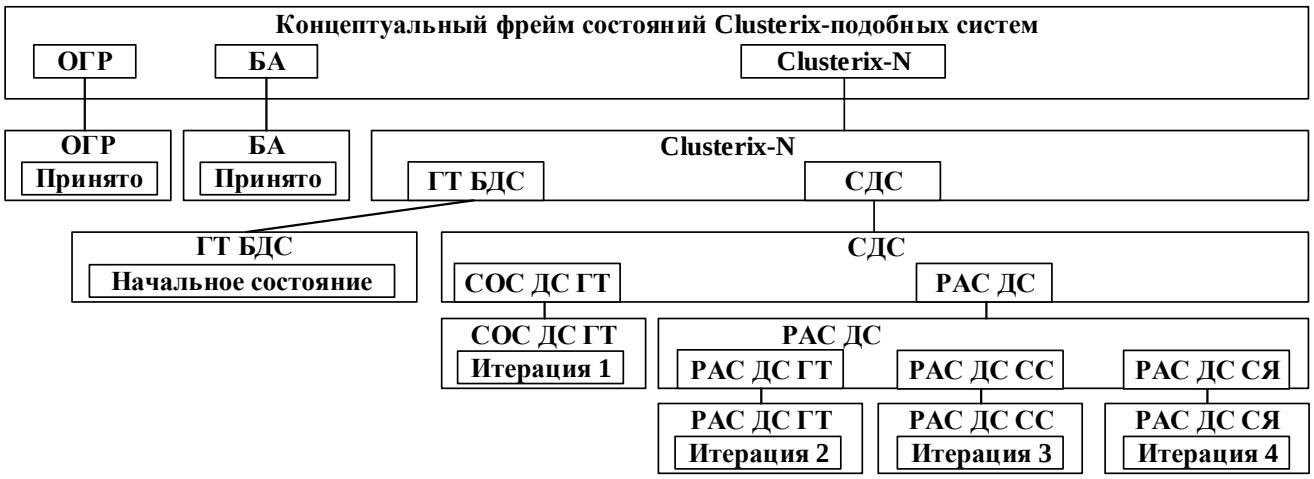


Рис. 2.2. Внешняя фреймовая модель процессов синтеза Clusterix-подобных систем

Это и есть пресловутая математическая модель. Здесь (суть использованных понятий пояснена в соответствующих разделах):

ОГР – фрейм принятых ограничений.

БА – фрейм принятой блочной архитектуры в составе 5 программных блоков: IO (модуль доступа к данным БД, выполняет подзапросы типа *select-project*), JOIN (модуль обработки подзапросов типа *join*), Mgm (модуль управления), SORT (модуль конечной обработки запроса), HASH (реализует динамическую сегментацию отношений; в начальном состоянии он отсутствует).

Clusterix-N (N – от New) – фрейм развития Clusterix-подобных систем.

ГТ БДС – фрейм начального состояния ***IS***-модели (переход к гибридной технологии Clusterix-подобных систем без динамической сегментации отношений).

СДС – фрейм Clusterix-N с динамической сегментацией отношений.

РАС ДС – фрейм СДС-систем с распределенной динамической сегментацией.

СОС ДС ГТ – фрейм СДС-систем с сосредоточенной динамической сегментацией отношений в рамках гибридной технологии (первая итерация ***IS***-моделирования).

РАС ДС ГТ – фрейм (РАС ДС)-систем, реализованных по гибридной технологии (вторая итерация ***IS***-моделирования).

РАС ДС СС – фрейм (РАС ДС)-систем в конфигурации «совмещенная симметрия» (третья итерация ***IS***-моделирования).

РАС ДС СЯ – фрейм (РАС ДС)-систем в конфигурации «совмещенное ядро» (четвертая итерация ***IS***-моделирования).

2.6. Выбор и разработка начального состояния *IS*-модели

С учетом требования упрощения разработки СУБД, в работе [88] была выдвинута гипотеза*: для параллельных СУБД консервативного типа и значительных объемов баз данных, регулярный план обработки запросов предпочтителен для повышения эффективности пакетной обработки запросов за счет сравнительно малотрудоемкой статической конвейеризации*. Но первые разработки СУБД, использующие этот план (СУБД Clusterix [16] и Clusterix-M [18]) были недостаточно производительны.

В работе [89] было показано, что поправить положение можно переходом к архитектуре Clusterix-N (N – от New) путем отказа от принципа «однородности» (характерного для конфигурации «совмещенная симметрия» [18]) в пользу «гибридности», подразумевающей разделение кластера на две различные части – блоки IO и JOIN – с независимой вариацией числа узлов в каждом блоке. База данных хешировалась на уровне узлов IO. На уровне узлов JOIN (как и в узлах IO) использовалась стратегия «запрос на ядро» (реализуемость такой стратегии средствами MySQL доказана в работе [17]), что позволяло исключить динамическое сегментирование промежуточных и временных отношений и выполнять соединение в виде единой процедуры R1′ *join* (*join* R2′(*join* R3′(…))) …) по каждому запросу. Это значительно быстрее ее последовательного выполнения и, как показано далее, если суммарный объем оперативной памяти блока IO достаточен для распределенного размещения в нем базы данных, приводит к значительному повышению эффективности по сравнению с СУБД Clusterix-M.

Архитектура СУБД выбранного начального состояния IS-модели показана на рис. 2.3. Здесь:

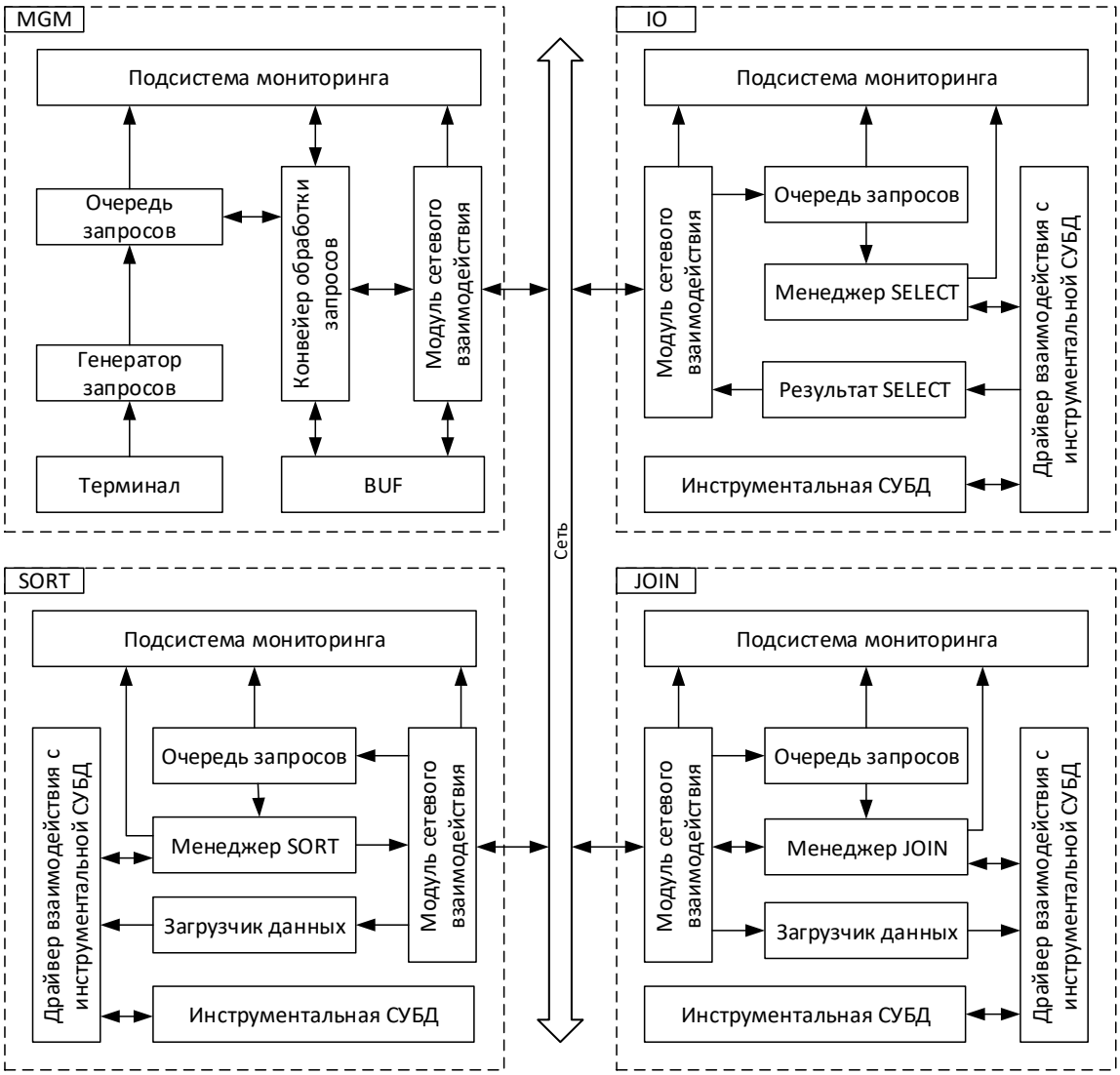


Рис. 2.3. Архитектура начального состояния *IS*-модели

- MGM – программный модуль управления, занимающийся передачей подзапросов (*select-project, join* или *sort*) и данных в соответствующие программные модули.
- IO – программный модуль доступа к данным БД, выполняет запросы типа *select-project*, преобразует результат и передает его в MGM для дальнейшей обработки
- JOIN – программный модуль обработки запросов типа *join*. Получает данные и запрос от MGM, выполняет его и передает результат обратно в MGM.
- SORT – программный модуль конечной обработки запроса. Выполняет операции агрегации и сортировки над данными, полученными от MGM.

Сопоставление типичной архитектуре реляционных СУБД (рис. 1.1) с архитектурой Clusterix-N (рис. 2.3) показывает, что архитектура Clusterix-N в целом отвечает типичной архитектуре реляционных СУБД:

1. Менеджер коммуникации с клиентом в Cluserix-N реализуется MGM с использованием модуля сетевого взаимодействия.

2. Менеджер управления процессами реализуется во всех модулях системы с использованием следующих компонентов: очередь запросов; конвейер обработки запросов; менеджерами подзапросов SELECT, JOIN, SORT.

3. Менеджер управления транзакциями реализуется модулем MGM с использованием компонентов очереди запросов и BUF.

4. Общие компоненты и утилиты в Clustrix-N представлены в виде общих модулей: подсистемы журналирования, сбора статистики, сетевого взаимодействия, драйвер СУБД и т.д.

5. Процессор реляционных запросов в Clusterix-N ограничено реализован в драйвере СУБД, который позволяет разобрать запрос и выполнить его перезапись в нужном формате, но сама реляционная обработка отдана инструментальной СУБД (MySQL).

Все модули разработаны на языке C# для платформы .Net 4.0 с возможностью сборки и запуска на платформе Mono 3.10 [90] и новее. Разработанные сервисные средства – подсистема сбора статистики и визуализации (в составе подсистемы мониторинга), подсистема журналирования, модуль сетевого взаимодействия, драйвер СУБД – являются достаточно общими (при их возможной перекомпановке и некоторых изменениях) для всех представленных в работе итераций. Они рассмотрены в Приложении А.

Начальное состояние ***IS***-модели предполагает запуск разных программных модулей на отдельных узлах. Обработка запроса в системе выполняется по схеме *select-project-join*. Каждый программный модуль обрабатывает свою часть запроса: IO – *select-project,* JOIN – *join* и SORT – *sort*. В модуле IO обработка ведется по стратегии «ядро на отношение». При этом, если после запуска очередного запроса остались свободные ядра, то они занимаются отношениями следующего запроса. В модуле JOIN операции загрузки данных выполняются по той же стратегии. Но поскольку операция *join* «интегрирована», то она выполнятся на одном ядре. Модуль SORT, как правило, обрабатывает одно отношение на каждый запрос. Поэтому реализует стратегию «ядро на запрос».

Связь между программными модулями осуществляется по сети. Передача данных производится пакетами вариативной длины. За корректность сетевых передач отвечает специальный программный модуль, который используется всеми программными модулями для сетевого взаимодействия. Взаимодействие с инструментальной СУБД MySQL организуется через специальный драйвер, который выполняет трансляцию запросов и данных из формата Clusterix-N в формат, требуемый СУБД.

Работа системы начинается с передачи команды на запуск генератора запросов из терминала. Генератор начинает передать запросы из перечня заранее претранслированных 14 запросов теста TPC-H без операций записи в очередь MGM, где каждому новому запросу устанавливается статус ожидания. Как только MGM обнаруживает, что все подключенные IO способны принять очередной подзапрос *select-project*, он отправляет его и устанавливает статус запроса – *select*. Операция отправки запроса повторяется до тех пор, пока не будут отосланы все подзапросы *select-project* для данного запроса или пока не будут загружены все ядра IO. Если все подзапросы для данного запроса отправлены, а узлы IO загружены не полностью (есть свободные ядра), то отправляются подзапросы следующего запроса. Результат работы узлов IO поступает в BUF MGM в виде блоков данных. Эти блоки представляют собой набор строк из результата выполнения подзапроса *select-project*. Блоки накапливаются в буфере MGM по отношениям. Когда все блоки одного из отношений получены, они передаются в узел JOIN.

Балансировка нагрузки между узлами и ядрами JOIN осуществляется по алгоритму *round-robin* (круговой алгоритм) [91]. При этом сначала в выбранный JOIN загружаются все отношения одного запроса. После чего MGM инициирует в нем операцию *join*. Затем «по кругу» аналогичным образом загружаются незанятые ядра всех узлов. Если заняты все ядра текущего JOIN, то выполняется переход к следующему. Аналогично происходит работа с процессором SORT: результат выполнения подзапроса *join* поступает BUF MGM и отправляется для дальнейшей обработки в SORT.

Предложенный выбор начального состояния ***IS***-модели (его архитектуры) позволяет подключать к системе различные инструментальные СУБД в специальных конфигурациях без существенного изменения системы (требуется только разработать драйвер СУБД). Терминал управления в модуле MGM позволяет управлять системой, а генератор запросов – выполнять тестирование производительно-

сти с различной нагрузкой. Динамический мониторинг свободной оперативной памяти позволяет системе избегать ее переполнения, останавливая узлы с модулем IO и прекращая передачу данных в узлы с модулями JOIN.

Разделение программных модулей по этапам обработки запроса позволяет использовать разные стратегии обработки в каждом отдельно взятом модуле. Это может быть полезно при настройке системы на обработку специфичной нагрузки. Подсистема мониторинга позволяет оценить эффективность работы системы. Отчет по работе системы предоставляет информацию о распределении работ по этапам обработки запроса.

Модуль MGM – ядро системы. Управляет выполнением запросов, передачей данных, ведением очереди запросов и сбором данных мониторинга. Включает в себя модули:

- Терминал – средство «общения» MGM с пользователем/администратором.
- Генератор запросов – модуль, содержащий набор заранее претранслированных запросов и методы генерации потока запросов.
- Очередь запросов – подсистема ведения очередей по каждому этапу обработки запроса.
- Конвейер обработки запросов – подсистема поэтапной обработки запроса.
- BUF – буфер данных для промежуточных/временных отношений.
- Модуль сетевого взаимодействия и подсистема мониторинга.

Рассмотрим каждый из них, кроме модуля сетевого взаимодействия и подсистемы мониторинга, которые были рассмотрены ранее.

ТЕРМИНАЛ. Предоставляет текстовый командный интерфейс управления MGM. Текстовый режим выбран в силу его универсальности и простоты реализации. Команды кодируются следующим образом: `name -key value`, где `name` – имя команды, `-key` – ключ параметра команды, `value` – значение параметра команды. Например, команда запуска потока запросов из 3 перестановок теста TPC-H с помещением в очередь до 7 запросов будет выглядеть так: `tpchseq -c 3 -l 7`.

В модуле терминала по умолчанию реализовано 2 команды:

1. `exit` – завершение работы приложения

2. `help` – вывод справки по всем доступным командам с указанием всех параметров. Пример вывода команды (вместе с реализованными для MGM командами) представлен в листинге Б.4.

Остальные команды реализуются и регистрируются в самих программных модулях. Регистрация команд позволяет унифицировать их обработку и организовать вывод команды `help`.

ГЕНЕРАТОР ЗАПРОСОВ. Предоставляет возможность генерирования последовательностей запросов для экспериментальных исследований. Источником запросов для генерации служит реализация интерфейса `IQuerySourceManager`, который требует содержит метод получения запроса по его номеру `Query GetQueryByNumber(int number)`, где `Query` – полученный запрос, `number` – номер запроса. Генератор номеров запросов реализуется интерфейсом `IQueryNumberGenerator` с методом получения нового номера запроса `int GetNextQueryNumber()`. Оба интерфейса передаются в генератор с установкой максимального числа сгенерированных запросов. Соответственно выбор очередного запроса из источника запросов осуществляется по полученному номеру от генератора номеров запросов. Генерация запросов прекращается при достижении максимального числа сгенерированных запросов.

В MGM реализовано 2 генератора номеров запросов. Первый генератор номеров запросов выдает номер очередного запроса случайным образом по равномерному закону, второй – выдает номера запросов последовательно из рекомендуемых перестановок запросов в тесте TPC-H.

Источников запросов реализовано также два. Первый предоставляет претранслированные запросы с интегрированным *join*, второй – с последовательным *join*.

ОЧЕРЕДЬ ЗАПРОСОВ. Организует хранение запросов и их состояний, отражает текущее состояние системы (ведает информацией о том, какой запрос на каком этапе обработки находится). Запрос находится в очереди вплоть до завершения его выполнения, а затем вытесняется из нее. Каждое состояние ассоциируется с этапом обработки запроса, его подзапросов и отношений. Всего предлагается 7 статусов для запроса, 15 статусов для подзапросов и 8 статусов для отношений. Каждый статус определят текущую операцию, производимую над запросом, подзапро-

сом или отношением. Их комбинация однозначно определяет статус каждого запроса, подзапроса и отношения.

Статусы прикрепляются к каждому запросу и показывают его глобальное состояние. Всего используется 7 следующих статусов:

1. `Wait` – ожидание выполнения, запрос находится в очереди, но никакие операции с ним еще не производились.
2. `Select` – выполняется этап выполнения подзапросов *select-project*.
3. `SelectComplete` – завершен этап выполнения подзапросов *select-project*.
4. `Join` – выполняется этап выполнения подзапросов *join*.
5. `JoinComplete` – завершен этап выполнения подзапросов *join*.
6. `Sort` – выполняется этап выполнения подзапросов *sort*.
7. `SortComplete` – завершен этап выполнения подзапросов *sort*.

Статусы подзапросов не зависят от статусов запросов, т.к. этапы выполнения запроса могут накладываться друг на друга во времени. Например, во время выполнения подзапросов *select-project* состояние запроса соответствует `Select`, но если по этому запросу началась обработка подзапросов *join*, то статус запроса не изменится на `Join` пока не будут завершены все операции *select-project*. Каждый статус подзапроса соответствует своему типу подзапроса, т.е. у подзапроса *select-project* не может быть статуса `TransferToSort`. Статусы подзапросов могут принимать следующие значения:

1. `Wait` – ожидание, никаких операций над подзапросом еще не производилось (подзапрос ожидает в очереди).
2. `SelectProcessing` – подзапрос *select-project* выполняется.
3. `TransferSelectResult` – идет передача результата *select-project*.
4. `SelectProcessed` – подзапрос *select-project* выполнен.
5. `TransferToJoin` – передача данных, необходимых для выполнения подзапроса *join*, в узел JOIN.
6. `JoinProcessing` – выполнение подзапроса *join*.
7. `JoinProcessed` – подзапрос *join* выполнен.
8. `TransferJoinResult` – передача результата последнего подзапроса *join*.
9. `JoinResultTransfered` – передача результата подзапроса *join* завершена.

10. `TransferToSort` – передача данных, необходимых для выполнения подзапроса *sort*, в узел SORT.

11. `TransferedToSort` – передача данных в SORT завершена.

12. `ProcessingSort` – выполнение подзапроса *sort*.

13. `SortProcessed` – подзапрос *sort* выполнен.

14. `TransferSortResult` – передача результата подзапроса *sort*.

15. `SortResultTransfered` – передача результата подзапроса *sort* завершена.

Подзапросы *join* и *sort* требуют передачи отношений для обработки в соответствующие узлы. Текущее состояние отношений отражается статусами:

1. `Wait` – ожидание, никаких операций еще не производилось.
2. `Preparing` – подготовка отношения.
3. `Prepared` – отношение подготовлено.
4. `TransferData` – передача данных для отношения.
5. `WaitAnotherRelation` – ожидание завершения операций с другим отношением. Например, ожидание результата предыдущего *join*.
6. `Transfered` – передача данных для отношения завершена.
7. `Processing` – обработка отношения: загрузка в БД, преобразования данных и т.д.
8. `Processed` – обработка отношения завершена, оно готово для последующих операций.

Изменение всех состояний подзапросов и отношений инициируется конвейером обработки запросов. Модуль очереди запросов не просто изменяет статус запроса, но и выполняет некоторую предобработку. Так, например, если по какой-то причине статус подзапроса или отношения будет понижен (например, у отношения был статус `Prepared`, а устанавливается статус `Preparing`), то статус изменен не будет, а в журнал добавится запись с ошибкой изменения статуса запроса.

Статусные состояния запроса в целом управляются самой очередью. Например, если все подзапросы *select-project* перешли в статус `SelectProcessed`, то статус запроса устанавливается в `SelectComplete`. А если текущий статус установлен как `SelectComplete` и есть подзапросы *join* со статусом, отличным от `Wait`, то статус изменяется на `Join`. И т.д.

Помимо учета статусов, очередь запросов производит подсчет времени нахождения каждого подзапроса в каждом статусе и общее время выполнения запроса с учетом времени ожидания в очереди. Результаты подсчета записываются в файл журнала времени выполнения запросов (в конфигурации по умолчанию `times_manager.log`) после завершения обработки запроса в целом. Отрывок из журнала времени выполнения запросов показан в листинге Б.6, где каждый новый `id` соответствует новому подзапросу.

Идентификация запроса, его подзапросов и отношений производится назначением каждому новому запросу, его подзапросам и отношениям, уникальных идентификаторов (`id`) GUID (Globally Unique Identifier) [92] – статистически уникальных 128-битных идентификаторов, являющихся некоторой реализацией стандарта, имеющего название Universally Unique Identifier (UUID) [93]. В тексте GUID записывается в виде строки из тридцати двух шестнадцатеричных цифр, разбитой на группы дефисами: `865fda1c-db06-459a-8a43-ba4f39b11d2a`.

Уникальные идентификаторы позволяют однозначно определять, на каком узле какой запрос/подзапрос/отношение обрабатывается и однозначно определить его статус в очереди. При этом для изменения статуса отношения в очереди не требуется сначала найти запрос, потом в нем найти нужный подзапрос и в подзапросе найти нужное отношение. Достаточно лишь выполнить поиск по идентификатору отношения.

КОНВЕЙЕР ОБРАБОТКИ ЗАПРОСОВ. Позволяет выполнять требуемые операции над запросом последовательно (поэтапно) и совмещать обработку сразу нескольких запросов в разных этапах. Реализация конвейера выполнена в виде последовательности вызовов специальных объектов – обработчиков стадий. Каждый обработчик реализует интерфейс стадии `IPipelineNode` с единственным методом `void DoAction()` и является наследником абстрактного класса `HandlerBase`, представленного в листинге Б.5. Где:

- `Server` – модуль сетевого взаимодействия (обработчики стадий могут передавать и принимать пакеты).
- `QueryManager` – очередь запросов с информацией об их статусах (статусы активно используются обработчиками).
- `NodesManager` – модуль управления подключенными узлами. Используется для выбора и передачи данных в конкретный узел.

- `PauseLogManager` – модуль журналирования остановок системы. Используется в случае нехватки оперативной памяти на узлах.
- `QueryBufferManager` – модуль BUF. Обеспечивает хранение временных/промежуточных отношений и предоставляет обработчикам возможность передавать данные отношений требуемым узлам.
- `QueueManager` – модуль асинхронной очереди. Длительные операции, такие как передача блоков по сети, помещаются в асинхронную очередь, чтобы не задерживать конвейер.
- `Logger` – модуль журналирования.

Такое обилие модулей, предоставляемых для обработчиков этапов конвейера, связано с тем, что каждая стадия самостоятельна. Она сама определяет, какие пакеты принимать и отправлять; сама решает, на какие статусы в очереди запросов реагировать; сама обеспечивает передачу данных в нужный узел. Это дает большую универсальность построения конвейера и упрощает разработку.

Настройка конвейера производится регистрацией обработчиков стадий в порядке их работы с запросом: `_pipeline.RegisterPipelineNode<Select>()`, где `_pipeline` – конвейер, `Select` – тип обработчика этапа. Все зарегистрированные стадии вызываются друг за другом и выполняют действия в соответствии со своим назначением. Как только была завершена работа последней стадии, процесс повторяется с начала.

В Clusterix-N в конвейере реализовано 11 стадий:

1. Получение промежуточных отношений R_i – выполнение подзапросов *select-project* и передача результата в BUF MGM.
2. Подготовка операций join – создание необходимых отношений в узлах JOIN.
3. Передача данных отношений *join* – передача готовым узлам JOIN данных для загрузки в подготовленные отношения.
4. Запуск операций *join* – передача команды на запуск *join* готовым узлам.
5. Передача результата *join* в BUF MGM – передача требования отправки результата *join* всем узлам, завершим эту операцию.
6. Подготовка операций sort – создание необходимых отношений в узлах SORT.
7. Передача данных отношений *sort* – передача готовым узлам SORT данных для загрузки в подготовленные отношения.

8. Запуск операций *sort* – передача команды на запуск *sort* готовым узлам.

9. Передача результата *sort* в BUF MGM – передача требования отправки результата *sort* всем узлам, завершившим эту операцию.

10. Проверка времени работы выполняющихся запросов на каждой стадии и удаление запросов, если время их выполнения превысило лимит.

11. Очистка – запись времени обработки запроса, удаление связанных с ним данных из BUF MGM, удаление его из очереди.

В процессе работы первая стадия ищет запросы в очереди со статусом `Wait`. Как только такой запрос найден, из него извлекается первый подзапрос *select-project* со статусом `Wait` и отправляется в узлы IO, статус подзапроса изменяется на `SelectProcessing`, а статус запроса изменяется на `Select`. Следующий статус подзапроса `TransferSelectResult` присваивается сразу при поступлении пакета `SelectResult` в обработчик первой стадии. После получения последнего пакета с данными статус подзапроса устанавливается в `SelectProcessed`, а обработчик стадии переходит к следующему подзапросу *select-project* или к новому запросу, если статус всех подзапросов *select-project* текущего запроса отличен от `Wait`.

Обработчик второй стадии ищет в очереди запрос со статусом `SelectProcessed` и извлекает из него первый подзапрос *join* со статусом `Wait`. Из подзапроса извлекается информация об отношениях, над которыми будет выполнена операция *join*. Выбирается узел JOIN для обработки. В выбранный узел передается пакет `RelationPreparePacket`. Отношениям в подзапросе присваивается статус `Preparing`, а подзапросу *join* – статус `TransferToJoin`. Как только обработчик стадии получит ответный пакет `RelationPreparedPacket`, статус отношения изменяется на `Prepared`, а управление передается следующему обработчику.

На третьей стадии обработчик находит в очереди запрос со статусом подзапроса, установленным в `TransferToJoin`, и статусами отношений `Prepared`. Находит ассоциированный с запросом узел и передает ему данные для этих отношений пакетами `RelationDataPacket`. После отправки всех пакетов устанавливает статусы отношений в `Transfered`.

На четвертой стадии выполняется поиск запроса в очереди с подзапросом, для которого все отношения находятся в статусе `Transfered`. В ассоциированные

узлы JOIN отправляются пакеты запуска операции *join* `JoinStartPacket`, статус подзапроса устанавливается в `JoinProcessing`, а статусы отношений – в `Processing`. Как только обработчиком будет получен пакет о завершении *join* `JoinCompletePacket`, статус подзапроса устанавливается в `JoinProcessed`, а статусы отношений – в `Processed`.

Пятая стадия конвейера вызывает обработчик, который выполняет поиск запроса в очереди со статусом последнего *join*-подзапроса `JoinProcessed`. Выполняется отправка запроса на передачу результата *join* в BUF пакетом `QueryPacket`. Статус подзапроса изменяется на `TransferJoinResult`. Обработчик принимает пакеты `SelectResult` с результатом *join* и сохраняет их в BUF. После получения последнего пакета он устанавливает статус подзапроса в `JoinResultTransfered`.

Обработчик шестой стадии ищет в очереди запрос с подзапросом *join*, статус которого соответствует `JoinResultTransfered`. Из запроса извлекается подзапрос *sort*, а из него – информация об отношениях, над которыми будет выполнена операция *sort*. Выбирается узел SORT для обработки. В выбранный узел передается пакет `RelationPreparePacket`. Отношениям в подзапросе присваивается статус `Preparing`, а подзапросу *sort* – статус `TransferToSort`. Как только обработчик стадии получит ответный пакет `RelationPreparedPacket`, статус отношения изменяется на `Prepared`, и управление передается следующему обработчику.

На седьмой стадии, по аналогии с третьей стадией, обработчик находит в очереди запрос со статусом подзапроса `TransferToSort` и статусами отношений `Prepared`. Находит ассоциированный с запросом узел и передает ему данные для этих отношений пакетами `RelationDataPacket`. После отправки всех пакетов, устанавливает статусы отношений в `Transfered`.

На восьмой стадии, как и на четвертой, выполняется поиск запроса в очереди с подзапросом, для которого все отношения имеют статус `Transfered`. В ассоциированные узлы SORT отправляются пакеты `SortStartPacket` для запуска операции *sort*, статус подзапроса устанавливается в `SortProcessing`, а статусы отношений – в `Processing`. Как только обработчиком будет получен пакет `SortCompletePacket` о завершении *sort*, статус подзапроса устанавливается в `SortProcessed`, а статусы отношений – в `Processed`.

Девятая стадия конвейера вызывает обработчик, который выполняет поиск запроса в очереди со статусом *sort*-подзапроса `SortProcessed`. Выполняется отправка запроса на передачу результата *sort* в BUF пакетом `QueryPacket`. Статус подзапроса изменяется на `TransferSortResult`. Обработчик принимает пакеты `SelectResult` с результатом sort и сохраняет их в BUF. После получения последнего пакета он присваивает подзапросу статус `SortResultTransfered`.

На десятой стадии осуществляется анализ времени нахождения всех запросов очереди в текущей стадии. Если какой-либо запрос находится в одной стадии слишком долго, то к нему применяется процедура очистки: рассылаются пакеты `DropQueryPacket`, запрос удаляется из очереди, информация о запросе заносится в журнал отброшенных запросов.

Одиннадцатая стадия выполняет очистку от выполненных запросов. Выполняется поиск в очереди всех запросов со статусом `SortComplete`. Для каждого найденного запроса выполняется: сохранение результата из BUF MGM в файл на диске, запись времени обработки запроса в журнал, удаление запроса из очереди.

BUF. Предоставляет хранилище «ключ-значение». В качестве ключа используется идентификатор подзапроса/отношения, а в качестве значения – динамический список блоков данных с возможностью их добавления и удаления. Блок данных состоит из идентификатора подзапроса, данных (массива байт), порядковый номер блока и метку последнего блока. Порядковый номер необходим для контроля получения всех пакетов, а пометка последнего блока для завершения передачи блоков в другие узлы.

Модуль IO отвечает за получение данных от СУБД в результате обработки запросов *select-project*. Работа модуля организуется по стратегии «запрос на ядро». Модуль взаимодействует с СУБД через драйвер, ведет контроль выполнения запросов *select-project*, передает результаты работы модулю MGM.

Он включает:

– Очередь запросов – подсистема ведения очереди запросов *select-project*.
– Менеджер SELECT – подсистема управления процессом выполнения запросов *select-project*.
– Обработчик результата SELECT – модуль формирования блока данных и отсылки его в MGM.

– Модуль сетевого взаимодействия.
– Подсистема мониторинга.
– Драйвер СУБД.

Рассмотрим каждый из них подробнее, за исключением рассмотренных ранее модуля сетевого взаимодействия, подсистемы мониторинга и драйвера СУБД.

ОЧЕРЕДЬ ЗАПРОСОВ. Является упрощенным аналогом очереди запросов в MGM. Основное его отличие – работа только с подзапросами *select-project*, статусы не учитываются. Так, при поступлении в очередь запросов IO нового *select-project*, он сразу передается в *МЕНЕДЖЕР SELECT*.

МЕНЕДЖЕР SELECT. Управляет обработкой запроса *select-project*: передает запрос драйверу СУБД, приостанавливает и возобновляет его работу по требованию MGM, передает результат выполнения запроса в *ОБРАБОТЧИК РЕЗУЛЬТАТА SELECT* с указанием идентификатора запроса и флагом окончания обработки (если полученный результат был последним для этого запроса).

ОБРАБОТЧИК РЕЗУЛЬТАТА SELECT. Получает данные результата *select-project*, идентификатор запроса и флаг окончания его обработки. Если ранее идентификатор запроса не обрабатывался, то ему сопоставляется счетчик блоков. Формирует пакет `SelectResult` с указанием номер блока := текущее значение счетчика блоков. Увеличивает счетчик блоков. Если флаг окончания обработки был установлен, то ассоциированный с идентификатором запроса счетчик удаляется.

Модуль JOIN предназначен для выполнения операций *join*, загрузки/удаления временных отношений в СУБД с помощью драйвера СУБД. Он управляет процессом обработки запросов *join* и подготовкой данных для них, выдает результат последнего *join* в MGM. Программный модуль включает:

– Очередь запросов – подсистема ведения очереди запросов *join*.
– Загрузчик данных – модуль подготовки отношений и загрузки данных в СУБД.
– Менеджер JOIN – подсистема управления процессом выполнения запросов *join*.
– Модуль сетевого взаимодействия.
– Подсистема мониторинга.
– Драйвер СУБД.

Рассмотрим каждый из них подробнее, за исключением модуля сетевого взаимодействия, подсистемы мониторинга и драйвера СУБД, т.к. они рассмотрены ранее.

ОЧЕРЕДЬ ЗАПРОСОВ. Является упрощенным аналогом очереди запросов в MGM. Основное его отличие – работа только с подзапросами *join* и их отношениями. Статусы подзапросов *join* определяются на основе статусов отношений: если все отношения находятся в статусе `Processed`, то этот подзапрос *join* выполнен, иначе – нет. Статусы отношений устанавливаются менеджером *join* и могут принимать те же значения, что и отношения в очереди запросов MGM.

ЗАГРУЗЧИК ДАННЫХ. Выполняет загрузку данных в 3 этапа: подготовка отношений для загрузки данных, подготовка данных к загрузке и загрузка данных в СУБД с помощью драйвера.

Этап подготовки отношений начинается сразу с поступлением пакета `RelationPreparePacket` с описанием схемы отношения. Причем, в очереди запросов должен находиться подзапрос *join* с идентификатором отношения, соответствующим таковому из полученного пакета. Статус отношения устанавливается в `Preparing`. В СУБД оправляются команды по созданию нового отношения с полученной схемой, созданию для него индексов и ключей. По получении ответа драйвера об успешном завершении всех операций, статус отношения изменяется на `Prepared`, и в MGM отравляется пакет `RelationPreparedPacket` с идентификатором подготовленного отношения. В случае возникновения ошибки во время создания отношения, делается соответствующая запись в журнал, работа над запросом *join* прерывается.

Этап подготовки к загрузке данных отношения начинается с поступления пакета `RelationDataPacket` с данными для отношения. В очереди запросов выполняется поиск запроса *join* с идентификатором отношения из полученного пакета. Отношению присваивается статус `TransferData`. Данные из полученного пакета сохраняются в файл с именем в формате `[идентификатор отношения]_[номер блока]_[флаг окончания]`. Например, для не последнего пятого пакета имя примет вид: `865fda1c-db06-459a-8a43-ba4f39b11d2a_0005_false`.

Этап загрузки данных в СУБД начинается с появлением в очереди запросов отношений со статусом `TransferData`. Для таких отношений выполняется поочередная загрузка всех полученных на предыдущем шаге файлов данных путем вызова метода его загрузки `LoadFile` в драйвере СУБД. Возврат из метода означает

окончание загрузки файла, и он удаляется. Если загруженный файл был последним, то загрузка данных в отношение завершена, и его статус переводится в `Transfered`.

МЕНЕДЖЕР JOIN. Управляет обработкой запроса *join* над подготовленными отношениями. Начинает свою работу с поступлением пакета начала операции *join* `JoinStartPacket`. Содержит подзапрос *join* и идентификатор нового отношения, куда будет записан результат. Отношение с результатом создается загрузчиком данных, но проходит только первый этап. В полученном запросе *join* выполняется проверка готовности всех необходимых для его выполнения отношений. Если статус хотя бы одного из требуемых отношений не `Transfered`, то выполнение *join* откладывается до готовности всех отношений. Интервал проверки готовности равен 100 мс. Как только все отношения готовы, их статус изменяется на `Processing`, в СУБД, через метод драйвера `QueryIntoRelation`, передается запрос *join* и название нового отношения в формате `tmp_[идентификатор отношения]` (`tmp_865fda1c-db06-459a-8a43-ba4f39b11d2a`). В результате будет выполнен запрос, и результат его работы помещен в новое отношение. Исходные два отношения больше не нужны. Поэтому они удаляются. Статус нового отношения изменяется на `Transfered`, а статусы исходных отношений устанавливаются в `Processed`. Процесс повторяется до тех пор, пока не будут выполнены все подзапросы *join* в рамках одного запроса. Выполнение каждого *join* вызывает отправку пакета `JoinCompletePacket` в MGM с указанием его идентификатора.

Передача результата работы *join* производится по требованию MGM (по получении пакета `QueryPacket` с запросом на выборку данных от итогового отношения). Выборка данных из результирующего отношения выполняется аналогично выборке данных в программном модуле IO: запрос передается в драйвер СУБД, полученные блоки пересылаются в MGM. После отправки последнего блока итоговое отношение уничтожается.

Модуль SORT предназначен для выполнения операций *sort*, загрузки/удаления временных отношений в СУБД с помощью драйвера СУБД. Он управляет процессом обработки запросов *sort* и подготовкой данных для них, выдает результат в MGM. Программный модуль включает:

– Очередь запросов – подсистема ведения очереди запросов *sort*.
– Загрузчик данных – модуль подготовки отношений и загрузки данных в СУБД.
– Менеджер SORT – подсистема управления процессом выполнения запросов *sort*.
– Модуль сетевого взаимодействия.
– Подсистема мониторинга.
– Драйвер СУБД.

Рассмотрим каждый из них подробнее, за исключением модуля сетевого взаимодействия, подсистемы мониторинга и драйвера СУБД, т.к. они рассмотрены ранее. Кроме того, модули очереди запросов и загрузчика данных являются полными аналогами этих модулей в JOIN. Единственное отличие заключается в том, что работа в данном случае производится с подзапросами *sort*, а не *join*. Поэтому их рассмотрение тоже опущено.

МЕНЕДЖЕР SORT. Управляет обработкой запроса *sort* над подготовленными отношениями точно так же, как это делает JOIN с подзапросами *join*. Отличие лишь в том, что у подзапроса *sort* нет других зависимых подзапросов, т.е. после загрузки данных в отношения нужно выполнить лишь один запрос. Это существенно упрощает реализацию менеджера SORT, т.к. после выполнения подзапроса *sort* сразу отправляется пакет `SortCompletePacket` в MGM.

Передача результата работы из SORT в MGM производится аналогично передаче из JOIN в MGM.

Конфигурация программных модулей хранится в файлах с расширением *.config* для каждого программного модуля отдельно:

– `ClusterixN.Manager.exe.config` – конфигурация MGM.
– `ClusterixN.IO.exe.config` – конфигурация IO.
– `ClusterixN.Join.exe.config` – конфигурация JOIN.
– `ClusterixN.Sort.exe.config` – конфигурация SORT.
– `ClusterixN.JoinManager.exe.config` – конфигурация HASH.

Во всех конфигурационных файлах имеются настройки подсистемы журналирования, настройки модуля сетевого взаимодействия, а также ряд специфичных настроек для конкретного модуля.

Все конфигурационные параметры для всех программных модулей представлены в документации Clusterix-N [94].

Параметр порядка начального состояния. Для его выявления был проведен эксперимент при $V_{БД}$ = 70 GB. На узлах IO, JOIN и MGM (для модуля SORT) выполнены соответствующие настройки MySQL. Режим выполнения *join* – интегрированный. Распределение узлов кластера: IO – 2 узла, JOIN – 3 узла, модули MGM и SORT совмещены в одном узле. Во избежание перегрузки RAM узлов использовалось правило 20% запаса [76]: если объем доступной оперативной памяти в узле оказываеся менее 30 GB, то передача новых данных в узел приостанавливается.

При этом из-за недостатка памяти и превышения границы времени нахождения запроса на одном этапе (0,5 часа), системой было «отброшено» 19 запросов, что составляет 45% от общего числа запросов. Время ожидания 0,5 часа выбрано в силу того, что время обработки отдельно взятого запроса к БД объемом 70GB не превышает 30 минут (согласно линейному росту времени выполнения запроса [95]).

Переход к последовательной операции *join*, увеличивает время обработки более чем в 3,5 раза [96]. Такое «замедление» объясняется необходимостью создания промежуточных отношений и их индексацией. С учетом накладных расходов на передачу данных по сети, подготовки отношений и загрузки данных время ожидания было увеличено в 4 раза (до 2 часов). Но и в этом случае не удалось избавиться от отбрасывания запросов. При выполнении первой перестановки из ПТ (14 запросов) были отброшены запросы с номерами 2, 4, 9. Запрос №9 характеризуется большим удельным весом операций *join*, запрос №4 использует в *join*-обработке 0,45 $V_{БД}$ [97]. Обработка обоих запросов превысила время нахождения запроса на одном этапе: запрос №4 «не успел» загрузиться в БД, а операции *join* в запросе №9 «не успели» выполниться вовремя, т.к. в такой конфигурации существенно увеличилось время выполнения *join*. Запрос №2 выполнялся сразу за запросом №4 и ожидал его завершения. Как только до него дошла очередь, он уже превысил время нахождения запроса на одном этапе и был сразу отброшен.

Таким образом, *параметром порядка* для начального состояния ***IS***-модели является *ненадежность работы системы даже при объемах БД менее 100 GB*.

2.7. Выводы по главе 2

Задача актуализуется лишь тогда, когда она может быть решена в современных условиях. Поэтому, согласно приведенной во введении формулировке основной задачи работы, помимо принятых ограничений, ее постановка необходимо включила:

– разработку предлагаемых методов претрансляции запросов и настройки MySQL на полную загрузку всех процессорных ядер;
– формулировку системы постулатов, выбор и разработку начального состояния процесса ***IS***-моделирования, согласно принятой методологии КМС.

Дальнейшая задача исследований состоит в разработке соответствующих программных систем (этапы внутреннего ***IS***-моделирования) и сравнительной оценке эффективности получаемых решений. Несмотря на практическую несостоятельность начального состояния как представителя класса BigData, его детальная программная разработка позволила создать своеобразные «модули-заготовки», которые в дальнейшем модифицировались для каждого нового состояния. Наличие таких «заготовок» существенно облегчило проведение последующих итераций.

Сделанный выбор начального состояния ***IS***-модели Clusterix-N допускает подключение к системе различных инструментальных СУБД в специальных конфигурациях без существенного изменения системы (требуется только разработать драйвер СУБД). Терминал управления в модуле MGM позволяет управлять системой, а генератор запросов выполнять тестирование производительности с различной нагрузкой. Динамический мониторинг свободной оперативной памяти позволяет системе избегать переполнения RAM, останавливая узлы с модулем IO и прекращая передачу данных в узлы с модулями JOIN. Разделение программных модулей по этапам обработки запроса позволяет использовать разные стратегии обработки в каждом отдельно взятом модуле. Это может быть полезно при настройке Clusterix-N на обработку специфичной нагрузки. Подсистема мониторинга позволяет оценить эффективность работы системы под нагрузкой и показать, какие операции совмещаются во времени. Отчет по работе системы предоставляет информацию о распределении работ по этапам обработки запроса.

ГЛАВА 3. *IS*-МОДЕЛИРОВАНИЕ CLUSTERIX-N. ИТЕРАЦИИ 1 – 4

В разделе 3.1 рассматривается архитектура Clusterix-N для первой итерации внутреннего ***IS***-моделирования [81]. Изменениям подверглись практически все модули начального состояния ***IS***-модели, кроме SORT. Появилось множество новых конфигурационных параметром. Никакие прежние режимы работы или конфигурационные параметры удалены не были.

В разделе 3.2 проведено экспериментальное исследование на первой итерации внутреннего ***IS***-моделирования при объеме базы данных $V_{БД}$ = 60GB и 120GB. Разделы 3.3, 3.4, 3.5 посвящены второй, третьей и четвертой (заключительной) итерациям ***IS***-моделирования системы Clusterix-N.

3.1. Архитектура для первой итерации внутреннего *IS*-моделирования Clusterix-N

Ранее в Clusterix-N была исключена динамическая сегментация (хеширование) промежуточных/временных отношений. Такой отказ был связан с высокой трудоемкостью этих операций и чрезмерной загрузкой сети при их выполнении. Вместе с тем, работа ***надежная*** работа с БД объемом в десятки и сотни GB требует распределения работ по ядрам и узлам, чему способствует такая сегментации.

Суть модификации 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. В результате внесенных изменений Clusterix-N теперь состоит из 5 модулей: MGM, IO, JOIN, HASH, SORT. Новая архитектура Clusterix-N представлена на рис. 3.1.

В модуле сетевого взаимодействия добавлен новый пакет `HashedRelationDataPacket`, являющейся полной копией пакета `RelationDataPacket` с указанием индекса хеширования. Изменилась и схема обмена пакетами между программными модулями JOIN и MGM (рис. 3.2). Теперь между ними находится новый модуль HASH, который повторяет интерфейс сетевого взаимодействия модуля JOIN. Модуль IO модифицировался для реализации стратегии «узел на запрос», т.е. для вы-

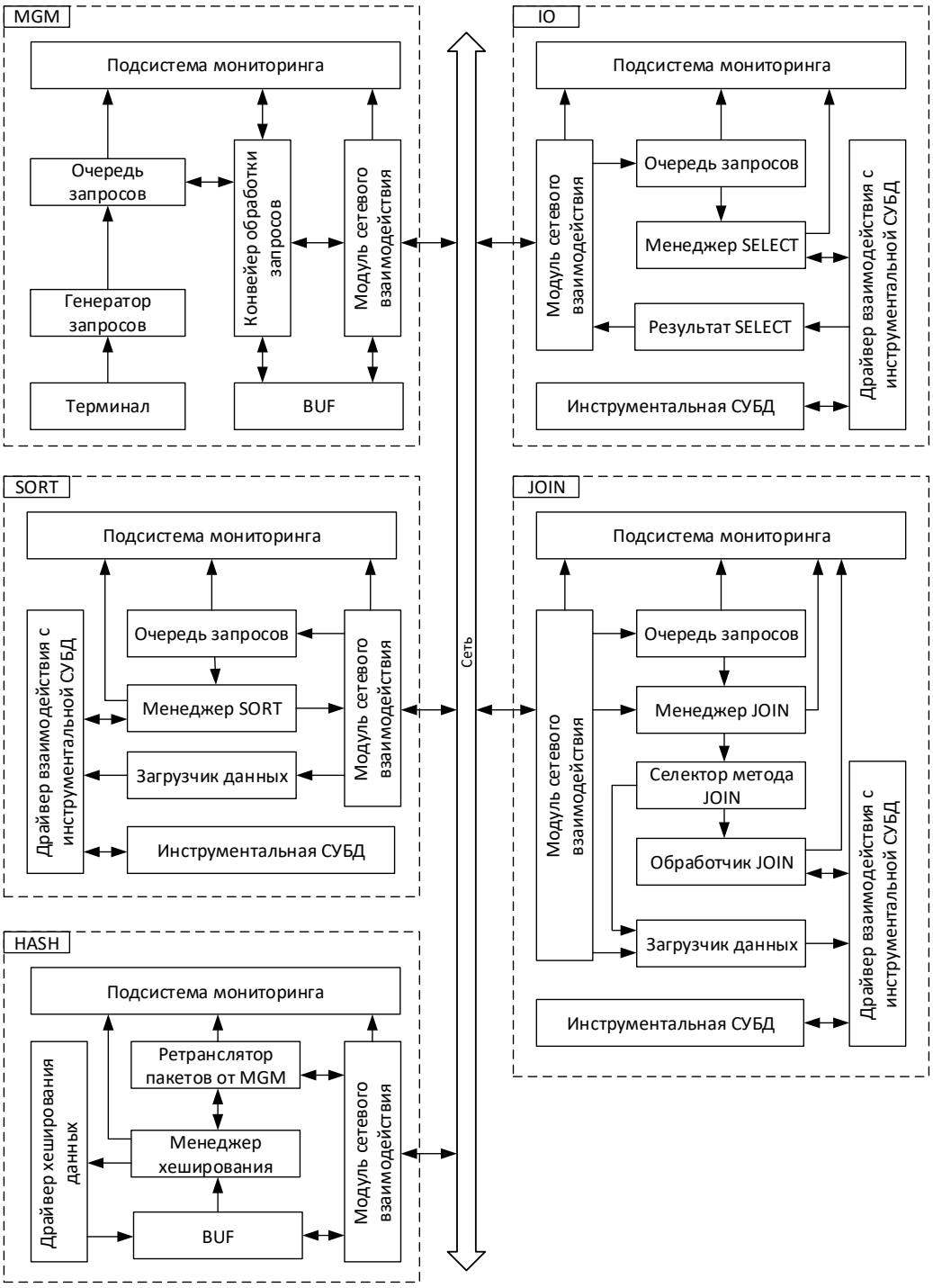


Рис. 3.1. Новая архитектура Clusterix-N

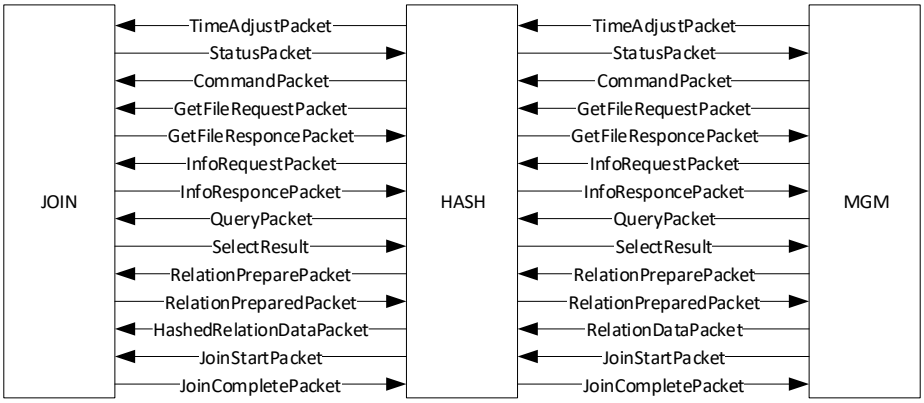


Рис. 3.2. Новая схема обмена пакетами между программными модулями JOIN и MGM

полнения запросов из очереди по одному параллельно на всех доступных процессорных ядрах. В то же время модуль SORT и модуль журналирования не претерпели каких-либо серьезных модификаций.

Работа модифицированной версии Clusterix-N для БД объемом в десятки и сотни GB представлена на диаграмме функционирования в приложении В.1. Здесь все блоки расположены сверху вниз в соответствии с временем обработки запроса. Блоки, расположенные в одном ряду, выполняются параллельно. Запуск отдельных блоков происходит либо передачей управления (черные стрелки), либо по готовности данных (пунктирные стрелки). Блоки, расположенные в левом столбце диаграммы, образуют внешний конвейер.

После начала работы MGM подзапросы *select-project* и *join* поступают соответственно в узлы IO и JOIN, где они модифицируются и передаются в MySQL, который их выполняет. MGM генерирует очередной запрос, помещает его в очередь и устанавливает статус в `Wait`. Как только он обнаруживает, что все узлы IO способны принять очередной подзапрос *select-project*, он отправляет его. Операция отправки запроса повторяется до тех пор, пока не будут отосланы все подзапросы на выборку для каждого из отношений. IO выполняют подзапросы последовательно согласно стратегии «запрос на узел». Результат работы узлов IO поступает в BUF MGM в виде блоков данных. Они накапливаются в буфере MGM по отношениям. Когда все блоки одного из отношений получены, они отправляются в узел HASH, где подвергаются хешированию и перераспределению. Хешированные данные передаются 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 обработки подзапросов передается в узел HASH, где хешируется и отправляется в узлы JOIN для обработки следующего подзапроса *join*. Операция повторяется до завершения обработки всех подзапросов *join*. После завершения всех подзапросов *join*, итоговый результат передается в BUF MGM. Оттуда результат отправляется для дальнейшей обработки в SORT. Результат *sort*-обработки передается в MGM, где он сохраняется. Основное отличие нового модуля MGM от

представленного в разделе 2.6 кроется в переходе на стратегию «группа узлов на запрос». При этом, все узлы IO одновременно работают над одним подзапросом. В то же время узлы JOIN (и, соответственно, HASH) могут работать над другим.

Программный модуль HASH предназначен для выполнения операций хеширования и распределения данных по узлам JOIN. Он является прокси-сервером между модулем MGM и модулем JOIN. Получает задания и данные от MGM. Выполняет хеширование данных на GPU или CPU. 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.

Хеширование данных на CPU не обладает достаточной производительностью, т.к. скорость хеширования оказывается ниже скорости передачи данных по сети. Поэтому модуль HASH организует динамическую сегментацию промежуточных/временных отношений на выделенном узле с GPU-ускорителями, выполняя операции хеширования на GPU и распределяя данные по всем процессорным ядрам уровня JOIN [98]. Хеширование выполняется с использованием алгоритма деления [64]. Значение хеш-функции – остаток от деления суммы ключевых полей строки на количество процессорных ядер JOIN.

Программный модуль включает в себя модули, показанные на рис. 3.1:

- Ретранслятор пакетов от MGM – модуль передач пакетов от MGM в подключенные к HASH узлы JOIN.
- Менеджер хеширования – подсистема управления процессом выполнения хеширования.
- Драйвер хеширования данных – реализация метода хеширования.
- BUF – промежуточный буфер данных для хранения хешированных данных.
- Модуль сетевого взаимодействия.
- Подсистема мониторинга.

Рассмотрим каждый из них подробнее, за исключением модуля сетевого взаимодействия, подсистемы мониторинга и драйвера СУБД, т.к. они были рассмотрены ранее.

РЕТРАНСЛЯТОР ПАКЕТОВ ОТ MGM. Обеспечивает сокрытие группы узлов JOIN за одним HASH. Он принимает пакеты от MGM, тиражирует их всем подключенным узлам JOIN «как есть» или перехватывает и передает менеджеру хеширования. Пакеты, полученные от узлов JOIN, собираются в буфере ретранслятора. Как только получены все пакеты для определенного отношения со всех подключенных JOIN, HASH отправляет в MGM один консолидированный пакет. Однако в системе есть пакеты, содержащие уникальную информацию для каждого узла. Эти пакеты ретранслируются без консолидации.

Следующие пакеты MGM → JOIN перехватываются модулем HASH:

- `RelationDataPacket` – данные из пакета передаются на обработку в менеджер хеширования. Результаты хеширования рассылаются по узлам JOIN в виде пакетов `HashedRelationDataPacket`.
- `RelationPreparePacket` – данные пакета используются для подготовки менеджера хеширования к работе над отношением, сам же пакет тиражируется по всем JOIN.
- `QueryPacket` – не передается узлам JOIN. Модуль HASH перехватывает его и отправляет готовый результат по готовности BUF.
- `JoinStartPacket` – данные пакета используются менеджером хеширования для управления процессом выполнения операций *join*. Пакет тиражируется по всем узлам JOIN.
- `GetFileRequestPacket` – пакет тиражируется по всем узлам JOIN, но обработка производится и в модуле HASH.

Следующие пакеты JOIN → MGM обрабатываются модулем HASH:

- `InfoResponcePacket` – перехватывается модулем HASH и не перенаправляется в MGM. Содержит информацию о подключенных узлах JOIN.
- `StatusPacket` – игнорируется и не перенаправляется в MGM, т.к. нет необходимости собирать информацию о нагрузке на подключенные узлы JOIN. Если отношение было успешно размещено в узле HASH, то после распределения по нескольким таким же узлам, оно точно будет успешно размещено в них.
- `JoinCompletePacket` – перехватывается модулем HASH, передается в менеджер хеширования и отправляется в MGM. Пакет отправляется в MGM только после его получения со всех подключенных улов JOIN.

- `SelectResult` – перехватывается модулем HASH и не перенаправляется в MGM. Служит для передачи временных отношений на хеширование для следующего *join* или для передачи результата последнего *join* в BUF.
- `RelationPreparedPacket` – перенаправляется в MGM и передается в менеджер хеширования. Пакет отправляется в MGM только после его получения со всех подключенных улов JOIN.
- `GetFileResponcePacket` – перехватывается модулем HASH и передается в MGM. Модуль HASH отправляет совместно с пакетами из узлов JOIN свой пакет с файлом.

Интегрированный *join* модулем HASH не поддерживается. Поэтому пакеты `IntegratedJoinStartPacket` и `IntegratedJoinCompletePacket` игнорируются.

МЕНЕДЖЕР ХЕШИРОВАНИЯ. Управляет процессом хеширования и параллельным выполнением *join* на нескольких узлах. Он включает в себя очередь запросов, подобную очереди в модуле JOIN, для отслеживания статусов отношений и подзапросов *join*. Вновь поступающие пакеты, относящиеся к операции *join*, обрабатываются менеджером хеширования. В очередь выполнения устанавливаются подзапросы *join* с требуемыми отношениями. Статусы отношений определяются аналогично модулю MGM, но статус `TransferData` означает не просто передачу данных отношения, а его хеширование и передачу хешированных данных.

Работа менеджера хеширования схожа с работой конвейера по обработке подзапроса *join* в модуле MGM:

1. Подготовка операций *join* – создание необходимых отношений во всех подключенных узлах JOIN по требования MGM.
2. Хеширование данных – выполнение операций хеширования и размещение результат в BUF.
3. Передача хешированных данных отношений *join* – передача всем подключенным узлам JOIN данных для загрузки в подготовленные отношения в соответствии с их порядковым номером.
4. Запуск операций *join* – передача команды на запуск *join* всем подключенным узлам JOIN по требования MGM.

5. Передача результата *join* в BUF – передача требования отправки результата *join* всем узлам.

Цикл из 5 шагов повторяется вплоть до получения последнего результата JOIN и отправки пакета `JoinComplete` в MGM. Результат последнего *join* хранится в BUF без применения операций хеширования и выдается из него в MGM по требованию.

ДРАЙВЕР ХЕШИРОВАНИЯ ДАННЫХ. Предоставляет интерфейс (листинг 3.1) для реализации произвольного метода хеширования. Единственный метод интерфейса `ProcessData` получает на вход: `data` – блок данных, `nodeCount` – количество узлов или суммарное количество процессорных ядер в узлах (если используются многоядерные узлы), для которых будет произведено хеширование, `keys` – номера ключевых столбцов, по которым будет произведено хеширование. Результат метода – список блоков, с индексами от 0 до `nodeCount` – 1, где блок с индексом 0 соответствует первому узлу/процессорному ядру, 1 – второму и т.д.

Листинг 3.1. Интерфейс драйвера хеширования

```
1. public interface IHasher
2. {
3.   List<byte[]> ProcessData(byte[] data, int nodeCount, int[] keys);
4. }
```

В составе Cluseterix-N реализовано 3 драйвера хеширования с использованием алгоритма деления [64]: хеширование на CPU, параллельное хеширование на CPU, параллельное хеширование на GPU. Значение хеш-функции – остаток от деления суммы ключевых полей строки на количество процессорных ядер JOIN. Первый драйвер обладает крайне низкой производительностью, его скорость хеширования в разы меньше скорости передачи данных по сети. Второй драйвер – улучшенная версия первого, обладает производительностью близкой к скорости передачи по сети. Эксперименты показали, что параллельное хеширование на CPU всего в 3-4 раза быстрее хеширования на одном ядре 12-ядерного узла. Столь низкий прирост производительности можно объяснить наличием всего 4-х каналов RAM, что не позволяет использовать все процессорные ядра для хеширования. Хеширование на GPU показало производительность на уровне удвоенной скорости передачи данных по сети GigabitEthernet. Поскольку только хеширование на GPU дало необходимую производительность все дальнейшее рассмотрение будет производится с ним. Хеширование на GPU выполняется согласно следующему алгоритму.

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* преобразовываются в блоки данных и передаются в BUF. В процессе передачи хешированных блоков данных по ядрам адрес назначения (узел и ядро) определяется однозначно. Каждый узел JOIN содержит одина-

ковое 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 $N{\cdot}Len$~1000MB для исходного блока данных плюс объем массивов H, Se и Ss: $N{\cdot}4{\cdot}3$~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.

BUF. Предоставляет промежуточной буфер для хешированных данных. Реализован аналогично BUF MGM с тем лишь отличием, что каждый хранимый в BUF блок данных содержит список хешированных блоков. Индекс хешированного блока соответствует номеру ядра, для которого он предназначен.

Модификация программного модуля IO. В модуле IO модификации подверглись менеджер SELECT и драйвер СУБД. Менеджер SELECT теперь поддерживает работу по стратегии «узел на запрос». Новые запросы поступают в очередь и выполняются из нее по одному. В отличие от предложенной ранее стратегии «запрос на ядро», новая стратегия позволяет ускорить обработку подзапроса *select-project*. Ускорение достигается за счет модификации драйвера СУБД: поступающий в него запрос оптимизируется для многоядерных узлов, реализуя метод параллельной обработки селективных запросов согласно следующему алгоритму.

АЛГОРИТМ ПАРАЛЛЕЛЬНОГО ВЫПОЛНЕНИЯ SELECT-PROJECT. 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 – гото-

вый 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.

1. $N :=$ значение, установленное в настройках.
2. Подготовить множество модифицированных запросов в количестве процессорных ядер: $Q'_i := Q + \text{«}LIMIT\ P,N\text{»}$, $P := N \cdot (B + i)$, 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 не ключевому 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 достигается 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 по-

ложительно 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.

Модификация программного модуля JOIN. Методы последовательного и интегрированного *join* не позволили добиться приемлемой производительности. Разделение работ должно ускорить операции загрузки данных и выполнения *join*, за счет распараллеливания их по ядрам и узлам. Новый, параллельный метод *join* реализован в двух вариациях: встроенной и управляемой. Управляемый параллельный *join* организуется с выделенным узлом HASH. Встроенный параллельный *join* позволяет использовать все процессорные ядра узла для обработки одного запроса без необходимости выделения специализированного узла для хеширования. Хеширование по ядрам выполняется по тем же методам, с использованием тех же реализаций драйверов хеширования, что и в модуле HASH.

Загрузка данных небольшими блоками в БД оказалась довольно медленной, особенно на больших объемах данных. В то же время загрузка сразу всех данных отношения за один вызов метода `LoadFile` выполняется быстрее [99] (примерно на 15%). Вероятно, такое ускорение связано с тонкостями организации движка MEMORY. Поэтому загрузчик данных был модифицирован таким образом, чтобы полученные данные для отношений записывались в файл, и только после записи последнего блока выполнялась загрузка файла в БД.

Этим модификация загрузчика не ограничилась. В связи с введением хеширования по ядрам/узлам, потребовалось выполнять загрузку в несколько отношений в БД. Поэтому загрузчик данных для идентификации отношения в БД использует не только идентификатор отношения, а еще и индекс хеширования из нового пакета `HashedRelationDataPacket`. Никаких дополнительных настроек для загрузчика производить не требуется, т.к. обработка пакета `RelationDataPacket` не изменилась, а для нового пакета реализован обработчик, который занимается обработкой исключительно пакетов `HashedRelationDataPacket`. В случае же работы в режиме встроенного параллельного *join*, все полученные пакеты с данными отно-

шения `RelationDataPacket` передаются в драйвер хеширования. Результирующие хешированные буферы преобразуются в пакеты `HashedRelationDataPacket` и передаются в соответствующий обработчик, как будто он получен от модуля HASH.

Загрузка данных из пакета `HashedRelationDataPacket` производится в отношения с именем, указанным в формате `[имя отношения]_[идентификатор отношения]_tmp_[индекс хеширования из пакета]`. Например, если получен пакет `HashedRelationDataPacket` с индексом хеширования `6` и идентификатором отношения `d4a9b569-fd95-45d7-a522-2026cc2abb09`, то по идентификатору отношения будет определено его имя (простой поиск по очереди запросов). Пусть, например, имя определено как `NATION`. Тогда имя отношения для *join* примет вид: `NATION_d4a9b569-fd95-45d7-a522-2026cc2abb09_tmp_6`.

Реализация нового метода *join* и возможности встраивания драйвера хеширования потребовало добавления в архитектуру JOIN двух новых модулей:

– Селектор метода JOIN – позволяет выбирать метод выполнения операции *join* из числа представленных. Выполняет загрузку модуля реализации указанного метода *join* и передает ему управление.
– Обработчик JOIN – модуль непосредственного выполнения *join*.

С введением новых модулей изменился модуль менеджера JOIN. Теперь менеджер JOIN не выполняет непосредственный запуск *join* вызовом метода `QueryIntoRelation` в драйвере СУБД. Вместо этого он, после получения команды, передает управление селектору метода JOIN.

ОБРАБОТЧИК JOIN. Загружается и подготавливается селектором метода JOIN. Выполняет операцию *join* в соответствии со своей реализацией. Работа модуля зависит от загрузчика данных. Если данные были подготовлены неправильно, то обработчик JOIN не сможет выполнить подзапрос *join*. Например, если модуль настроен на работу по методу управляемого параллельного *join* и подключен к MGM напрямую, то окажется, что все данные отношения загружены в одно временное отношение рабочей БД, а не в несколько, как предполагает метод. Поэтому требуется точная и правильная настройка программного модуля JOIN в соответствии с его ролью в системе.

Обработчиков JOIN реализовано всего два:

– Простой *join* – выполняет операцию *join* над двумя отношением с одним вызовом драйвера СУБД.
– Параллельный *join* – выполняет операцию *join* над двумя хешированными отношениями параллельно в *N* потоков, делая *N* вызовов драйвера СУБД для выполнения подзапросов сразу над всеми отношениями, полученными в ходе хеширования.

Простой *join* был реализован в ранней версии и использовался в методах последовательного и интегрированного *join*. Теперь же он вынесен в отдельный модуль.

Обработчик параллельного *join* ожидает завершения загрузки хешированных отношений в БД, инициализирует потоки выполнения *join* в количестве, указанном в настройках [94], передав им подзапрос *join*. Каждый поток, получив подзапрос, заменяет в нем названия отношений `{LEFTRELATION}` и `{RIGHTRELATION}` на необходимые в формате `[имя отношения]_[идентифкатор отношения]_tmp_[номер потока/ядра]`, например, для отношения `ORDERS` с идентификатором `af0db745-fb97-4593-8ed9-4cc06e2d34a4` подставляемое имя во втором потоке примет вид: `ORDERS_af0db745-fb97-4593-8ed9-4cc06e2d34a4_tmp_2`. Именно такое имя временного отношения было использовано загрузчиком на этапе загрузки данных.

Результат операции *join* при ее управляемой параллельной реализации не записывается в новое отношение, а сразу передается в узел HASH.

Модификация программного модуля MGM. В модуле MGM изменился алгоритм работы в условиях недостатка оперативной памяти. Теперь он учитывает состояние всех подключенных узлов и позволяет сбрасывать часть BUF на диск. В конвейере обработки запросов появилась возможность замены реализации этапов работы, что позволяет подстраивать его под определенный тип нагрузки. Был добавлен новый источник запросов, специально для реализации распределенного *join*.

Рассмотрим модификации в каждом из модулей отдельно.

ТЕРМИНАЛ, ГЕНЕРАТОР ЗАПРОСОВ и *ОЧЕРЕДЬ ЗАПРОСОВ* не претерпели никаких модификаций или доработок.

КОНВЕЙЕР ОБРАБОТКИ ЗАПРОСОВ был доработан для обеспечения выбора реализации этапов обработки. Этапов осталось столько же, но некоторые из них теперь имеют несколько реализаций:

1. Получение промежуточных отношений R_i

1.1. Отправка подзапросов *select-project* в узлы IO по количеству их процессорных ядер.

1.2. Отправка подзапросов *select-project* в узлы IO по количеству, указанному в конфигурационном файле.

4. Запуск операций *join.*

4.1. Запуск *join* по количеству ядер в узлах.

4.2. Запуск, указанного в настройках, количества *join* запросов в узлах.

Выбор реализации каждого из этапов конвейера производится в конфигурационном файле.

Ограничение количества отправляемых подзапросов в IO и JOIN связано с тем, что модифицированные модули обрабатывают один подзапрос параллельно на всех доступных ядрах. На уровне JOIN возможно организовать обработку одного запроса на нескольких узлах JOIN с помощью выделенного узла HASH, который для MGM выглядит как один мощный узел JOIN.

Этап подготовки операций *join* теперь производит выбор узлов JOIN как с учетом свободной оперативной памяти, так и без него. Отключение учета свободной оперативной памяти необходимо при работе с запросами, для которых требуется большой объем данных, превышающий объем RAM. В этом случае инструментальная СУБД настраивается на работу с дисками и нет необходимости контролировать объем свободной RAM.

BUF. Модифицирован для возможности работы в условиях дефицита объема оперативной памяти. Суть модификации заключается в сбросе на диск части блоков из буфера и их чтение с диска при необходимости и их восстановлении во время передачи промежуточных/временных отношений.

Наличие мониторинга доступной RAM позволяет организовать сброс части буфера на диск при достижении минимального доступного объема RAM, указанного в конфигурационном файле. При достижении порогового значения все последние блоки данных в BUF для каждого запроса сбрасываются на диск. Операция повторяется с получением каждого нового значения доступного объема RAM из подсистемы мониторинга до тех пор, пока оно не станет больше указанного минимального доступного объема RAM.

Чтение (восстановление) сохраненных на диск блоков начинается при первом обращении к BUF с требованием выдачи данных по идентификатору подзапроса/отношения. По идентификатору в хранилище находится множество блоков данных, некоторые из которых могут быть сброшены на диск. Сброшенные на диск блоки идентифицируются по флагу `IsFlushedToDisk` и считываются в память. Дальнейшая работа выполняется без изменений.

Диаграмма функционирования Clusterix-N для первой итерации показана в приложении В.1. Здесь все блоки расположены сверху вниз в соответствии с временем обработки запроса. Блоки, расположенные в одном ряду, выполняются параллельно. Запуск отдельных блоков происходит либо передачей управления (черные стрелки), либо по наличию данных (пунктирные стрелки).

Блоки «Передача запроса select-project для одного отношения в модули IO», «Создание схем промежуточных/временных отношений в узлах JOIN», «Запуск операции join», «Создание схем временных отношений в SORT», «Запуск операции sort» для данной итерации образуют внешний конвейер. Запросы select-project вновь поступившего запроса передаются в модули друг за другом по одному. Результат выполнения slect-project формируется в BUF MGM. Как только все отношения для данного запроса переданы в BUF MGM, начинается выполнение операции select-project для следующего запроса.

3.2. Экспериментальное исследование на первой итерации *IS*-моделирования Clusterix-N

Проверку успешности модификации архитектуры Clusterix-N выполним с помощью натурного эксперимента при объемах БД в десятки и сотни GB [100].

Постановка эксперимента с БД объемом в десятки и сотни GB. Эксперимент организован с принятыми в разделе 2.1 ограничениями. После загрузки данных в MySQL и их индексации, размер загруженных БД составил ~100 GB и ~200 GB соответственно для БД объемом 60 и 120 GB.

Конфигурация экспериментального полигона показана на рис 3.3. Она включает 2 узла IO, 3 узла JOIN, 1 узел HASH и 1 узел MGM, совмещающий модули MGM и SORT. БД распределена между узлами IO.

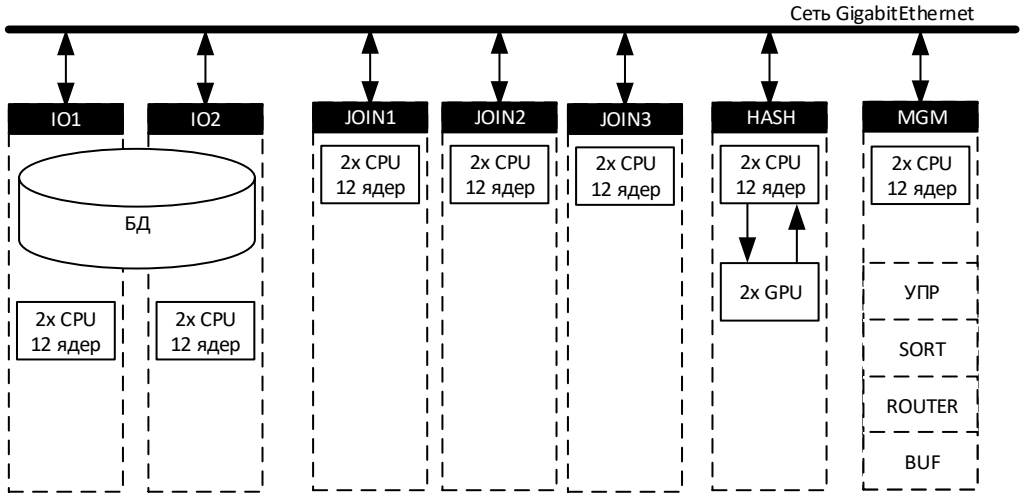


Рис. 3.3. Конфигурация экспериментального полигона для Clusterix-N

СУБД MySQL на узлах сконфигурирована согласно разделу 2.3. Конфигурация программных модулей для случая $V_{БД}$=60 GB выполнена следующим образом. Для всех модулей установлен порог доступной оперативной памяти в размере 30GB, размер блоков данных равен 1 млн. строк. Для модулей JOIN включено синхронное удаление данных в целях исключения ситуация одновременной загрузки и удаления данных. Количество ядер, отводимых для параллельного *join*, равно 10, т.к. из 12 доступных ядер, 1 ядро занято приемом/передачей пакетов по сети и 1 ядро отведено под сервисные службы (журналирование, подготовка отношений и т.д.). Для модуля HASH установлено 2 очереди хеширования, соответствующие двум графическим ускорителям. Модуль MGM конфигурируется для реализации на уровнях IO и JOIN стратегии «узел на запрос», т.е. в узлы IO и JOIN запросы на обработку передаются по одному.

В случае $V_{БД}$=120GB применялась та же конфигурация с некоторыми отличиями. Во-первых, был изменен порог доступной оперативной памяти для увеличения рабочего объема: IO – 1 GB, JOIN – 20 GB, HASH – 5 GB. Для модуля MGM порог доступной оперативной памяти был увеличен до 40 GB в целях уменьшения вероятности исчерпания памяти узла. Во-вторых, размер блока для JOIN уменьшен до 500000 строк, т.к. выделение памяти под большие блоки требует больших свободных участков в оперативной памяти, которые ОС не всегда может выделить, поэтому размер блока уменьшен. Остальные настройки – без изменений.

Поскольку Clusterix-N использует множество узлов для работы на каждом уровне, а уровней обработки 3, то для обеспечения эффективности системы требу-

ется, чтобы в ней велась одновременная обработка по крайней мере 3-х запросов. Объем данных, необходимый для обработки всех 14 запросов, составляет $1{,}4V_{БД}$ [80], 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 была принята равной 128GB/(0.43•60GB)+3-1=4.96+3-1≈7, где 128GB – объем RAM узла MGM, (0.43•60GB) – объем памяти для хранения промежуточных отношений самого требовательного запроса, 3 – количество уровней обработки (после передачи данных требуемым узлам, данные из памяти MGM удаляются) , -1 – модуль SORT совмещен с модулем MGM. Для $V_{БД}$=120GB длина очереди также равна 7.

Результаты эксперимента с БД объемом в десятки и сотни GB. Основные результаты выполнения ПТ для двух объемов представлены в таблице 3.1. Нетрудно видеть, что рост времени обработки ПТ и времени ожидания практически линейно зависит от размера БД.

Таблица 3.1. Результаты обработки ПТ

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

Визуализация работы Clusterix-N при обработке ПТ для $V_{БД}$=60GB представлена на рис. 3.4, где JOIN_MGM1 – узел HASH. Как видно, двух узлов IO вполне достаточно для обеспечения полной загрузки уровня JOIN. Белые участки в визуализации работы IO говорят о том, что узлы простаивали, ожидая новых подзапросов *select-project*. Большое количество белых «просветов» присутствует в визуализации работы модуля SORT. Это означает, что модуль практически не загружен, т.к. после выполнения всех операций *join* объема данных существенно уменьшается. Узлы JOIN работают, в основном, без перерывов, кроме случаев ожидания данных для загрузки. На визуализациях узлов JOIN можно заметить, что загрузка данных в некоторые моменты времени производилась параллельно и сама операция *join* требует гораздо меньше времени, чем ее подготовка (передача данных, их подготовка и загрузка).

Процесс индексации был совмещен с загрузкой данных, поэтому срока визуализации индексации отсутствует. Узел HASH высоко нагружен, но одна закрашен-

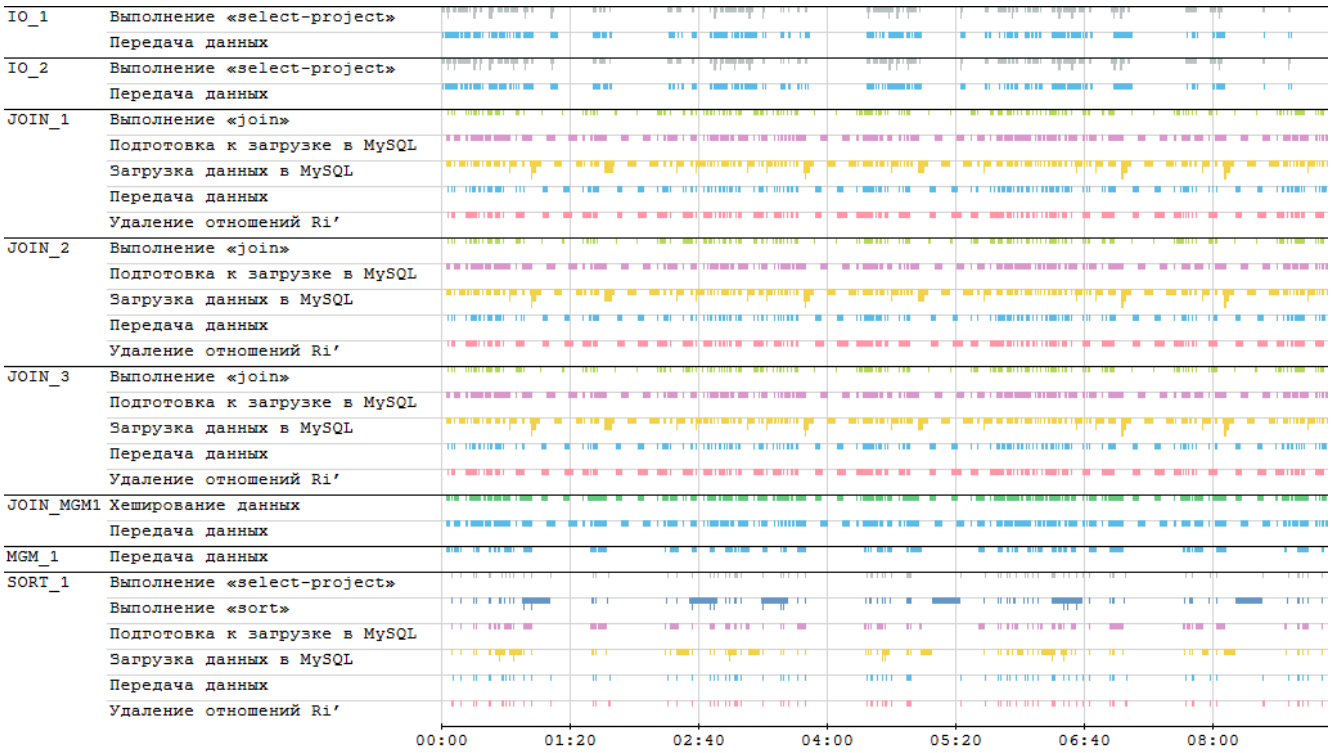


Рис. 3.4. Визуализация выполнения ПТ с $V_{БД}$=60GB

ная строка в операции хеширования говорит о том, что все операции выполнялись в одной очереди (на одном GPU). Связано это с тем, что один графический ускоритель выполняет хеширование в ~2 раза быстрее передачи данных по сети, поэтому второй ускоритель оказался не у дел. Передача данных по во всех модулях занимает существенное время, но особенно активно сеть используется узлами MGM и HASH.

Для $V_{БД}$=120GB визуализация (рис. 3.5) выглядит так же как для $V_{БД}$=60GB, но с одним явным отличием – нагрузка на модуль SORT существенно возросла. Это не удивительно, ибо объем результатов *join* увеличился.

Полученные графики загрузки CPU (рис. 3.6а для $V_{БД}$=60GB и 3.7а для $V_{БД}$=120GB) показывают использование всех процессорных ядер (загрузка 100%) в периоды активности. На узлах JOIN в процессе загрузки используется только одно ядро и в эти периоды загрузка CPU не превышает 8-16% (загружено 1-2 ядра). В то же время на узле HASH загрузка не превышает 8-16% за все время работы. Это объясняется тем, что операции хеширования (основная нагрузка) перенесены в графические ускорители. Низкая загрузка узла MGM при $V_{БД}$=60GB объясняется малым объемом работ. Но при $V_{БД}$=120GB нагрузка на узел существенно возрастает.

Графики загрузки RAM узлов (рис. 3.6б для $V_{БД}$=60GB и 3.7б для $V_{БД}$=120GB) показывают ее утилизацию во времени. Для $V_{БД}$=60GB все данные

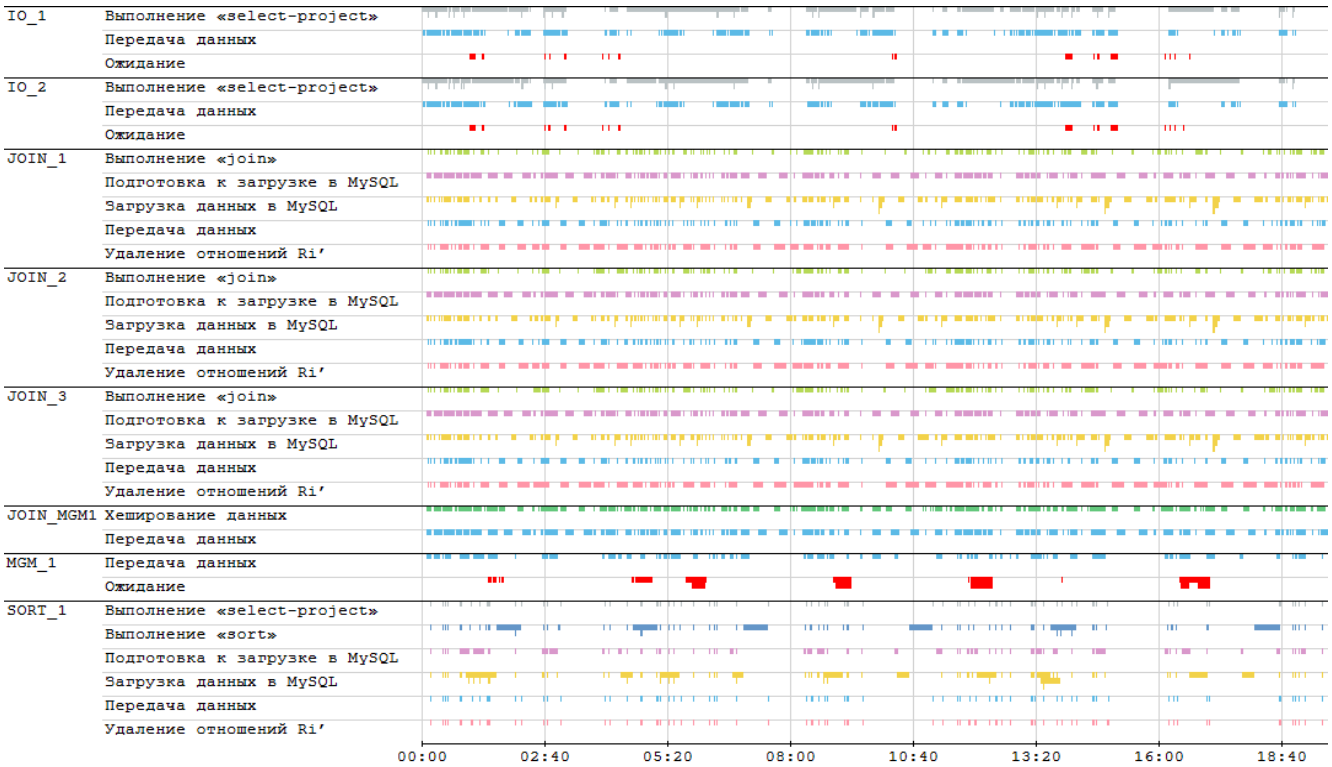
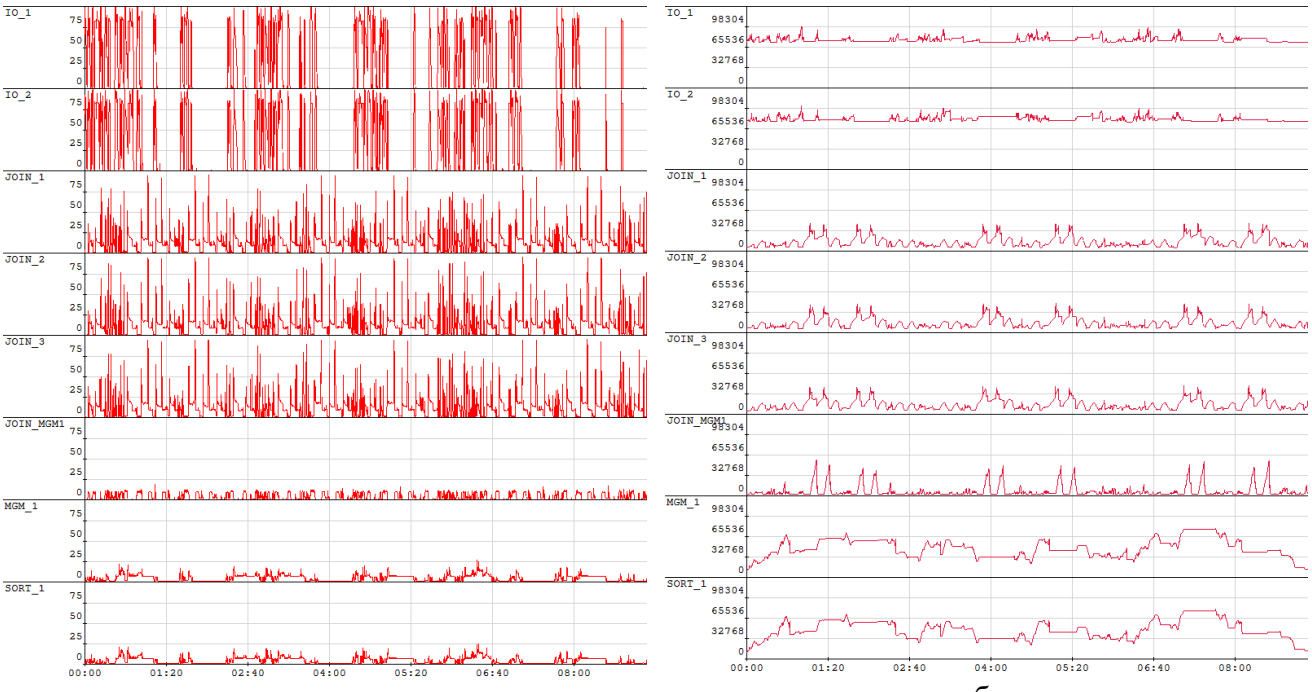


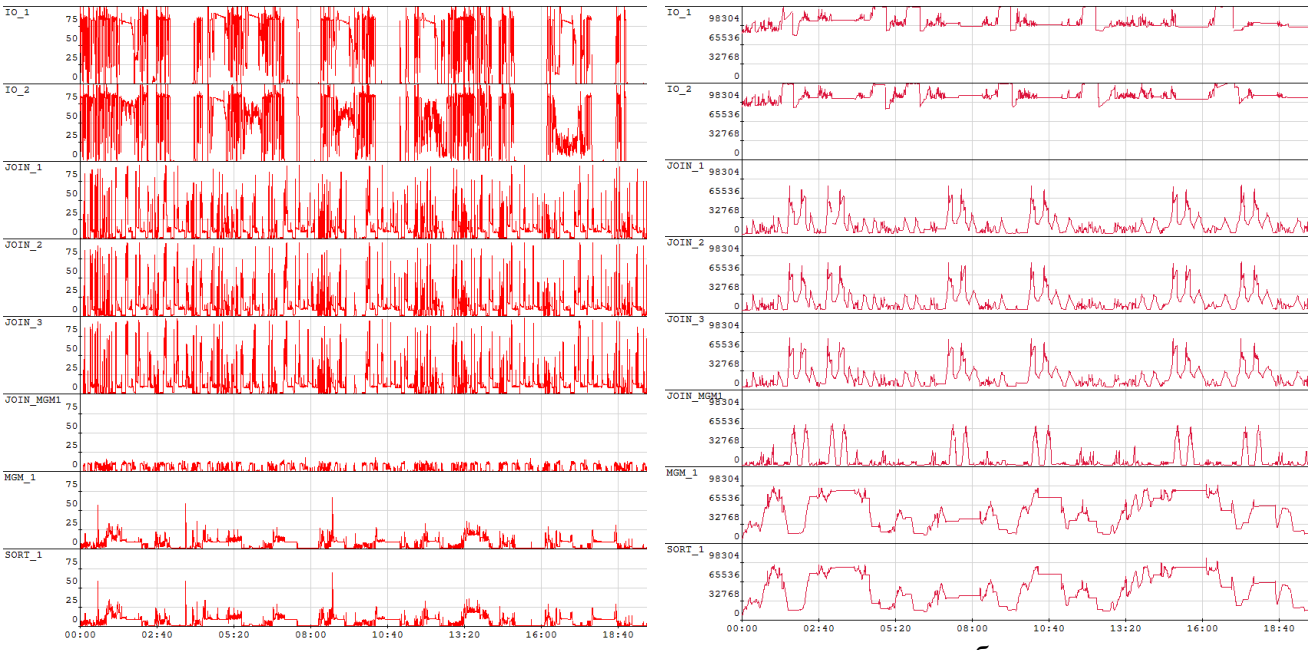
Рис. 3.5. Визуализация выполнения ПТ с $V_{БД}$=120GB

умещаются в RAM. На графике 3.6б можно наблюдать пиковое потребление памяти, которое не превышает 45GB на узлах JOIN и HASH, 70GB на узле MGM и 95GB на узлах IO. Так что порог свободной оперативной памяти нигде не был превышен. Однако при $V_{БД}$=120GB все данные не умещаются в RAM. На графике 3.7б видны горизонтальные промежутки на уровне порога доступной RAM для узлов



а б

Рис. 3.6. Визуализация а – загрузки CPU, б – объема занятой RAM во время обработки ПТ для $V_{БД}$=60GB



а б

Рис. 3.7. Визуализация а – загрузки CPU, б – объема занятой RAM во время обработки ПТ для $V_{БД}$=120G

IO и MGM. Но для узлов JOIN и HASH потребление RAM не превышает 80-90 GB, что ниже указанных в конфигурации ограничений. Использование диска на узлах IO и MGM, вследствие превышения порогового объема памяти, отразилось на производительности слабо (время обработки ПТ выросло линейно).

Как показал эксперимент с $V_{БД}$=60GB длины очереди запросов равной 7 достаточно, т.к. память MGM не используется полностью и узлы JOIN/HASH не простаивают. Увеличение длины очереди позволит полностью утилизировать память MGM, но никакого выигрыша во времени обработки ПТ получить не удастся.

В случае $V_{БД}$=120GB длина очереди в 7 запросов также оказывается достаточной и, возможно, даже избыточной, т.к. в процессе возникает перегрузка памяти MGM и приостановка работы узлов IO. Тем не менее перерывов в работе узлов JOIN/HASH не возникает, т.к. в BUF MGM уже имеются 2 полностью готовых к обработке результатов *select-project*. Уменьшение очереди до 6 запросов может помочь избежать приостановки работы узлов IO, но целесообразность такого изменения сомнительна. BUF MGM при необходимости сбрасывает часть данных на диск. Поэтому перегрузки RAM возникать не должно. Но увеличение длины очереди приведет к возрастанию среднего времени ответа, а уменьшение должно привести к его сокращению.

После проведенной итерации ***S***-моделирования система Clusterix-N осталась все же неконкурентоспособной в сравнении с open source системой Spark. Поэтому необходимы дополнительные итерации ***S***-моделирования Clusterix-N с целью получения сравнимого со Spark результата.

Для перехода к следующей итерации необходимо обозначить новый параметр порядка. Поиск его производился путем вычисления средних времен выполнения каждой операции при обработке ПТ и построением соответствующей гистограммы (рис. 3.8). Исходя из рис. 3.8 наибольшее среднее время обработки принадлежит передаче данных. Поэтому новым параметр порядка определен сетевым уровнем. Основной целью следующей итерации следует считать *оптимизацию процесса передачи данных*.

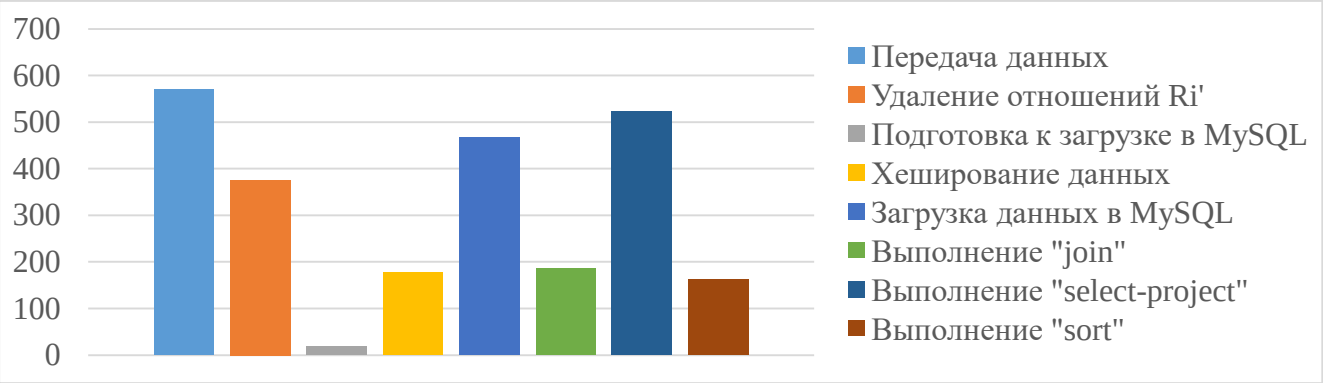


Рис. 3.8. Среднее время обработки операций для VБД = 120 GB (1 итерация)

Несмотря на то, что данная версия не может конкурировать со Spark, она может быть применена для обработки запросов в специфических конфигурациях. Например, поскольку выделенный узел HASH позволяет группировать узлы JOIN за собой, возможно применение нескольких таких групп в целях масштабирования системы на уровне JOIN. Это позволит Clusterix-N обрабатывать больше запросов с высоким удельным весом операций *join* и обслуживать больше пользователей, но потребует существенного увеличения количества узлов. Другим вариантом для применения данной версии может служить использование ее в организациях, которые не в состоянии оснастить все узлы своего кластера GPU-ускорителями. Таким организациям потребуется оснастить GPU-ускорителем лишь один узел.

3.3. Вторая итерация *IS*-моделирования Clusterix-N

Основная идея, положенная в основу второй итерации и позволяющая надеяться на успех, состоит в реализации операций динамического сегментирования промежуточных/временных отношений (хеширования) в модулях IO и JOIN с передачей хешированных данных напрямую между исполнительными узлами (минуя MGM).

Отказ от выделенного хеширующего узла HASH и BUF MGM и перенос их функционала на исполнительные узлы должен уменьшить объем передаваемых данных и, тем самым, повысить производительность системы. Вместо передач IO – MGM – HASH – JOIN – … – HASH – JOIN – … – HASH – MGM – SORT предлагается использовать передачи IO – JOIN – … – JOIN – … – JOIN – SORT. В первом случае на запрос с 2 операциями *join* и 3 отношениями приходится 14 сетевых передач, во втором случае всего 5. Вследствие такого переноса при хешировании используются GPU ускорители на всех доступных узлах, что, вместе с уменьшением объема данных для хеширования в каждом отдельном узле, должно уменьшить время хеширования в целом.

Передача данных между множеством узлов позволяет использовать особенность работы сетевых коммутаторов: связывать два узла напрямую при передаче данных. При этом два связанных узла не влияют на передачу данных между другими узлами. Это позволит увеличить общую пропускную способность сети в системе. При использовании сети GigabitEthernet, в конфигурации 3 IO и 3 JOIN общая пропускная способность сети должна увеличиться в 3 раза (по количеству узлов на уровне обработки), т.е. до 3 Gb/s.

Модификация Clusterix-N (2 итерация). Отказ от выделенного хеширующего узла HASH и BUF MGM потребовал существенной модификации Clusterix-N. Реализация хеширования, разработанная для модуля HASH, была адаптирована и перенесена в программные модули IO и JOIN. При этом реализован новый режим работы системы с прямой передачей данных между исполнительными узлами. Организация такого режима потребовала переработки модуля управления MGM. Теперь он занимается исключительно управлением: раздает задачи подчиненным узлам, отслеживает их статусы, тем самым выполняя координацию работы всей системы в целом.

Принцип работы такой модификации Clusterix-N иллюстрирует рис. 3.9, где MGM – управляющий модуль; IO, JOIN, SORT – исполнительные модули. Более детально работа данной итерации показана на новой диаграмме функционирования системы (приложение В.2). Здесь сохранены все обозначения из итерации 1, блоки левого столбца диаграммы образуют внешний конвейер.

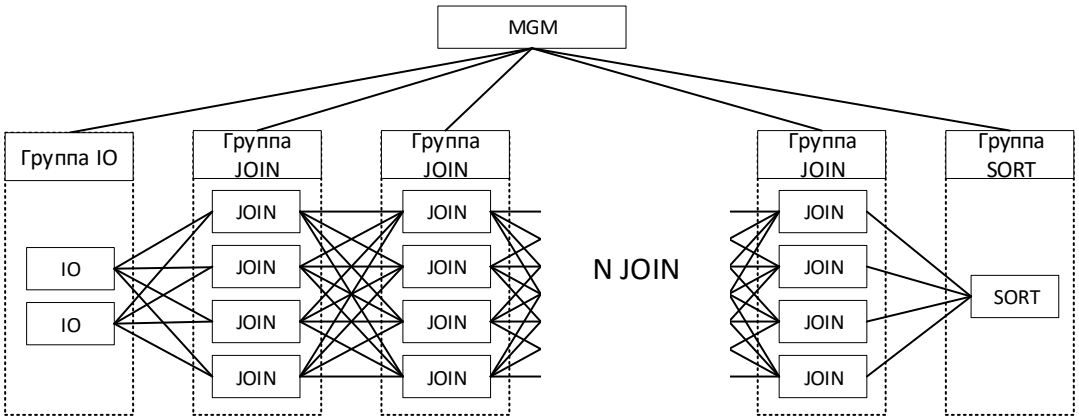


Рис. 3.9. Взаимодействие программных модулей Clusterix-N (итерация 2)

При поступлении очередного запроса MGM передает подзапросы *select* во все подключенные IO с указанием адресов и количества ядер JOIN для хеширования. В процессе работы IO отчитываются MGM о текущем статусе обработки запросы. Как только MGM получает отчет от всех IO о полной передаче данных в JOIN, необходимых для выполнения первого подзапроса *join*, MGM дает команду запуска операции *join* для двух хешированных отношений всем подключенным JOIN при этом сообщая им адреса других JOIN и количества их ядер для хеширования. Модули JOIN (по аналогии с IO) отчитывается о выполнении очередного подзапроса *join* и завершении передачи данных. MGM, получив такой отчет, дает команду на запуск следующего *join* до тех пор, пока в очереди на выполнение не останется последний подзапрос *join*. Его запуск производится особым образом. Вместо адресов JOIN с командой на запуск передается адрес SORT, с пометкой передачи данных без хеширования, чтобы узлы JOIN при выполнении последнего подзапроса не тратили время на хеширование результата, а сразу передавали его в SORT. После завершения последнего *join* MGM отправляет команду на запуск операции *sort* в соответствующий модуль. Результат *sort* передается в MGM и от него пользователю.

Модуль IO, как и прежде, выполняет операцию *select-project* для одного отношения параллельно на множестве доступных процессорных ядер с получением

набора блоков результата. Но вместо передачи в BUF MGM блоки подвергаются хешированию с ускорением на GPU [98] и передаются сразу в определенные узлы JOIN. Поблочная выборка позволяет совместить во времени все 3 операции (рис. 3.10): после формирования одного блока запускается задача выборки следующего, а результат передается в очередь на хеширование, в то же время хешированные блоки поступают в очередь на передачу.

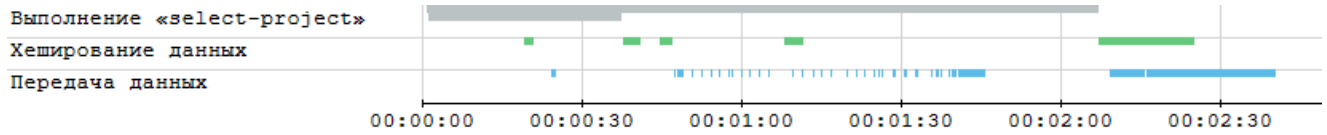


Рис. 3.10. Работа модуля IO (итерация 2)

Модуль JOIN полностью повторяет алгоритм работы из предыдущей модификации. Отличие заключается лишь в обработке результата *join*. Теперь он подвергается хешированию с ускорением на GPU [98] и передаче в узлы JOIN для следующей операции *join* или передаче в SORT с аналогичным IO совмещением операций, как это показано на рис. 3.11.

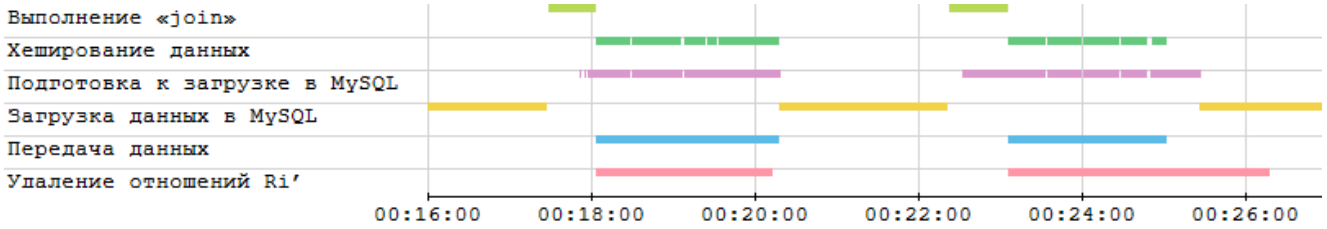


Рис. 3.11. Работа модуля JOIN (итерация 2)

Модуль SORT не претерпел никаких существенных изменений. Он все также использует стратегию «запрос на ядро» [17] и передает результат в MGM. Единственное изменение в его работе – это получение данных от узлов JOIN, а не из BUF MGM.

Экспериментальное исследование полученной модификации Clusterix-N производилось на платформе GPU-кластера в конфигурации (рис. 3.12): 2 узла IO, 4 узла JOIN, и 1 узел MGM, совмещающий модули MGM и SORT.

Результат эксперимента: время выполнения ПТ – 14,5 часа, т.е. общее время обработки ПТ сократилось всего на ~26% по сравнению с предыдущей модификацией Clusterix-N. Этого явно недостаточно для того, чтобы говорить о возможной конкуренции со Spark.

Более детальный анализ показал, что время передач по сети уменьшилось в ~3 раза, операции хеширования ускорились незначительно, операции *join* ускори-

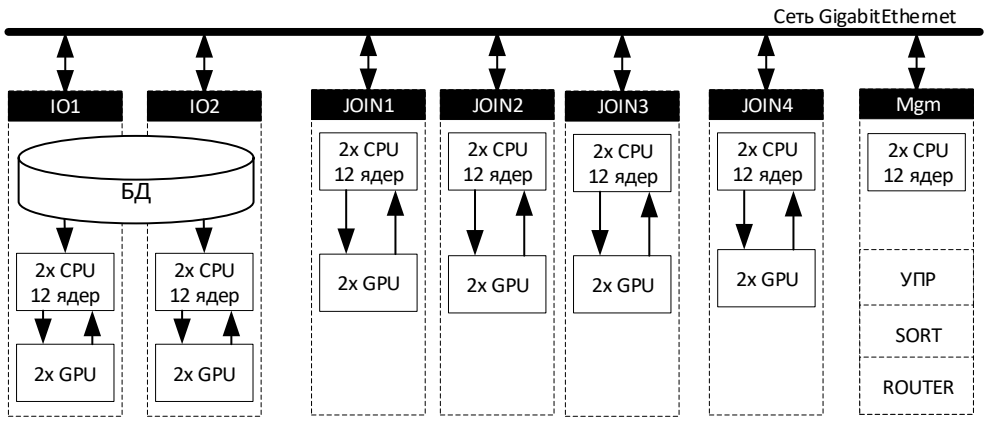


Рис. 3.12. Конфигурация GPU-кластера для Clusterix-N (итерация 2)

лась в 1,6 раза (за счет добавления еще одного узла и уменьшения объема данных в каждом узле). Операции хеширования показали незначительное ускорение. Объясняется это выполнением их в разное время на разных узлах. Пример такого поведения представлен в отрывке визуализации работы на рис. 3.13, где видно, что на узлах IO хеширование данных производится не синхронно.

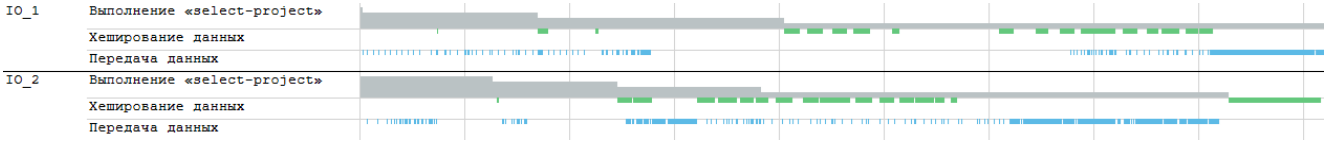


Рис. 3.13. Отрывок визуализации выполнения запроса №9 теста TPC-H в IO Clusterix-N (итерация 2)

Ранее самой длительной операцией обработки запросов была передача данных по сети. После внесенных изменений получена новая гистограмма средних времен обработки отдельных операций (рис. 3.14), которая показывает существенное уменьшение (в 3 раза) среднего времени передачи данных по сравнению с предыдущей итерацией. Теперь самой длительной операцией оказалась *select-project*. Это и есть новый параметр порядка.

Как и предыдущая, новая итерация неконкурентоспособна со Spark. Но она может быть применена в конфигурациях с асимметричным распределением работ по уровням IO и JOIN. Это может быть полезно, например, при обработки данных небольшого объема, но с большим количеством вычислительных операции и операций *join*. Кроме того, новая версия Clusterix-N менее требовательна к интерконнекту по сравнению с предыдущей версией. Однако, передача данных напрямую между узлами в условиях конвейерной обработки, по сравнению с первой итерацией, делает невозможным одновременную обработку нескольких запросов на каждом уровне.

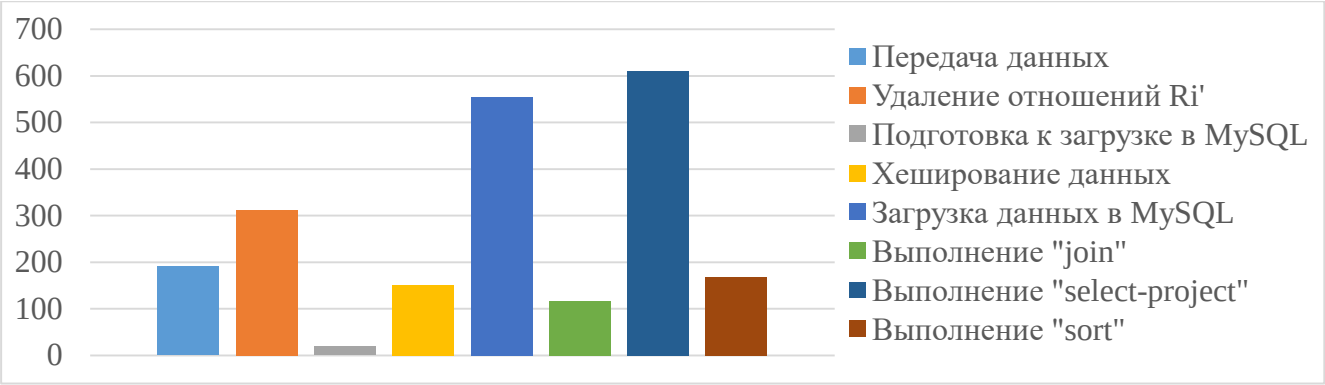


Рис. 3.14. Среднее время обработки операций для $V_{БД}$ = 120 GB (2 итерация)

3.4. Третья итерация *IS*-моделирования Clusterix-N

Наиболее простой способ добиться ускорения выполнения операции *select-project* – уменьшить объем обрабатываемых данных. Уменьшения объема данных, обрабатываемых на каждом узле, можно добиться увеличением количества узлов (за счет хеширования данных по узлам на каждом из них будет меньшая порция данных). Обеспечение лучшего совмещения работ (как и увеличение количества узлов для приведенной операции) возможно с применением конфигурации «совмещенная симметрия» [18], которая предполагает размещение на одном узле сразу двух модулей: IO и JOIN.

Для запуска IO и JOIN совместно на одном узле требуется небольшая модификация системы. Во-первых, необходима настройка количества занимаемых процессорных ядер. Во-вторых, необходима настройка для связывания каждого модуля со «своим» GPU-ускорителем. Поскольку хеширование на одном GPU-ускорителе производится в 2 раза быстрее передачи по сети, то для одного модуля вполне достаточно одного ускорителя. Какой-либо модификации архитектуры или диаграммы функционирования не требуется, они остаются прежними (как в итерации 2).

Эксперимент в конфигурации «совмещенная симметрия» проводился на платформе GPU-кластера (рис. 3.15).

Результаты эксперимента в сравнении со Spark представлены в табл. 3.2, где $V_{БД}$ – объем БД в эксперименте, T – время выполнения всего ПТ, M – среднее время ожидания результата, σ – среднеквадратическое отклонение.

Из таблицы видно, что при $V_{БД}$ = 60 GB время выполнения ПТ в Clusterix-N и в Spark примерно одинаковое, но M у Clusterix-N вдвое выше, а σ больше ~ в 14 раз.

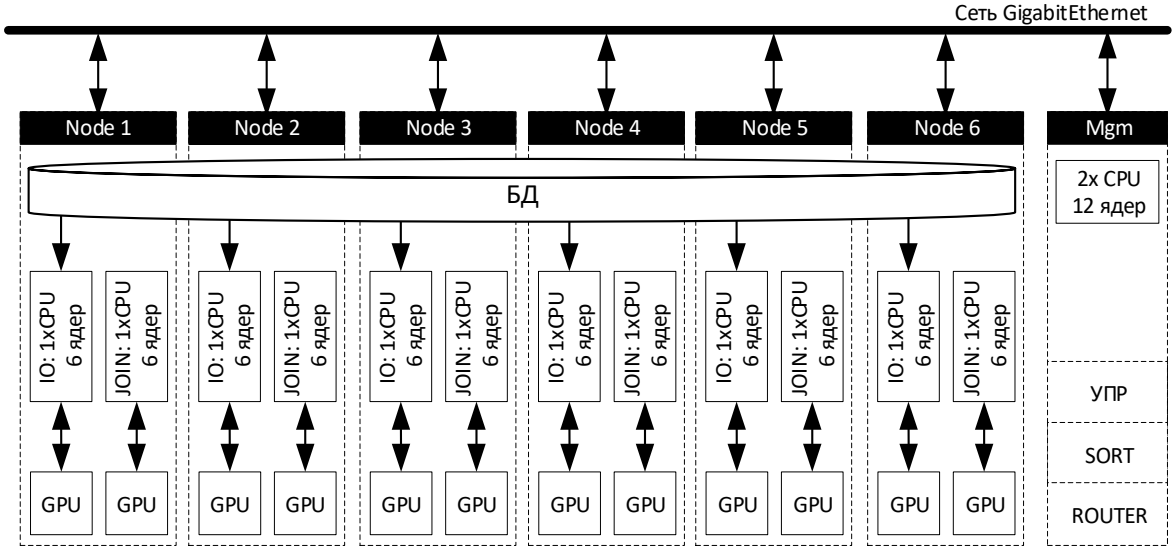


Рис. 3.15. Взаимодействие программных модулей Clusterix-N в конфигурации «совмещенная симметрия» (итерация 3)

Таблица 3.2. Сравнительные результаты эксперимента в конфигурации «совмещенная симметрия»

	Clusterix-N		Spark		Отношение Spark/Clusterix-N	
$V_{БД}$	60GB	120GB	60GB	120GB	60GB	120GB
T, мин	181,4	455,8	200,0	260,6	1,10	0,57
M, мин	5,6	14,3	2,4	3,1	0,43	0,22
σ, мин	7,3	15,7	0,5	0,9	0,07	0,06

При $V_{БД}$ = 120 GB Clusterix-N уступает ~ в 2 раза по *T*, ~ в 3 раза по *M* и ~ в 16 раз по σ. Такое большое различие по *M* можно объяснить конвейерной обработкой: запрос должен пройти все стадии конвейера. Улучшить параметр *M* в Clusterix-N можно либо сокращением количества стадий конвейера, либо дальнейшим ускорением наиболее длинных стадий (что было произведено в итерациях 1-3). Большие показатели σ говорят о существенном разбросе времени обработки отдельных запросов.

По условию, принятому при постановке задачи, рассматриваемые СУБД являются многопользовательскими системами с пакетной обработкой запросов. Большие показатели σ и *M* не считаются недостатком при эксплуатации системы в многопользовательском (пакетном) режиме. В этом случае администраторы информационных систем заинтересованы в обеспечении максимальной пропускной способности этих систем, т.е. в минимизации времени пакетной обработки. Время ожидания отдельным пользователем ответа на свой запрос их интересует в меньшей степени.

Средние времена выполнения отдельных операций при объеме БД 120GB представлены в табл. 3.3. Гистограмма средних времен обработки отдельных операций представлена на рис. 3.16.

Таблица 3.3. Среднее время выполнения отдельных операций в Clusterix-N для $V_{БД}$ = 120GB

	1 итерация α, сек	2 итерация β, сек	3 итерация γ, сек	$\frac{\alpha}{\beta}$	$\frac{\beta}{\gamma}$	$\frac{\alpha}{\gamma}$
Передача данных	569,77	190,82	133,62	2,99	1,43	4,26
Удаление отношений R_i' R_{Bj}	375,27	311,87	180,03	1,20	1,73	2,08
Подготовка к загрузке	18,90	19,01	22,40	0,99	0,85	0,84
Хеширование данных	177,03	151,05	93,61	1,17	1,61	1,89
Загрузка данных в MySQL	466,64	555,30	447,28	0,84	1,24	1,04
Выполнение «*join*»	186,09	116,06	89,09	1,60	1,30	2,09
Выполнение «*select-project*»	522,73	611,32	159,86	0,86	3,82	3,27
Выполнение «*sort*»	162,61	168,97	178,42	0,96	0,95	0,91

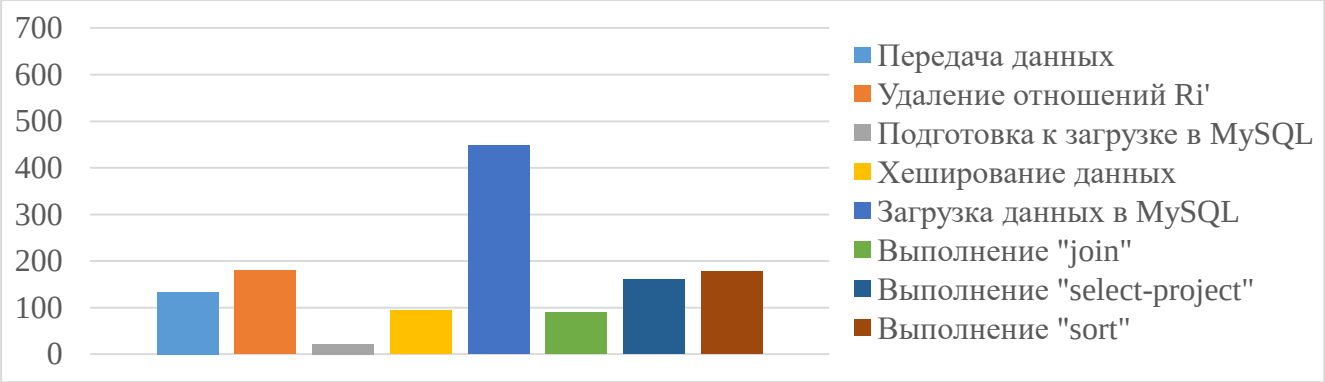


Рис. 3.16. Среднее время обработки операций для $V_{БД}$ = 120 GB (3 итерация)

Сравнение этих времен показывает, что в конфигурации «совмещенная симметрия» самой длительной операцией осталась загрузка данных в MySQL. Следующими наиболее «затратными» операциями стали удаления данных из JOIN и *sort*.

За счет уменьшения времени обработки *select-project* достигнуто существенное ускорение работы системы. Сильное уменьшение времени выполнения операций *select-project* можно объяснить особенностью работы драйвера параллельной выборки из MySQL, реализованного в Clusterix-N. На каждом ядре запускается запрос с добавлением к нему в конце конструкции «*LIMIT P,N*». Она заставляет MySQL выдавать результат в количестве не более *N* строк, начиная со строки *P*. В случае выполнения *select-project* с условиями по неиндексированным полям MySQL выполняет полное сканирование обрабатываемого отношения для каждого модифицированного запроса. В тесте TPC-H поля для выборки не индексированы. Поэтому с уменьшением объема обрабатываемых данных существенно ускоряется операция *select-project*.

Визуализации выполнения ПТ в Clusterix-N (рис. 3.17) подтверждает существенное уменьшение времени на обработку операций *select-project* за счет хеши-

рования БД по 6 узлам. Замечаем, что модули IO не всегда заняты обработкой запросов, т.е. один из CPU (6 ядер) простаивает довольно продолжительное время. В тоже время существенную часть работы в JOIN занимает удаление и загрузка данных. SORT, в силу малого объема данных, быстро обрабатывает свои запросы. Но в ПТ имеется запрос, который требует большого объема данных на этапе *sort*. Именно он изображен на визуализации в виде длинной линии и именно его обработка увеличила общее время выполнения ПТ на 40 минут.

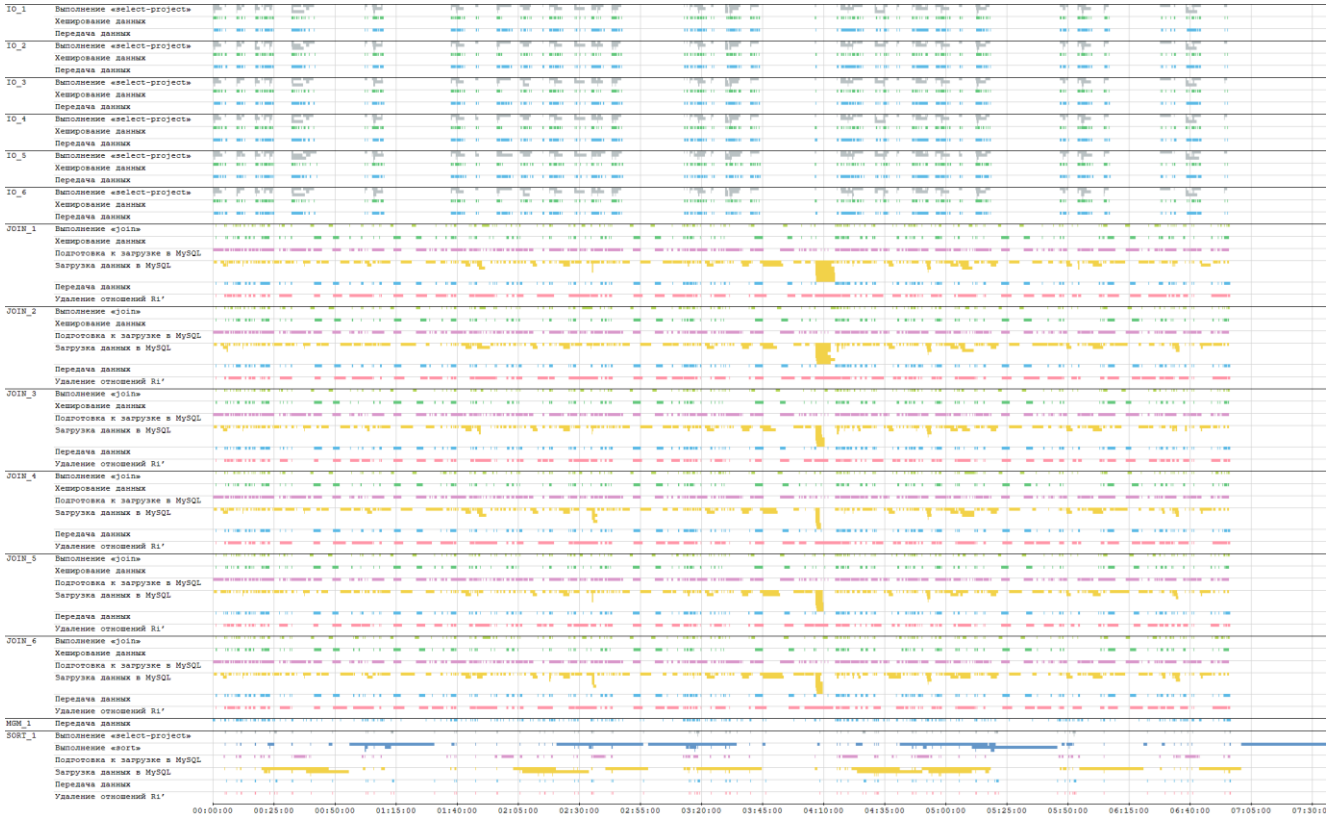


Рис. 3.17. Визуализация выполнения ПТ в конфигурации «совмещенная симметрия», $V_{БД}$=120GB

3.5. Четвертая итерация *IS*-моделирования Clusterix-N

Загрузку данных в MySQL для модулей JOIN можно ускорить увеличением числа ядер, на которых реализуются эти модули. Как уже было отмечено, исходя из рис. 3.17, модули IO далеко не всегда заняты обработкой запросов. Возникает естественное сомнение в целесообразности конвейеризации операций *select-project* и *join* в данном случае. Не лучше ли будет их последовательная реализация на одном ядре с загрузкой этими операциями всех ядер кластера? В таком случае модули IO будут работать без простоев, а операции загрузки в MySQL будут значительно ускорены, что должно снизить значения M и σ. Но не снизит ли эффективность та-

кое нарушение внешней конвейеризации? Для получения ответа на этот вопрос выполнено предварительное исследование при минимальных доработках системы: запросы *select-project* поступают в работу только в случае отсутствия выполняемых запросов *join*. Выполненные доработки не отразились на архитектуре системы, но диаграмма функционирования (приложение В.3) претерпела ряд изменений. Если ранее весь левый столбец представлял собой внешний конвейер с соответствующими обратными связями для *select-project*, *join* и *sort*, то теперь обратная связь завершения операции *join* ведет к запуску новой операции *select-project* (в случае необходимости выполнения еще одной операции *join* для запроса) или к запуску *sort* (в случае завершения обработки). Поэтому внешний конвейер для этой итерации строится из блоков левого столбца, но блоки «Передача запроса *select-project* для одного отношения в модули IO» и «Запуск операции *join*» объединены в один шаг.

Теперь все 72 ядра исполнительной части кластера выполняют одинаковую работу, но с разными данными. В каждом узле используется по одному GPU для целей хеширования результатов работы модулей IO и JOIN. Экспериментально полученные гистограммы представлены на рис. 3.18. Из него видно, что выполнение операций *sort* серьезно ускорены по сравнению с предыдущей итерацией. Это стало возможным за счет смены «движка» MySQL с MyISAM на MEMORY для модуля SORT.

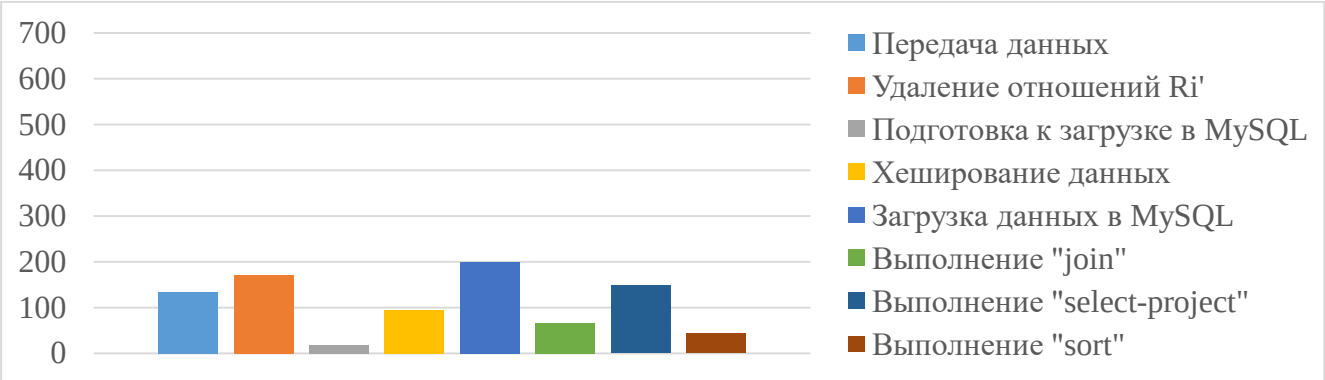


Рис. 3.18. Среднее время обработки операций для $V_{БД}$ = 120 GB (начало 4 итерации)

Им отвечают данные таблиц 3.4 и 3.5. Они подтверждают сделанные прогнозы и опасения.

Несмотря на серьезное ускорение операции загрузки в MySQL, уменьшение M и σ, время обработки ПТ увеличилось ~ на 23% в сравнении с итерацией 3.Такова плата за частичное нарушение конвейеризации: теперь внутренние конвейеры *se-*

Таблица 3.4. Сравнительная оценка ускорений для итераций 3 и начала 4 для $V_{БД}$ = 120GB

	3 итерация, сек	Начало 4 итерации, сек	Отношение и.3 /н.4
Передача данных	133,62	133,36	1,00
Удаление отношений R_i' R_{Bj}	180,03	170,47	1,06
Подготовка к загрузке	22,40	17,80	1,26
Хеширование данных	93,61	94,62	0,99
Загрузка данных в MySQL	447,28	198,75	2,25
Выполнение «*join*»	89,09	66,83	1,33
Выполнение «*select-project*»	159,86	148,17	1,08
Выполнение «*sort*»	178,42	43,03	4,15

Таблица 3.5. Сравнительные оценки по T, M и σ итераций 3 и начала 4 для $V_{БД}$ = 120GB

	3 итерация, сек	Начало 4 итерации, сек	Отношение и.3 /н.4
T	455,8	560,6	0,81
M	14,3	8,2	1,74
σ	15,7	6,4	2,45

lect-project и *join* выполняются в одной стадии внешнего конвейера, увеличение продолжительности которой ведет к снижению пропускной способности системы. Несколько поправить положение можно дальнейшим ускорением *select-project* и ряда других операций.

При неизменной платформе, предлагаемый подход к повышению эффективности итерации 4 состоит, во-первых, в переходе в IO от хеширования БД по узлам к хешированию по ядрам. Это должно значительно ускорить *select-project*. Во-вторых, – в переходе на последнюю, более совершенную версию MySQL 8.0 [101]. Можно ожидать, что ее применение ускорит выполнение и ряда других операций.

Программная реализация предлагаемых переходов вызвала необходимость серьезной доработки драйвера СУБД и модуля IO. Результаты проведенного эксперимента иллюстрируют гистограммы рис. 3.19 и данные таблиц 3.6, 3.7. Теперь

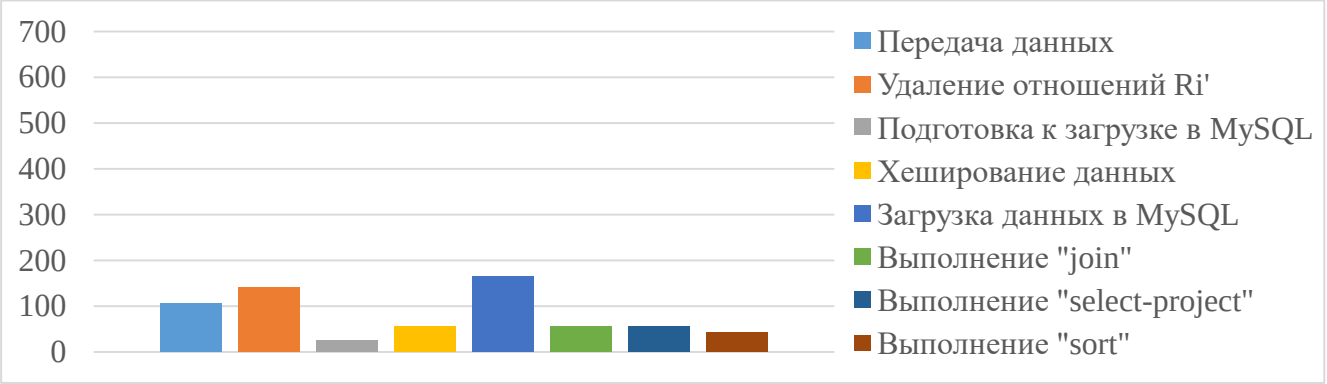


Рис. 3.19. Среднее время обработки операций для $V_{БД}$ = 120 GB (4 итерация)

оценки для Clusterix-N и Spark сравнимы в бóльшей степени. Улучшение по отношению к итерации 3: по T – на 20%, по M и σ – в 2,27 и 2,8 раз соответственно.

Таблица 3.6. Сравнительная оценка ускорений для итераций 3 и начала 4 для $V_{БД}$ = 120GB

	Начало 4 итерации, сек	4 итерация, сек	Отношение н.4 /и.4
Передача данных	133,36	106,12	1,26
Удаление отношений R_i' R_{Bj}	170,47	140,91	1,21
Подготовка к загрузке	17,80	24,15	0,74
Хеширование данных	94,62	56,43	1,68
Загрузка данных в MySQL	198,75	164,01	1,21
Выполнение «*join*»	66,83	54,77	1,22
Выполнение «*select-project*»	148,17	55,40	2,67
Выполнение «*sort*»	43,03	42,90	1,00

Таблица 3.7. Сравнительные результаты итерации 4 со Spark

	Clusterix-N		Spark		Отношение Spark/Clusterix-N	
$V_{БД}$	60GB	120GB	60GB	120GB	60GB	120GB
T, мин	172,4	403,8	200,0	260,6	1,16	0,65
M, мин	3,2	6,3	2,4	3,1	0,76	0,49
σ, мин	3,0	5,6	0,5	0,9	0,17	0,16

3.6. Выводы по главе 3

Проведенное ***IS***-моделирование процесса синтеза Clusterix-подобных систем показало, что использование регулярного плана обработки запросов, при соответствующей архитектурной и программно-алгоритмической разработке консервативных СУБД кластерного типа класса BigData, показывает результаты, сравнимые по эффективности с лучшими зарубежными открытыми системами.

Последняя версия системы Clusterix-N, разработанная на четвертой итерации ***IS***-моделирования, сравнима по производительности со Spark. Выполненные разработки могут быть использованы отечественными организациями с ограниченным бюджетом. Причем, поскольку последняя версия Clusterix-N включает в себя наработки предыдущих итераций, то она может быть легко подстроена под определенные задачи обработки данных и аппаратные платформы организаций в соответствии с их требованиями/бюджетом.

Но как следует из гистограммы рис. 3.19, процессы передачи данных, удаления отношений и загрузки данных в MySQL остаются достаточно медленными.

Следует ожидать значительного ускорения интерконнекта при переходе на работу со сжатыми базами данных (см. раздел 5.1). Это дополнительно повысит эффективность, ибо увеличит объемы БД, допустимые для бездисковой обработки при заданном числе узлов.

В отношении процессов удаления отношений и загрузки данных, проведенное исследование дает основания полагать, что для MySQL существует принципиальное ограничение на достижимое ускорение этих процессов. Как показывает анализ динамики выполнения ПТ в итерации 4, все ядра системы полностью заняты лишь в начальные моменты загрузки. Далее система переходит к работе на одном ядре. PostgreSQL не позволяет организовать бездисковую работу системы. Единственный выход из положения видится в разработке специальных движков MySQL для ускорения указанных процессов.

Стоимость приобретения и ввода в эксплуатацию GPU-кластера, аналогичного использованному при проведении сравнительного эксперимента, составит не более 5 млн. руб. Версии (1–4) программной системы Clusterix-N(ew) помещены в открытый доступ [102], достаточно надежны и могут быть использованы заинтересованными организациями.

ГЛАВА 4. АЛЬТЕРНАТИВА СИСТЕМЫ CLUSTERIX-N ПРИ УМЕРЕННЫХ ОБЪЕМАХ БАЗ ДАННЫХ

В данной главе приведена выполненная разработка IT-технологии «PerformSys» [103] и отражены результаты экспериментов по обработке запросов в рамках этой технологии к консервативным БД разных объемов.

В разделе 4.1 демонстрируется архитектура и принципы работы PerformSys [104] в составе из 4-х программ: *server*, *router*, *balancer* и *client*, разработанных на языке C# для платформы .Net 3.5 с возможностью сборки и запуска на платформе Mono 2.10 [90] и новее. В качестве инструментальной СУБД используется MySQL. Поскольку программы .Net исполняются в виртуальной машине [105] и существует ее открытая реализация – Mono, то PerformSys может быть запущен в различных операционных системах (Windows, Linux, Solaris и др.).

Разделы 4.2 и 4.3 посвящены экспериментам на платформах SUN- и GPU-кластеров. Исследуется ускорение обработки потока запросов к БД консервативного типа небольших (единицы GB) и повышенных (десятки и сотни GB) объемов, эффективность стратегии «запрос на ядро +1» с *router* и без него, а также масштабируемость PerformSys.

Сравнительные выводы по результатам экспериментов для технологий Clusterix-N и PerformSys и даны в разделе 4.4.

4.1. Архитектура PerformSys

Архитектура PerformSys ориентирована на кластерные платформы и предполагает запуск всех программ пакета на выделенных узлах, но допускается и запуск в рамках одного узла. Работа осуществляется по стратегии «запрос на ядро + 1» [17], согласно которой на каждом исполнительном узле выполняется запросов по количеству процессорных ядер + один запрос ожидает в очереди. Дополнительный запрос в очереди исполнительного узла позволяет избежать простоя процессорного ядра после его освобождения во время передачи очередного запроса. Связь между программными модулями осуществляется по сети с использованием стека TCP/IP [106]. Передача данных осуществляется пакетами вариативной длины. За коррект-

ность сетевых передач отвечает специальный программный модуль, который используется для упаковки объектов в поток байт и извлечения объектов из него.

Каждая из 4 программ предполагает выполнение следующих функции:

– *Server* – принимает запросы от *balancer* или *router*, добавляет в свою очередь, пересылает запрос СУБД и ожидает выполнения запроса. После выполнения запроса, принимает ответ от сервера БД и пересылает его в *balancer* или *router*.
– *Balancer* – принимает запросы от *client* и размещает их в своей очереди запросов, следит за состоянием подчиненных серверов и выдает им запросы из своей очереди. Выполненные запросы пересылает клиентам.
– *Router* – выполняет перенаправление запросов от *balancer* в группу *server* по алгоритму *round-robin* (как в [18, 91]), но с учетом занятости каждого *server*, выполняет передачу результатов от подчиненных *server* в *balancer*.
– *Client* – предназначен для симуляции работы пользователей.

Организация взаимодействия программ показана на рис. 4.1. Здесь N – количество симулируемых клиентов, M – количество подключенных к *balancer* узлов с программой *router*, n_M – количество узлов с программой *server* подключенных к *M-router*, K – количество процессорных ядер в узле.

Работа PerformSys начинается с поступлением нового SQL запроса. Генерацией потока запросов занимается программа *client*, которая запускает симуляцию работы N пользователей. Каждый пользователь устанавливает соединение с *balancer* и начинают отсылать запросы на выполнение в соответствии с заданным сценарием. Сценарий задает правила генерации потока запросов на основе списка подготовленных запросов, функции выбора очередного запроса из списка.

Balancer получает запросы от пользователей, назначает им уникальные идентификаторы и сохраняет их в очередь на исполнение. Как только *balancer* определяет, что один из M подключенных *router* может принять очередной запрос, он передает ему первый запрос из своей очереди. *Router*, получив запрос от *balancer*, перенаправляет его в первый свободный *server*, из подключенных к нему. *Server* запускает полученный запрос на первом свободном ядре. Запуск запроса производится установкой нового соединения с инструментальной СУБД и передачей ей за-

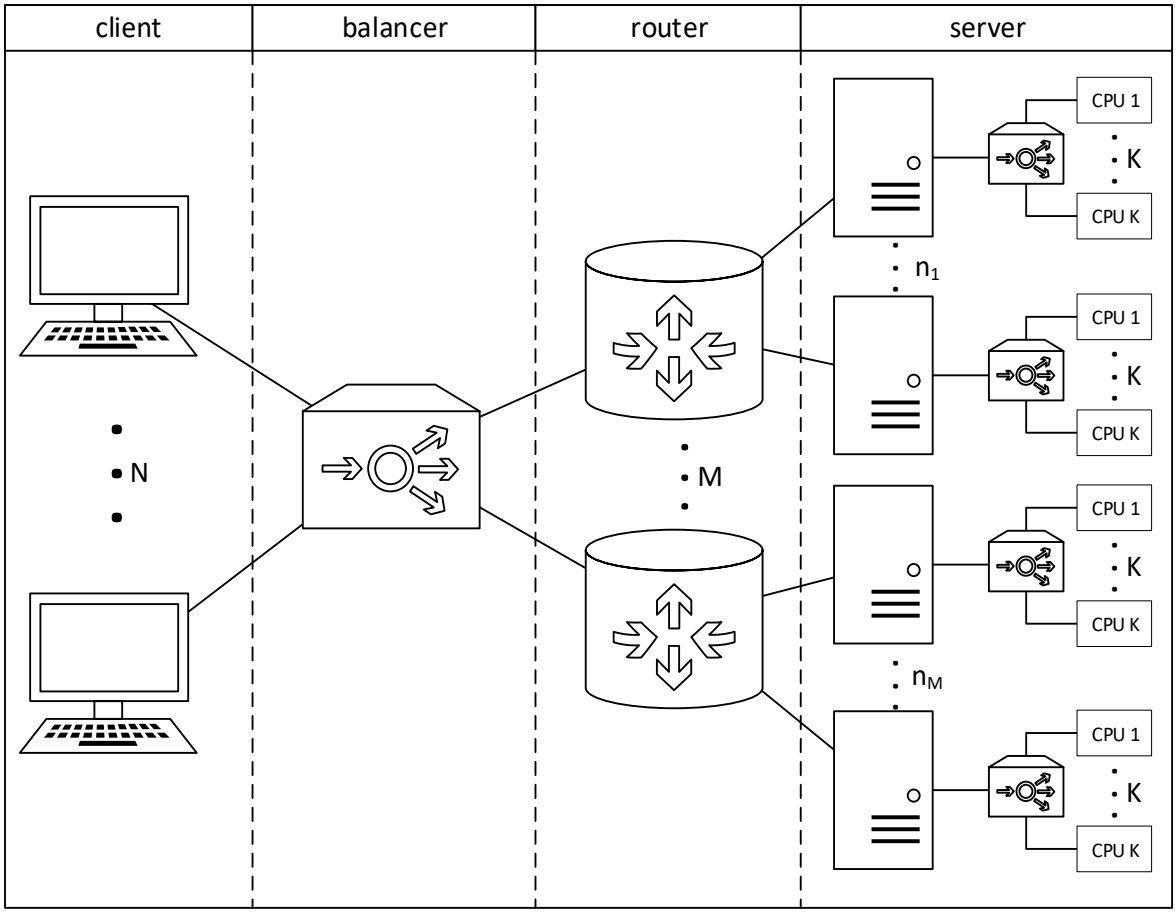


Рис. 4.1. Общая схема организации PerformSys

проса без каких-либо изменений. Соединение закрывается сразу после получения результата выполнения запроса. Если все *K* ядер заняты, то запрос записывается в очередь, а программе *router* высылается сообщение с отказом от приема новых. Если все подключенные к *router* узлы *server* выслали отказ от приема новых сообщений, то он высылает отказ в *balancer*, который прекращает передачу запросов до получения разрешения. Разрешение передачи нового запроса распространяется аналогично запрету: *server* → *router* → *balancer*. Условием разрешения передачи нового запроса является освобождение процессорного ядра на одном из узлов с *server*. Точно также передается результат выполнения запроса: от *server* в *router*, затем в *balancer*, а оттуда подключенному клиенту.

Дале более пробно рассмотрим организацию общих библиотек и алгоритмы работы программных модулей.

Общая библиотека программных модулей PerformSys. Общая библиотека `PerformSys.Common` используется всеми программами PerformSys. Она содержит программные модули журналирования, сетевого взаимодействия и различные помощники.

Модуль журналирования позволяет записывать в заданный файл и выводить на терминал события, происходящие в системе, с указанием даты/времени, уровня события и сообщения. Под уровнем события понимается его серьезность. В модуле журналирования предусмотрено 5 уровней сообщений. Они повторяют все уровни подсистемы журналирования из приложения А.1 кроме `TRACE`. Минимальным уровнем сообщения является `DEBUG`.

Вывод сообщений можно ограничивать, изменяя параметр минимального уровня сообщения. Так при минимальном уровне = `DEBUG` будут выведены и записаны все сообщения, а при уровне = `ERROR` будут выведены/записаны только сообщения уровней `ERROR` и `FATAL`.

Настройка модуля журналирования осуществляется в каждой программе отдельно в специальной секции конфигурационного файла `<Log> … </Log>`. Все конфигурационные параметры описаны в документации [107].

Записывать всю информацию в один файл для дальнейшего анализа неудобно. Поэтому модуль журналирования может писать сразу несколько файлов в разных форматах с выводом на терминал записанной информации. Наиболее удобной организацией журналирования оказалась схема с записью в двух форматах. В первом формате в файл записываются все сообщения в виде простого текста, а во втором – специфичная информация в формате CSV [108]. Формат CSV выбран в силу его удобства для последующей обработки в табличном процессоре. В файлы формата CSV записывается статистическая информация о работе системы: какой запрос, от какого пользователя, сколько времени выполнялся, сколько находился в очереди и т.д.

Сетевое взаимодействие организуется в стеке TCP/IP с применением сокетов. Передача данных осуществляется массивами байт переменной длины с добавлением концевика. Концевик – последовательность из 6 байт: `"\n\n\r\r\t\t"`. Массив байт на передачу формируется из передаваемых данных и заголовка. В заголовке указывается тип пакета, а данные кодируются в строку *base64* [109]. Строка *base64* формируется на основе сериализованных (преобразованных из исходного формата в формат, пригодный для хранения, передачи и восстановления данных) в XML [110] данных передаваемого пакета. Используются методы сериализации и

десериализации (преобразование объектов в XML и обратно), встроенные в .Net [111]. В результате достигается унификация передаваемых данных.

Для упрощения работы с передачей сообщений между узлами все данные скомпонованы в пакеты – структуры данных с возможностью их преобразования в массив байт и обратно с сохранением информации в определенных полях. В PerformSys используется всего 4 таких структур данных:

1. `DataBaseInfoPacket` – пакет передачи информации о БД из *server* в *balancer*. Он содержит информацию о таблицах и их размерах, а также о имени БД.
2. `DbAnswerPacket` – пакет передачи результата обработки запроса из *server* в *balancer*. Он содержит таблицу с результатом, которая, будет передана клиенту.
3. `DbRequestPacket` – пакет передачи запроса к БД из *balancer* в *server*. Предназначен для доставки SQL запроса в *server* и содержит только SQL запрос.
4. `ServerStatusPacket` – пакет передачи статуса *server* в *router* и *balancer*. Содержит флаг возможности передачи в сервер нового запроса.

За правильное получение и отправку пакетов отвечает специализированный программный модуль: `PacketTransmitHelper`. Он прослушивает сокет и считывает принятые байты в свой буфер, сканирует буфер на наличие концевика и формирует пакеты. Сформированные пакеты передаются другим программным модулям. Отправка организуется вызовом метода `Send` в `PacketTransmitHelper`. Этот метод выполняет преобразование пакета в массив байт и записывает их в сокет. Поскольку отправлять данные могут сразу несколько потоков, в `PacketTransmitHelper` реализована синхронизация передачи.

Помимо реализации модуля журналирования и сетевого приемо/передатчика, библиотека содержит помощники:

1. Модуль разбора параметров командной строки – преобразует входную строку в список ключей и значений, которые используются для переопределения конфигурационных параметров во время запуска программ.
2. Модуль сериализации/десериализации данных, их чтения и записи – предназначен для унифицированного чтения и сохранения конфигурационных файлов в формате XML.

Во всех программах сетевое взаимодействие и журналирование работы обеспечивается общей библиотекой программных модулей `PerformSys.Common`.

Программа server – программа внутриузлового балансировщика, которая устанавливает столько соединений с СУБД MySQL сколько имеется доступных ядер в узле и распределяет по ним запросы, полученные от *balancer* или *router*.

Балансировка нагрузки между ядрами стала возможна благодаря особенности СУБД MySQL. Она создает отдельный поток для каждого пользовательского сетевого подключения [112]. В каждом потоке запросы выполняются последовательно. Поэтому для параллельного выполнения *k* запросов необходимо установить *k* параллельных соединений.

Работа *server* начинается с подключения к СУБД. Если подключения установлено не было, то программа выдает сообщение об ошибке и завершает свою работу. В противном случае, она инициализирует сеть и пытается подключиться к *balancer* или *router*. Как только подключение установлено, по нему отправляется информация о БД (размеры таблиц) и сообщение о готовности принять очередной запрос.

При поступлении очередного запроса выполняется балансировка нагрузки между ядрами узлами по стратегии «ядро на запрос +1» согласно алгоритму.

АЛГОРИТМ РАБОТЫ ВНУТРИУЗЛОВОГО БАЛАНСИРОВЩИКА. Обозначим $C = \{C_i\}$, $i = \{\overline{1,K}\}$ – множество *i*-соединений с СУБД; $Cst = \{Cst_i\}$, $i = \{\overline{1,K}\}$ – множество статусов *i*-соединений с СУБД; B – буфер для одного очередного запроса; B^n – буфер идентификатора очередного запроса; $Qn = \{Qn_i\}$, $i = \{\overline{1,K}\}$ – множество идентификаторов запросов, выполняющихся в *i*-соединений с СУБД; Q – запрос, полученный от *balancer* или *router*; Q^n – идентификатор запроса, полученного от *balancer* или *router*;

1. $B := \varnothing$, $Q := \varnothing$, $Cst_i :=$ «свободно» для каждого $i = \{\overline{1,K}\}$
2. Установить K соединений с СУБД: $C_i :=$ новое соединение для каждого $i = \{\overline{1,K}\}$, где K – количество ядер в узле.
3. Если B – пустой, то перейти к шагу 5, иначе к шагу 4.
4. $Q := B$, $Q^n := B^n$, $B := \varnothing$ перейти к шагу 5.

5. Получить запрос и идентификатор запроса от *balancer* или *router*, записать запрос в Q, а его идентификатор в Q^n.
6. $c := 0$
7. Для первого $Cst_i =$ «свободно», $i = \{\overline{1,K}\}$ записать $c := i$.
8. Если $c = 0$ перейти к шагу 10, иначе к шагу 9.
9. Передать Q в C_c, $Qn_i = Q^n$, $Cst_c :=$ «занято», перейти к шагу 10.
10. $B := Q$, $B^n := Q^n$
11. Если хотя бы один $Cst_i =$ «свободно», $i = \{\overline{1,K}\}$, или B – пустой, то отправить сообщение о готовности принятия очередного запроса, иначе отправить сообщение о отказе приема новых запросов.
12. Перейти к шагу 2.

После завершения выполнения запроса в одном из соединений этому соединению присваивается статус $Cst_d =$ «свободно», где $d \in \{\overline{1,K}\}$ – номер соединения, а результат отправляется в *balancer* или *router*, с указанием идентификатора запроса Qn_d. В свободное соединение отправляется запрос из буфера (если он не пустой) или очередной полученный новый запрос.

В качестве инструментальной СУБД используется MySQL. Для связи с ней применяется MySQL Connector/NET [113] со специальной настройкой подключения.

Настройка программы производится с помощью конфигурационного файла `serverConfig.xml` [107] и параметров командной строки:

1. `--host` – адрес узла *balancer* или *router*.
2. `--port` – порт к которому будет осуществлено подключение.

Режим симуляции работы *server* полезен для отладки и проверки системы до запуска с реальной нагрузкой, т.к. она не требует больших вычислительных мощностей и позволяет запустить множество экземпляров программы *server* на одном маломощном компьютере. В режиме симуляции запросы не исполняются, вместо этого *server* просто ожидает заданное время миллисекундах. Время ожидания устанавливается в соответствии с номером запроса из конфигурации. В результате экспериментов установлено, что погрешность симуляции низкая по сравнению с

временем обработки запросов на чтение из состава теста TPC-H, которые могут выполняться минуты и часы при большом объеме БД.

Приостановка выполнения запроса на время ожидания позволяет оценить время обработки пакета запросов в разных конфигурациях системы, но не позволяет оценить объем передаваемого трафика. Поэтому, после приостановки формируется ответ с случайными данными. Размер ответа устанавливается в соответствии с номером запроса из конфигурации. Полученные данные отправляются как ответ на запрос.

Анализ работы программы возможен по журналам. Всего их ведется 3:

1. Общий журнал, куда записываются сообщения в процессе работы системы. Вывод производится как в файл, так на терминал.

2. Журнал статистики выполнения запросов, который содержит информацию о времени, идентификаторе запроса, номере клиента, номере запроса, времени его выполнения, длине ответа и длине очереди на момент пересылки ответа. Вывод производится только в файл формата CSV.

3. Журнал статистики изменения длины очереди содержит информацию об идентификаторе клиента и длине очереди на момент пересылки ответа. В журнал заносится каждое изменение длины очереди. Вывод производится в файл формата CSV без вывода на терминал.

В файл общего журнала выводятся все события в системе и ошибки. Он помогает определить на каком этапе начали возникать ошибки в работе системы и быстро их локализовать. Кроме того, он позволяет отслеживать состояние системы в реальном времени, простым наблюдением за сообщениями в терминале.

Оставшиеся два журнала служат для подсчета статистической информации после выполнения потока запросов. Их обработка производится в табличном процессоре после экспорта данных из формата CSV. В этих журналах поля `Глобальный идентификатор` и `Номер региона` предназначены для идентификации запросов из различных регионов. Но межрегиональная балансировка не была реализована, поэтому они не используются.

Программа router – программа балансировщика нагрузки между однотипными узлами *server* в составе одного кластера. Со стороны *balancer* она выглядит как один мощный *server*, а со стороны *server* как *balancer*, т.е. *router* является

прокси-сервером между *balancer* и *server*. *Router* был реализован в [96] как инструмент масштабирования PerformSys.

После запуска *router* подключается к *balancer* и ожидает подключения узлов *server*. При подключении нового узла, информация о нем (адрес и флаг возможности отправки запроса) сохраняется в списке подключенных узлов. Изменение флага возможности отправки очередного запроса в *server* выполняется «на лету» путем перехвата статусных сообщений с каждого подключенного узла *server*.

Список подключенных узлов позволяет программе *router* осуществлять балансировку нагрузки между узлами *server* без необходимости их прямого опроса. Сама балансировка осуществляется по следующему алгоритму.

АЛГОРИТМ РАСПРЕДЕЛЕНИЯ ЗАПРОСОВ ПО УЗЛАМ SERVER. Обозначим: $S = \{S_j\}$, $j = \overline{\{1, n_M\}}$ – множество подключенных узлов *j-server*, где n_M – количество подключенных узлов *server* в *M-router*; $St = \{St_j\}$, $j = \overline{\{1, n_M\}}$ – множество флагов разрешения отправки запроса в *j-server*;

1. $j := 0$
2. Получить запрос и идентификатор запроса от *balancer*, записать запрос в Q, а его идентификатор в Q^n.
3. $j := j + 1$
4. Если $j > n_M$, то $j := 1$
5. Если $St_j = true$, то перейти к шагу 6, иначе к шагу 7.
6. Передать Q и Q^n в S_j.
7. Если хотя бы один $St_j = true$ для $j = \overline{\{1, n_M\}}$, то отправить сообщение о готовности принятия очередного запроса, иначе отправить сообщение об отказе приема новых запросов.
8. Перейти к шагу 2.

Важной особенностью в работе *router* является сокрытие им оригинальных сообщений программы *server* о готовности принять очередной запрос. *Router*, получив очередное такое сообщение от *server*, выполняет просмотр всего списка подключенных узлов. Если в списке найдет хоть один узел, который готов принять

запрос ($St_j = true$ для $j = \{\overline{1, n_M}\}$), то в *blancer* будет отправлено сообщение о готовности принять очередной запрос, иначе – сообщение с отказом.

Конфигурация *router* производится изменением файла `routerConfig.xml` [107] и заданием параметров командной строки. Параметры командной строки полностью соответствуют таковым у *server*, в то время как файл конфигурации существенно отличается: отсутствует информация о подключении к БД, нет выбора режима работы и параметров симуляции, но появился новый параметр `Port` в секции `<Router>`. В него записывается номер порта, к которому ожидается подключение узлов *server*.

Журналирование производится аналогично *server*. Ведется 3 журнала: общий журнал, журнал статистики выполнения запросов и журнал статистики изменения длины очереди. Запись во все журналы идет так же, как и в *server*, но в журнал статистики выполнения запросов записывается время выполнения запроса в СУБД + время передачи данных по сети, а в журнал статистики изменения длины очереди записывается количество еще не перенаправленных запросов. Первый из них дает представление о вкладе задержек передачи по сети во время обработки запроса, а второй – о эффективности работы *router*.

Поскольку в *router* реализована балансировка нагрузки между узлами server, то его можно модифицировать до более самостоятельного модуля, организовав на нем очередь запросов. Это может быть полезно в работе с удаленными кластерами или в организации межрегионального доступа. Введение очереди может помочь уменьшить количество простоев, связанных с временем передачи запросов по сети.

Программа balancer – программа балансировки нагрузки между несколькими *server* или *router*. Основной задачей программы является перераспределение потока запросов от клиентов по подключенным узлам. Для выполнения этой задачи *balancer* использует круговой алгоритм распределения запросов с учетом готовности узлов и ведет очередь запросов, что в целом повторяет алгоритм распределения запросов по узлам *server*, реализованный в *router*, с добавлением к нему очереди.

Работа *balancer* начинается с инициализации очереди и запуска двух серверов. Первый сервер отвечает за связь с клиентами, а второй – за связь с узлами *router* или *server*. Оба сервера ведут списки подключений и могут отправлять сообщения подключенным программам по их идентификаторам. Идентификатором

клиента служит связка параметров: его *ip* адрес и порт, номер клиента. Номер клиента назначается при его подключении к *balancer*. Идентификатором подключения *router* или *server* служат параметры: *ip* адрес и порт, номера регионов. Номер региона идентифицирует БД, с которой может работать *server* или *router*. Клиентам не назначается номер региона, т.к. они работают с одним *balancer*, который «знает» номер своего региона. Такая идентификация позволяет однозначно адресовать клиентов, адресовать и использовать только те узлы *server* или *router*, которые располагают нужной БД и способны обработать клиентский запрос.

Когда система запущена и готова к работе (запущены все необходимые узлы *router* и *server*, запущен *balancer* и установлены все межузловые соединения) клиенты могут отправлять на обработку свои запросы. Каждый клиент отправляет не более одного запроса за раз и не может отправить новый запрос, пока не будет выполнен предыдущий. Поэтому размер очереди запросов в *balancer* не превышает количества подключенных клиентов.

Распределение клиентских запросов по исполнительным узлам *router* или *server* выполняется по следующему алгоритму.

АЛГОРИТМ МЕЖУЗЛОВОЙ БАЛАНСИРОВКИ НАГРУЗКИ. Обозначим: $R = \{R_r\}$, $r = \{\overline{1, M}\}$ – множество подключенных узлов *r-router* или *r-server*, где M – количество подключенных узлов *router* или *server*; $St = \{St_r\}$, $r = \{\overline{1, M}\}$ – множество флагов разрешения отправки запроса в *r-router* или *r-server*; $Q^c = \{Q_l^c\}$, $l = \{\overline{1, N}\}$ – множество запросов в очереди; $Q^{id} = \{Q_l^{id}\}$, $l = \{\overline{1, N}\}$ – множество идентификаторов запросов в очереди; Q – выбранный из очереди запрос; Q^n – идентификатор выбранного запроса;

1. $k := 0$
2. Если все $Q_l^c = \varnothing$ для $l = \{\overline{1, N}\}$, то перейти к шагу 9, иначе к шагу 3.
3. $Q := Q_0^c$, $Q^n := Q_0^{id}$
4. $Q_{i-1}^c := Q_i^c$ для $i = \{\overline{2, N}\}$, $Q_N^c := \varnothing$
5. $k := k + 1$
6. Если $k > M$, то $k := 1$

7. Если $St_k = true$, то перейти к шагу 8, иначе к шагу 9.
8. Передать Q и Q^n в R_k.
9. Перейти к шагу 2.

Результат обработки запроса передается клиенту по следующему алгоритму.

АЛГОРИТМ ДОСТАВКИ РЕЗУЛЬТАТА КЛИЕНТУ. Обозначим: $Cl = \{Cl_k\}$, $k = \{\overline{1, N}\}$ – множество подключенных клиентов; $Cl^{id} = \{Cl^{id}{}_k\}$, $k = \{\overline{1, N}\}$ – множество идентификаторов запросов клиентов Cl; Cl^n – номер клиента; Q^r – результат обработки запроса.

1. Получить результат обработки запрос и идентификатор запроса от *router* или *server*, записать результат в Q^r, а его идентификатор в Q^n.
2. Если $C_i^{id} = Q^n$ для одного из $i = \{\overline{1, N}\}$, то $Cl^n := i$, иначе перейти к шагу 4.
3. Отправить Q^r в Cl_{Cl^n}, перейти к шагу 5.
4. Нет клиента, отправившего запрос с идентификатором Q^n. Записать ошибку в журнал, а полученный результат уничтожить.
5. Перейти к шагу 1.

Обновление статусов подключенных узлов происходит при получении статусного пакета. По его обратному адресу и порту определяется номер подключенного узла $r = \{\overline{1, M}\}$, после чего в St_r записывается полученный статус. Аналогично определяется номер клиента $k = \{\overline{1, N}\}$ во время отправки им очередного запроса Q. Запрос Q добавляется в конец очереди Q^c, а его идентификатор ($Q^n :=$ [гобальный идентификатор]·10^8 + [номер региона]·10^6 + [номер клиента]) записывается в конец Q^{id}.

Конфигурация программы производится аналогично *router* и *server* через файл `rbnConfig.xml` [107]. Параметры командной строки в *blancer* не применяются.

Как и все рассмотренные ранее программы, *balancer* ведет 3 журнала: общий журнал, журнал статистики выполнения запросов и журнал статистики изменения длины очереди. Запись во все журналы идет аналогично *server*, но в журнал статистики выполнения запросов записывается время от отправки запроса на выполне-

ние до получения ответа, добавлено новое поле `Время ожидания` – время ожидания в очереди до отправки на обработку, удалено поле отметки времени. В журнале статистики изменения длины очереди в поле `длина очереди` записывается количество еще не распределенных запросов.

Программа client – программа симуляции работы подключенных клиентов в соответствии с заданным сценарием. Сценарий представляет собой цепочку действий с отметкой времени, выполняемую каждым симулируемым клиентом. С помощью сценария возможно задать периоды активности и неактивности клиентов. Настройка режима выбора очередного запроса выполняется в конфигурационном файле и применяется ко всем симулируемым клиентам. Настройка режима и сценария работы рассмотрены ниже в конфигурации программы.

Работа каждого симулируемого клиента организуется в отдельном потоке. В котором устанавливается соединение к *balancer*, отсылается очередной запрос согласно установленному методу в сценарии и ожидается результат обработки запроса. Как только результат получен, производится запись времени выполнения запроса в журнал и выбирается следующий запрос на обработку. Если запросов больше нет, то клиент закрывает соединение с *balancer*, и его поток уничтожается.

Конфигурация программы производится двумя файлами: `clientConfig.xml` и `queries.xml` [107]. Первый отвечает за общую конфигурация программы: конфигурацию журналирования, подключения к *balancer*, настройку режима выбора очередного запроса и сценария. Второй содержит список запросов, которые будут использованы в симуляции работы клиентов.

Шаг сценария с `Action=Sleep` не останавливает выполнение отправленных запросов, а лишь запрещает отправку новых. Отправленные запросы выполняются и результаты передаются клиентам. Аналогично шаг сценария с `Action=Stop` не завершает работу клиентов, а только запрещает отправку новых запросов. Как только все симулируемые клиенты получат результаты обработки запросов – работа программы *client* прекращается.

Все режимы выбора очередного запроса оперируют номерами запросов из `queries.xml` с последующей подстановкой самого запроса. Режимы `Random` и `Sequential` могут генерировать запросы бесконечно и требуют наличия последнего шага сценария с `Action=Stop`. Если в конфигурации нет такого шага, то он

будет добавлен автоматически. С другой стороны, режим выбора очередного запроса `FromList` не требует наличия шага сценария с `Action=Stop`, т.к. симуляция заканчивается сразу после исчерпания всего списка запросов.

Сценарии совместно со списком запросов могут применяться для симуляции различной нагрузки. Так, задав режим выбора очередного запроса `FromList` и указав единственный шаг сценария можно провести эксперимент с заданным множеством запросов. Или, задав режим выбора очередного запроса `Random`, и указав несколько шагов с чередованием `Action=Sleep` и `Action=Work`, в сценарии можно провести эксперимент связный с нагрузкой по периоду активности клиентов.

4.2. Платформа SUN-кластера. Исследование при объемах баз данных в единицы GB

Проверку эффективности работы PerformSys с БД объемом в единицы GB выполним экспериментально на платформе SUN-кластера. Платформа SUN-кластера представляет собой HPC-кластер, состоящий из 22 двухпроцессорных SMP-узлов со следующими характеристиками: 2 Quad-core Intel Xeon E5450 CPU/1,87GHz/32GB, 64-битная ОС Linux Red Hat Enterprise 6 с установленной СУБД MySQL 5.6 в специальной конфигурации (листинг Б.1). Интерконнект между узлами – GigabitEthernet / Infiniband 4X (20Gbps DDR) с 24-портовыми коммутаторами Cisco. Дисковая подсистема узла – SAS диски XRB-SS2CD-146G10KZ с пропускной способностью 300 MB/s и объемом 172 GB.

Представительский тест (ПТ), по аналогии с [18], формировался на основе теста TPC-H [11], который включает генератор базы данных `dbgen` и генератор запросов `qgen`. После генерации БД объемом 5GB получены следующие текстовые файлы с разделителями `'|'` и суммарным объемом 5.1 GB:

`customer.tbl, orders.tbl, part.tbl, supplier.tbl, lineitem.tbl, nation.tbl, partsupp.tbl, region.tbl`.

Название каждого файла соответствует отношению в схеме TPC-H, показанной на рис. 4.2.

Здесь стрелками показаны связи между отношениями, ключевые поля имеют окончание KEY, а под названием таблиц написано количество строк в зависимости от `SF` – scale factor, коэффициент роста БД, который задается параметром `-s` в

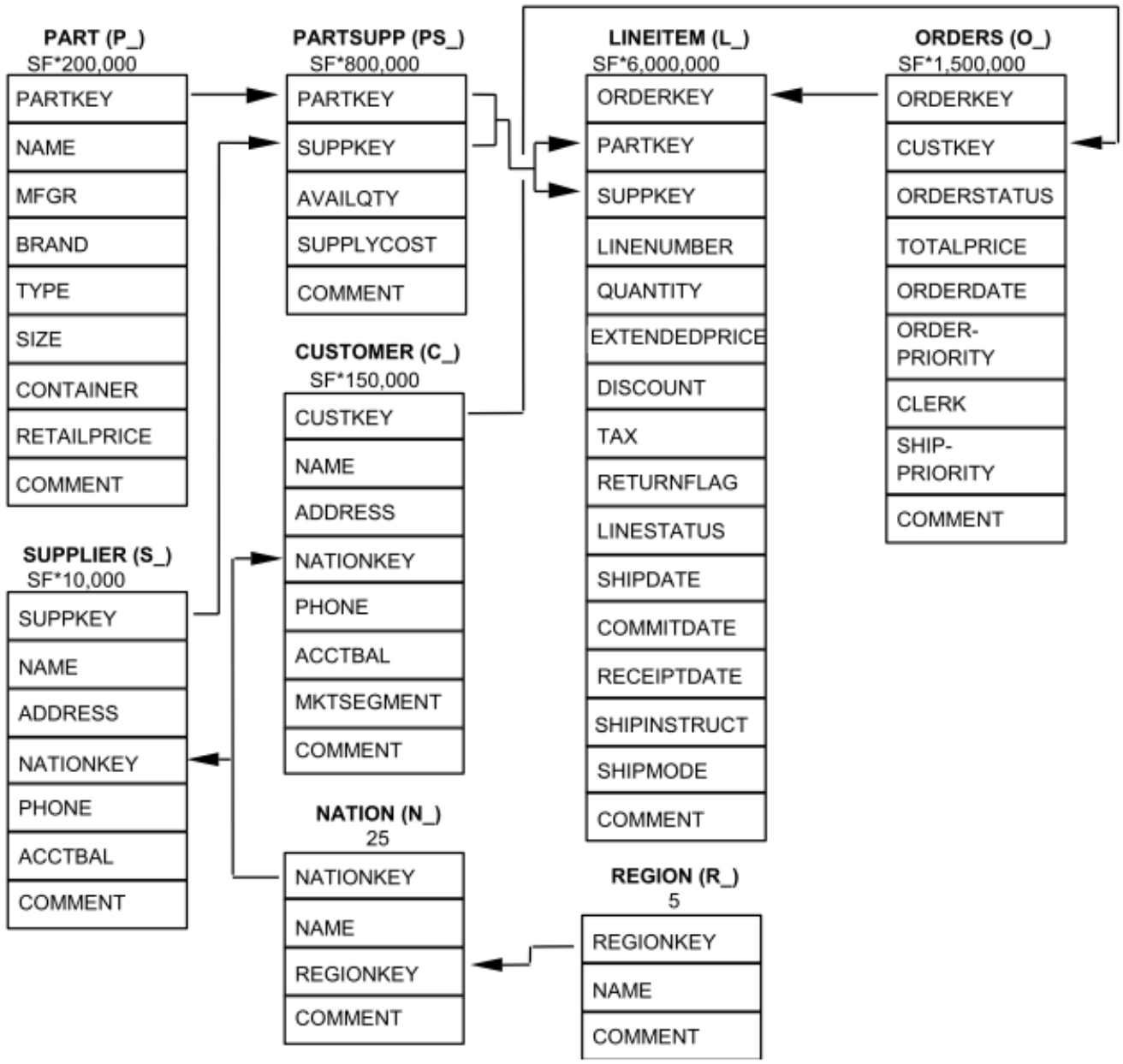


Рис. 4.2. Схема БД теста TPC-H

`dbgen`. Загрузка файлов выполняется специально разработанным пакетом команд (отрывок которого представлен в листинге Б.7) на основе предложенной схемы БД (файл `dss.ddl` в составе теста) и индексов (файл `dss.ri` в составе теста), который создает представленную схему БД без индексов, выполняет загрузку всех текстовых файлов в соответствующие отношения, создает индексы и связи.

После завершения загрузки данных в СУБД оказалось, что размер данных в созданных отношениях составляет 6.1 GB, а объем индексов 2.1 GB. Таким образом общий объем БД на диске составил 8.2 GB.

Тест TPC-H предлагает 22 запроса, но, т.к. к консервативным БД допускаются запросы «только на чтение», то из них будем использовать в ПТ только 14

запросов без операций записи под номерами от 1 до 14. Соответственно генератором запросов было создано 14 запросов. Полученные запросы были занесены в файл `queries.xml` программы *client* с соответствующими номерами.

Вместо предлагаемых в Throughput Test [11] перестановок запросов для ПТ был предложен набор из 1000 запросов, сгенерированных по равномерному закону на множестве из 14 запросов, выбранных из теста TPC-H. Основной причиной отказа от предложенных перестановок было желание получить тестовую нагрузку наиболее близкую к реальной.

Постановка эксперимента. Случай без router. SUN-кластер обслуживает N=50 клиентов, используя $n = \overline{1,7}$ исполнительных улов *server* (рис. 4.3). Запросы клиентов перенаправляются к определенному исполнительному узлу через *balancer*. Для распределения запросов между ядрами используется внутриузловой балансировщик в *server*.

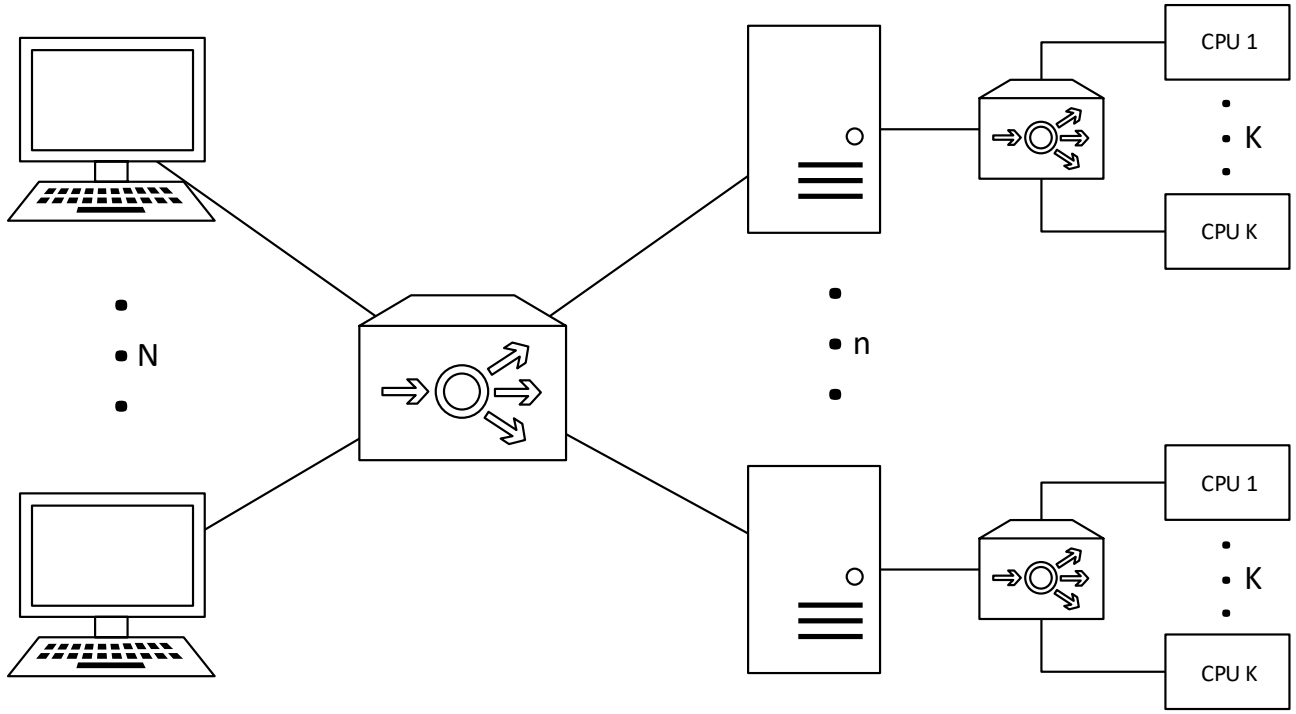


Рис. 4.3. Схема эксперимента

На каждом узле запускается один экземпляр СУБД MySQL и имеется полная локальная копия БД. После запуска СУБД начинает процесс инициализации: выполняются 14 запросов теста, указанные в файле `mysql_init.sql`. Это позволяет загрузить БД в оперативную память и исключить из работы дисковую подсистему. Поскольку в начале выполнения пакета происходит заполнение очередей («разгон»

системы), в конце – опустошение («торможение» системы), а нас интересует работа системы в стационарном режиме, то из полученных результатов будем исключать первые и последние 100 запросов. Число 100 при N=50 использовано потому, что в процессе заполнения очередей некоторые запросы выполняются быстрее, чем успевают заполниться все очереди. Это хорошо видно на рис. 4.4, где при n=4 стационарный режим работы достигается на 61 запросе.

Рис. 4.4. Длина очереди запросов в balancer при n=4

Качество системы оценивается двумя параметрами:

– Среднестатистическое время ожидания ответа на вновь поступивший запрос оценивается величиной математического ожидания:

$$M\left(t_{зг}^{n}\right)=\frac{\sum t_{зг}^{n}}{m}.$$

– Среднеквадратическое отклонение:

$$\sigma=\sqrt{M\left(\left(t_{зг}^{n}-M\left(t_{зг}^{n}\right)\right)^{2}\right)}.$$

Здесь: $t_{зг}^{n}$ – задержка ответа на вновь поступающий r-запрос ($r\in R=\{(n+1),800\}$), вычисляемая как разность времен поступления запроса в очередь и получения результата. m – интервал измерений, $m=800$ из-за исключения этапов установления стационарного режима работы системы и его завершения.

В рамках эксперимента было произведено 7 запусков системы с последовательным увеличением количества узлов *n* с 1 до 7 (соответственно, количества ядер с 8 до 56). По результатам всех запусков получены графики загрузки процессоров, времен ожидания в очереди и ускорения обработки всего ПТ. Графики загрузки

процессоров получены из системы мониторинга Zabbix. Один из графиков (а именно график загрузки процессоров узла *ibhost11*) представлен на рис. 4.5, здесь по оси абсцисс – время, а по оси ординат – процент загрузки узла. Т.к. в узле 8 вычислительных ядер, полная загрузка всех ядер показана на графике как 100%, а полная загрузка одного ядра как 12,5%. Во время инициализации СУБД используется 1 ядро узла, на графике это видно, как небольшой всплеск активности в самом начале временного интервала на уровне 12,5%. Каждый последующий всплеск активности нагрузки соответствует очередному запуску эксперимента.

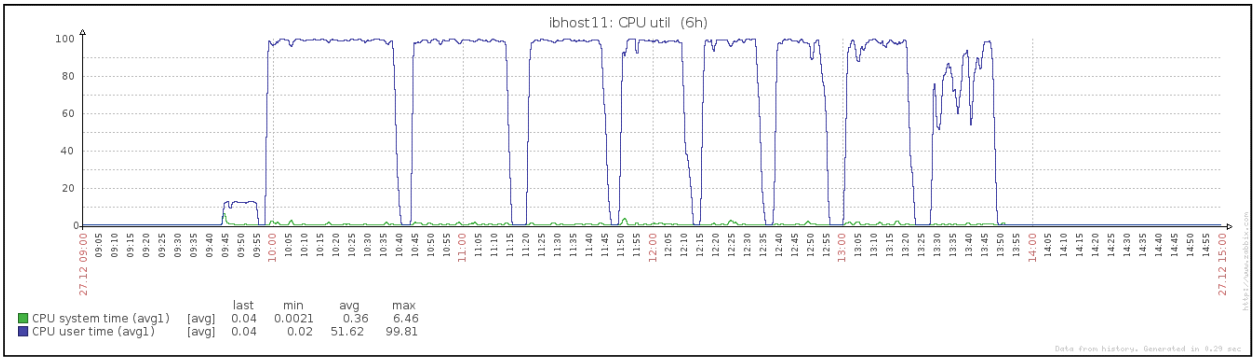


Рис. 4.5. Загрузка процессоров узла ibhost11

На рис. 4.6 представлен один из графиков времени ожидания результата выполнения запроса, полученный при запуске на 6 узлах. Здесь по оси абсцисс – порядковый номер запроса, а по оси ординат – время ожидания выполнения в минутах. Пунктирная линия показывает среднее значение времени ожидания.

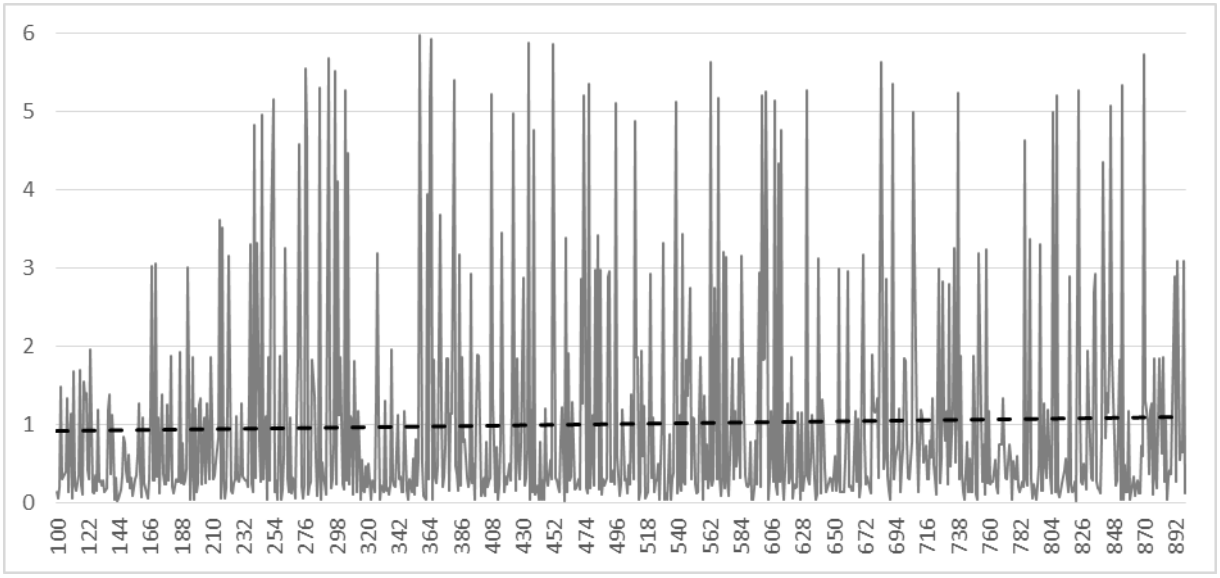


Рис. 4.6. Времени ожидания результата выполнения запроса

В таблице 4.1 приведены полученные значения среднего времени ожидания (M), времени обработки пакета запросов (T) под номерами с 100 по 900 и среднеквадратическое отклонение (σ).

По данным из таблицы 4.1 построены графики: рис. 4.7– график уменьшения времени ожидания, где по оси абсцисс – количество узлов, а по оси ординат время ожидания; рис. 4.8 – график ускорения обработки всего ПТ, где по оси абсцисс – количество узлов, а по оси ординат – коэффициент ускорения.

Таблица 4.1. Сводные результаты

n	M **, минут**	T **, минут**	σ **, минут**
1	6,0400	98,72	0,9151
2	3,0202	48,85	0,4097
3	2,0205	33,63	0,2460
4	1,5206	25,18	0,2220
5	1,1809	19,92	0,1577
6	1,0123	17,15	0,1535
7	0,9506	15,97	0,1337

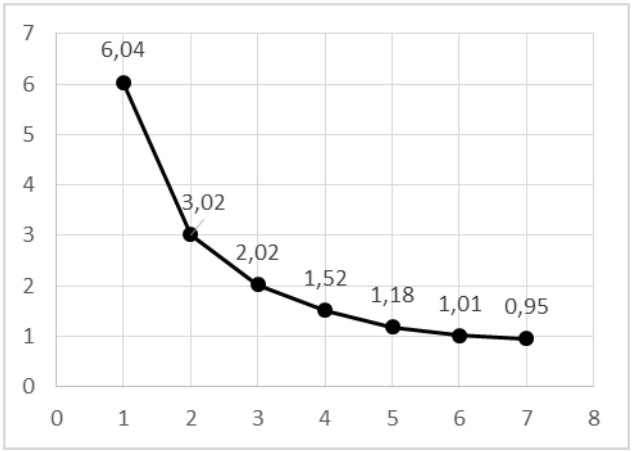


Рис. 4.7. График зависимости среднего времени ожидания в минутах от количества узлов в системе

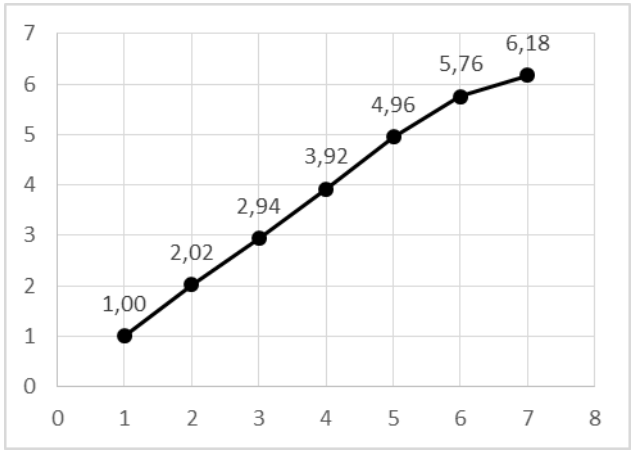


Рис. 4.8. График ускорения обработки ПТ в зависимости от количества узлов

Постановка эксперимента. Случай с router. В [96] было высказано предположение, что дальнейшая мультикластеризация, аналогичная рассмотренной ранее, должна дать дополнительный эффект. Чтобы проверить это предположение был разработан новый программный модуль *router*, который скрывал за собой группу узлов *server* и распределял запросы между ними ранее рассмотренному алгоритму.

Эксперимент проводился аналогично описанному ранее с некоторыми отличиями. Во-первых, изменилась организация узлов кластера (рис. 4.9).

Результаты эксперимента показаны в таблице 4.2, где n – количество задействованных узлов server, Mr – количество *router*, M – среднестатистическое время ожидания ответа на вновь поступивший запрос, T – время выполнения всего ПТ и σ – среднеквадратическое отклонение.

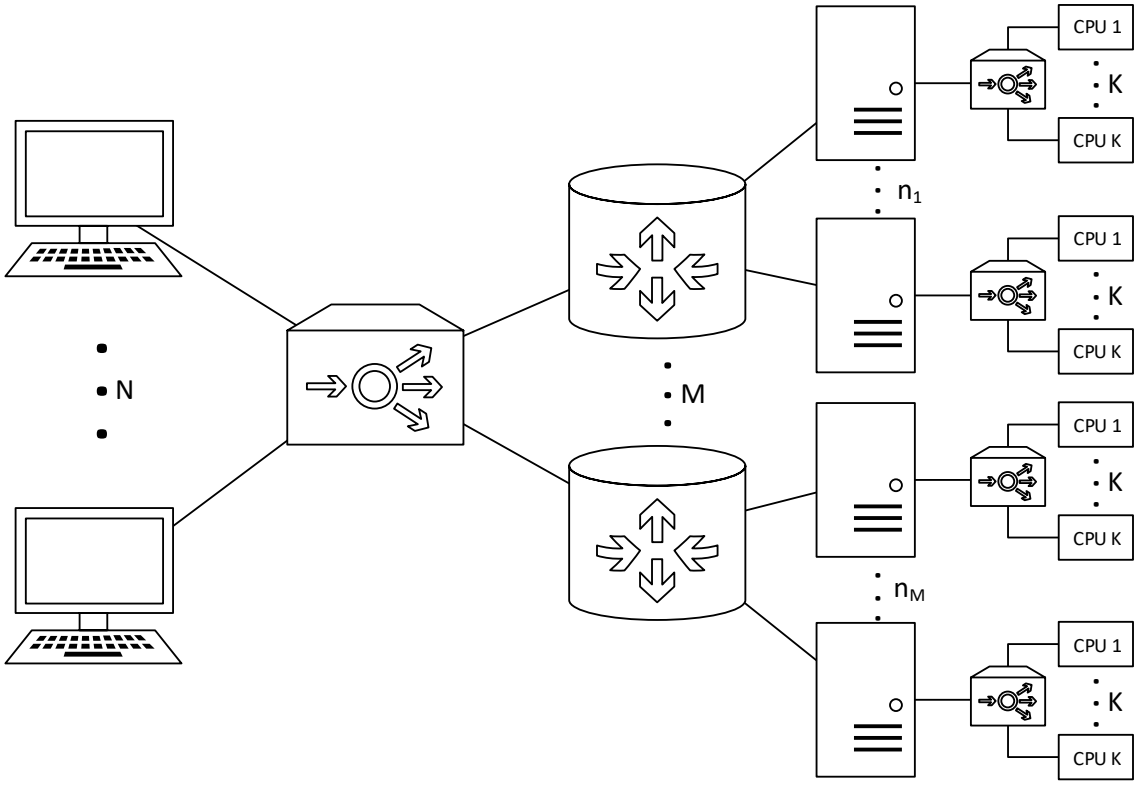


Рис. 4.9. Схема эксперимента с router

По данным из таблицы 4.2 построены графики: рис. 4.10 – график уменьшения времени ожидания, где по оси абсцисс – количество подключенных узлов *router*, а по оси ординат время ожидания; рис. 4.11 – график ускорения обработки всего ПТ, где по оси абсцисс – количество подключенных узлов *router*, а по оси ординат – коэффициент ускорения.

Таблица 4.2. Сводные результаты эксперимента в случае с router

n	**Mr**	M **, минут**	T **, минут**	σ **, минут**
3	1	7,29	42,8	2,48
6	2	3,69	22,68	1,74
9	3	2,49	16,07	1,57
12	4	1,86	12,8	1,46
15	5	1,7	10,3	2,54
18	6	1,53	9,57	3,25

Если сравнить полученные результаты с результатами эксперимента без *router*, то нетрудно видеть, что при N=200 (увеличение в 4 раза) и при увеличении максимального количества задействованных ядер до 144 (увеличение в 2,5 раза) время обработки всего ПТ существенно уменьшилось, среднее время ожидания ответа увеличилось, а ускорение обработки запросов ПТ с увеличением количества узлов хоть и осталось линейным, но коэффициент ускорения оказался ниже.

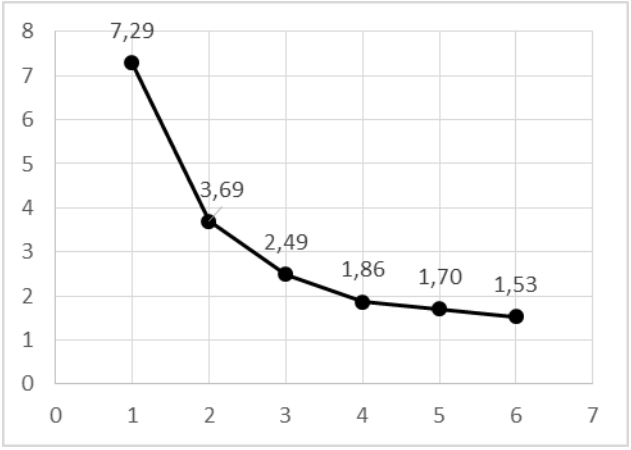


Рис. 4.10. График зависимости среднего времени ожидания в минутах от количества узлов в системе

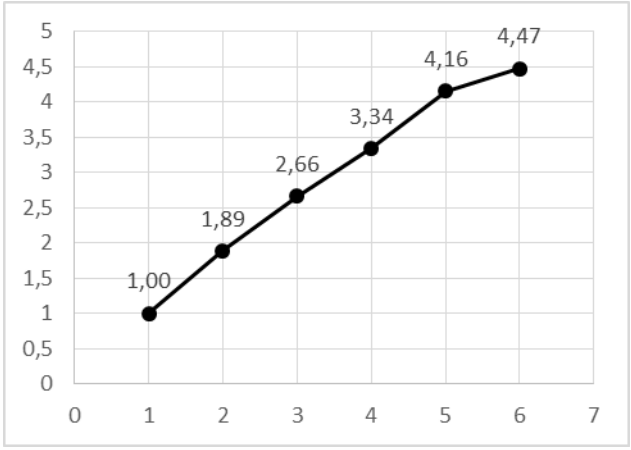


Рис. 4.11. График ускорения обработки ПТ в зависимости от количества узлов

В обоих экспериментах были задействованы все ядра двухпроцессорных узлов HPC SUN-кластера. Это оказалось возможным благодаря соответствующей роуторизации и настройке MySQL, использованию возможности СУБД обрабатывать запросы параллельно в разных сетевых соединениях, наличию достаточного оперативной памяти узла для размещения БД объема.

Результаты экспериментов показали, что стратегия «ядро на запрос +1» работоспособна. Поскольку каждый запрос выполняется на отдельном ядре, то имеет место практически линейное ускорение обработки ПТ с увеличением количества ядер (рис. 4.8 и 4.11). С этим же связно уменьшение среднего времени ожидания ответа (рис. 4.7 и 4.10). Кроме того, показано, что для полного использования вычислительных ресурсов необходимо, чтобы количество клиентов было большим или равным количеству задействованных ядер. Так, на рис. 4.5 можно наблюдать неполную загрузку процессоров во время проведения последнего запуска (7 узлов, 56 ядер, 50 клиентов). В этой итерации эксперимента количество ядер превышает количество клиентов.

Сравнивая результаты эксперимента с *router* и без, можно сделать вывод, что конфигурация без *router* предпочтительна в случае малого числа n узлов или N клиентов. С ростом n и N одного балансировщика может быть недостаточно, в этом случае предпочтительна конфигурация с *router*.

4.3. Платформа GPU-кластера без использования акселераторов. Случаи баз данных в десятки и сотни GB

PerformSys хорошо показал себя в работе с небольшими БД объемом в несколько GB. Но консервативные БД как правило обладают объемом в десятки и сотни GB, а порой в десятки и сотни TB. Поэтому целесообразно экспериментально проверить работоспособность стратегии «запрос на ядро +1» и PerformSys на таких объемах.

Постановка эксперимента. Ранее эксперименты проводились на платформе SUN-кластера, т.к. он удовлетворял всем требованиям для оптимальной производительности PerfromSys. Чтобы провести эксперимент с БД в десятки и сотни GB необходимо: наличие на каждом узле было достаточного объема дисковой памяти для хранения БД и как можно больший объем оперативной памяти. В узлах SUN-кластера установлено дисковой памяти всего на 172 GB. С учетом ее занятости ОС, программами и данными пользователей, свободное место для БД размером в несколько десятков GB отсутствует. Поэтому выполнен переход на платформу GPU-кластера без использования акселераторов.

Представительский тест (ПТ) для новой платформы был организован из 6 перестановок TPC-H Throughput Test без операций записи. На множестве этих перестановок сформирован поток из 84 запросов. Для экспериментов было сгенерировано 2 БД с размером $V_{БД}$ = 60 и $V_{БД}$ = 120 GB. После загрузки БД в MySQL и индексации их размер составил ~100 и ~200 GB соответственно.

Узлы кластера распределены следующим образом (рис. 4.12): 6 узлов исполнительных и 1 узел управляющий. На управляющем узле располагаются программы *client* и *balancer*. Каждый исполнительный узел содержит программу *server* и СУБД MySQL с полной копией БД. В связи со сделанными ранее выводами программа *router* отсутствует, *server* подключается к *balancer* напрямую. По этим же причинам количество симулируемых клиентов N=72 устанавливается в количестве процессорных ядер исполнительных узлов (6 узлов по 12 ядер в узле дают 72 ядра).

Набор запросов в файле `queries.xml` программы *client* такой же как в случае экспериментов на платформе SUN-кластера. Режим выбора очередного запроса установлен в `FromList` (последовательный выбор очередного запроса из заданного списка). Работа идет до исчерпания всего списка запросов.

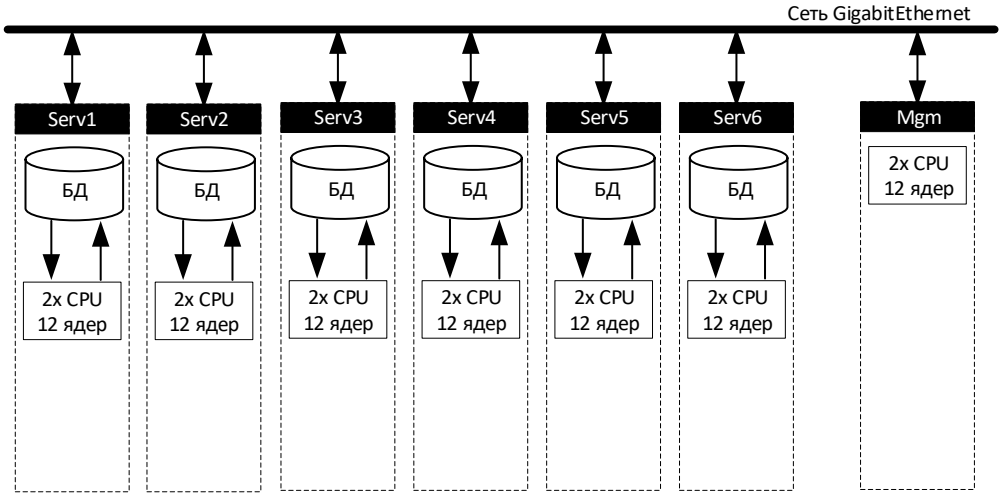


Рис. 4.12. Конфигурация экспериментального полигона для PerformSys

СУБД MySQL сконфигурирована согласно разделу 2.3 со следующими небольшими изменениями.

Результаты выполнения ПТ для двух БД объемом $V_{БД}$ = 60 и $V_{БД}$ = 120 GB представлены в таблице 4.3, где $V_{БД}$ – объем БД в эксперименте, T – время выполнения всего ПТ, M – среднее время ожидания результата клиентом, σ – среднеквадратическое отклонение.

Таблица 4.3. Результаты обработки потока запросов для БД объемом десятки и сотни GB

$V_{БД}$, GB	***T*, часов**	***M*, минут**	**σ, минут**
60 GB	0,93	11,05	14,04
120 GB	50,25	699,36	692,75

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 выполняются) во время 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 RAM-диск). Но вся оперативная память используется для выполнения запросов, поэтому остается один вариант: ввести ожидание готовности.

Ожидание готовности в MySQL реализовано следующим образом. В СУБД передается очередной запрос. Если во время его выполнения возникла ошибка связи с 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. Во время проведения эксперимента с $V_{БД}$ = 120 GB запросов отброшено не было. При этом время до успешного запуска запроса не превысило 5 часов.

4.4. Выводы по главе 4

Технология PerformSys обеспечивает полную загрузку процессорных ядер при обработке аналитических запросов к БД консервативного типа умеренного объема. В этом случае система эффективно обрабатывает поток запросов от множества клиентов благодаря применению стратегии «запрос на ядро +1», специальной настройке инструментальной СУБД и организации роутеризации запросов между узлами и ядрами вычислительного кластера.

Полученные в данной главе результаты говорят о лучшей пропускной способности PerformSys по сравнению с Clusterix-N при умеренных объемах БД: в случае $V_{БД}$ = 60 GB время выполнения ПТ составило 0,93 часа для PerformSys и 2,87 часа для Clusterix-N. При этом. Однако, ситуация меняется коренным образов при работе с $V_{БД}$ = 120 GB. В этом случае время обработки ПТ для Clusterix-N составили 6,73 часа, а для PerformSys – 50 часов. Превосходство PerformSys при $V_{БД}$ = 60 GB объясняется высоким параллелизмом работ: в системе сразу выполняется 72 запроса, в то время как в Clusterix-N всего 2 (один – *select-project-join*, один – *sort*).

Но значения M и σ для Clusterix-N всегда меньше: значение M при $V_{БД}$ = 60 и 120 GB в PerformSys составило 11,05 и 699,36 мин, в Clusterix-N – 3,2 и 6,3 мин соответственно; значения σ – 14,4 и 692,75 мин. против 3 и 5,6 мин. Это немаловажно для пользователя.

Столь низкая производительность PerfromSys в эксперименте с $V_{БД}$ = 120GB является следствием следующих причин:

1. Чтение данных с диска во время обработки запросов. Количество запросов, одновременно исполняемых одним узлом PerformSys, равно числу ядер в узле (12), а каналов доступа к диску – всего 2 (4 диска в RAID 10). Поэтому 10 запросов ожидают, пока 2 предыдущих завершат чтение данных.

2. Объем оперативной памяти оказывается недостаточным, т.к. MySQL хранит в ней результат обработки и все промежуточные/временные отношения по запросу.

3. Ограниченная производительность одного ядра процессора. С ростом объемов БД и RAM время обработки запроса растет линейно и может превысить разумные пределы.

Первая проблема решается использованием быстродействующих флеш-накопителей. Вторая – увеличением объема RAM. Но третья проблема на текущий момент трудноразрешима. Так, например, система Lenovo x3950X6 (8 вычислительных модулей, 144 ядра) с внешней флеш-памятью ~120 TB и объемом оперативной памяти 12 TB [10] оснащена процессорами с частотой работы одного ядра не более 2.8 GHz.

Эффективность Clusterix-N зависит от скорости работы внешнего хранилища только на уровне IO (на узлах MGM достаточно хранилища с быстродействием выше пропускной способности сети, а на узлах JOIN диск не используется). При этом влияние быстродействия внешнего хранилища можно нивелировать, распределив данные на большее количество узлов. Уровень JOIN зависти от объема RAM, но увеличение числа узлов JOIN ведет к снижению требований к нему для каждого узла JOIN. Ограничение производительности одного ядра компенсируется хешированием временных/промежуточных отношений по множеству узлов и параллельной обработкой на всех уровнях. Поэтому, в отличие от PerformSys, Clusterix-N может использовать все вычислительные ресурсы такой системы как Lenovo x3950X6 для ускоренного выполнения одного запроса.

ГЛАВА 5. НЕКОТОРЫЕ ПЕРСПЕКТИВЫ ДЛЯ ДАЛЬНЕЙШЕГО

В данной главе рассматриваются некоторые возможные перспективы для дальнейших исследований с прицелом на работу со сжатыми БД и применение графических ускорителей в блоке IO. Специальное внимание уделено территориально распределенные системам. Развитие таких систем продиктовано повсеместным проникновением сети интернет и глобализацией информационных систем.

В разделе 5.1 рассматриваются способы применения графических ускорителей для работы с БД. В частности, рассматриваются вопросы [114, 115]:

– Применения сжатия данных с помощью GPU при передаче по сети для увеличения быстродействия системы в условиях использования сети GigabitEthernet.
– Увеличения объема хранимых данных в результате хранения в сжатом виде.

Приводятся сравнительные теоретические оценки производительности Clusterix-N и Clusterix-G. При этом для упрощения расчетов рассматривается переход к работе со сжатыми базами данных для начального состояния ***IS***-модели (см. раздел 2.6).

В разделе 5.2 демонстрируется модель территориально распределенной СУБД, представляются методы и алгоритмы балансировки нагрузки в территориально распределенных СУБД [116, 117]. Демонстрация стоится на примере существующей сети RUNNet, которая охватывает всю территорию РФ [95].

Раздел 5.3 настоящей главы посвящен выводам по рассмотренным перспективам для дальнейших исследований.

5.1. Работа со сжатыми базами данных

Из полученных в главе 3 результатов экспериментов видно, что уже при $V_{БД}$ = 120 GB на уровне IO используется практически весь объем RAM. Поправить положение дел возможно с применением сжатия данных. Операция сжатия, как и восстановления, является трудоемкой задачей и может внести дополнительные задержки. Применение графических ускорителей должно их уменьшить. Сжатие данных и обработка запросов *select-project* в каждом узле кластера на GPU преобразует систему Clusterix-N в новую систему Clusterix-G (G от GPU).

Графические ускорители (GPU) применяются для ускорения работы СУБД CoGaDb [118], как расширение PG-Storm для Postgres [119] и др. Использование

GPU позволяет существенно снизить время выполнения отдельных операций. Однако скорость обмена данными с GPU сравнительно невелика. Ограничен и объем глобальной памяти GPU. Именно по этим причинам в [20] удалось достичь роста производительности сервера БД от использования GPU всего на 40%. Как установлено, работа по схеме «один узел кластера на множество запросов», сама по себе, уже обеспечивает достаточно высокую производительность. Но обладает существенным недостатком по сравнению с Clusterix [16]: объем БД ограничен размерами оперативной памяти узла. Поэтому для нее предлагается изменить цель подключения GPU на обеспечение работ с БД значительных объемов при сохранении, как минимум, достигнутого уровня производительности.

Достижение этой цели требует особой организации хранения данных и эффективной работы с графическим ускорителем. Требуется не только перенаправить часть вычислительной нагрузки на GPU, но и увеличить объемы обрабатываемых данных путем хранения данных в сжатом виде. Сжатие данных дополнительно позволяет минимизировать время передачи информации в графический ускоритель. Как и в СУБД Clusterix, обработка запроса происходит по регулярному плану. На графических ускорителях реализуются только операции *select-project*, а операция *join* – на CPU.

Причины выполнения на GPU только операций *select-project* таковы:

1. В связи с ограниченностью памяти графического ускорителя выполнение операции JOIN не всегда возможно. Объем необходимых данных для работы операции JOIN может превышать объем доступной памяти графического ускорителя.

2. Интерфейс подключения графического ускорителя (PCI-e) обладает сравнительно малой пропускной способностью: скорость чтения/записи для трёхканальной оперативной памяти типа DDR3-1600 составляет 38400 MB/c (37,5 GB/c) [120], в то время как для шины PCI-e 2.0 x16 – 6,4 GB/c (64 Gb/c · 0.8, где 64 Gb/c – пропускная способность шины в одну сторону, 0.8 – учёт избыточности 8b/10b для PCI-e 2.0) [121]. Поэтому в случае итерационного выполнения операции JOIN будет вызвано большое количество операций передачи данных между оперативной памятью хоста и графическим ускорителем по шине PCI-e, что не принесет никакого ускорения выполнения этой операции.

Операция *select-project* обеспечивает редуцирование исходных данных примерно в 7 раз [97], что должно положительно сказаться на времени передачи результата из памяти GPU в память узла.

Эффективность сжатия БД. Как уже отмечалось, скорость передачи данных по шине PCI-e значительно ниже скорости обмена с оперативной памятью. Поэтому предварительное сжатие данных должно повысить эффективность передачи данных. Для проверки данной гипотезы был проведен эксперимент с целью сравнения времени, затрачиваемого на простое копирование, с временем необходимым для копирования данных, сжатых по алгоритму RLE (Run Length Encoding) [122, pp. 25-26], и их восстановления.

Суть алгоритма заключается в замене подряд идущих повторяющихся символов/чисел на один символ/число и число его повторов. Например, числовой ряд «1,1,2,2,2,4,4,4,4», состоящий из 9 элементов, будет закодирован в ряд «1,**2**,2,**3**, 4,**4**», состоящий из 6 элементов, где выделенные числа – количество повторений предыдущего числа.

Эффективность передачи сжатых данных оценивалась с помощью эксперимента. Эксперимент проводился на системе следующей конфигурации: CPU – Core i5 4670K (3.8 ГГц), GPU – Nvidia GTX 770 (с объемом памяти 2 Гб GDDR5), RAM – 24 Гб (DDR3-1600 в двухканальном режиме). Данными для проведения эксперимента являются наборы рядов целых чисел, длиной L от 10^5 (0,38 MB) до $168 \bullet 10^5$ (64,09 MB) с шагом 10^5 (0,38 MB) элементов. Каждый набор генерировался таким образом, чтобы достигался заданный коэффициент сжатия по алгоритму RLE. На каждой итерации создавался новый набор, состоящий из 9 рядов сжатых данных с коэффициентами сжатия K от 2 до 10 и один несжатый ряд. Сжатые ряды представляют собой числовые массивы вида «1,**2**,2,**2**,3,**2**,4,**2**», где выделенные числа равны коэффициенту сжатия набора K, а не выделенные - инкрементируются до достижения значения равного L/K. Несжатый ряд A формируется по формуле $A_i = A_{i-1} + 1$, где $i = \overline{2, L}$, а $A_0 = 1$.

Полученные в ходе эксперимента результаты представлены на в таблице 5.1.

Согласно алгоритму сжатия RLE, при сжатии данных в 2 раза, объем передаваемых данных не изменятся, т.к. передается 2 массива половинной длины от ис-

Таблица 5.1. Время копирования в память GPU и восстановления сжатой информации

Размер (Мбайт)	**Копирование, мс**	**Копирование и восстановление сжатой информации, мс**								
		x2	**x3**	**x4**	**x5**	**x6**	**x7**	**x8**	**x9**	**x10**
0,38	1	5	5	4	4	4	4	4	4	4
4,20	3	9	7	6	8	8	7	7	7	7
8,01	5	13	10	9	9	10	10	11	10	10
11,83	7	18	14	12	12	15	15	15	14	15
15,64	21	24	19	15	15	17	19	20	19	19
19,45	26	28	23	20	19	21	24	25	24	23
23,27	31	34	27	23	24	26	27	29	28	26
27,08	35	37	30	26	27	30	31	33	31	31
30,90	41	42	34	30	29	33	37	36	36	34
34,71	45	47	38	35	32	37	39	41	40	39
38,53	50	51	42	35	35	40	44	45	44	43
42,34	55	56	45	38	38	45	47	49	47	47
46,16	59	61	49	41	42	48	51	53	52	52
49,97	64	66	53	44	44	51	55	57	55	55
53,79	68	69	56	48	48	55	59	61	59	59
57,60	75	76	60	50	51	59	63	64	64	63
61,42	80	80	63	53	53	63	67	69	68	66

ходных данных. Это подтверждается данными таблицы 5.1, где времена простого копирования и копирования сжатых в 2 раза данных примерно одинаково. Из таблицы видно, что наибольшая эффективность передачи данных, сжатых алгоритмом RLE, достигается в случае коэффициента сжатия в 4-5 раз. При большем коэффициенте сжатия увеличивается время восстановления данных, поэтому для восстановления чрезвычайно сильно сжатых данных может потребоваться больше времени, чем на обычное копирование.

В таблице 5.2 показан процент увеличения эффективности передачи сжатых данных (коэффициент сжатия – 4 и 5) с последующим восстановлением по сравнению с простым копированием. Как видно из таблицы, при передаче небольшого объема данных простое копирование оказывается быстрее копирования сжатых данных с последующим восстановлением. Но при объемах >12 Мбайт простое копирование занимает бóльшее время.

Несмотря на издержки восстановления информации, сжатой с большим коэффициентом сжатия, большой коэффициент сжатия позволяет хранить больше информации на тех же носителях информации. Коэффициент сжатия данных зависит

Таблица 5.2. Увеличение эффективности передачи данных

Размер (Мбайт)	Копирование, мс	Копирование и восстановление сжатой информации, мс		Увеличение эффективности передачи, %	
		x4	x5	x4	x5
0,38	1	4	4	-300%	-300%
4,20	3	6	8	-167%	-100%
8,01	5	9	9	-80%	-80%
11,83	7	12	12	-71%	-71%
15,64	21	15	15	29%	29%
19,45	26	20	19	27%	23%
23,27	31	23	24	23%	26%
27,08	35	26	27	23%	26%
30,90	41	30	29	29%	27%
34,71	45	35	32	29%	22%
38,53	50	35	35	30%	30%
42,34	55	38	38	31%	31%
46,16	59	41	42	29%	31%
49,97	64	44	44	31%	31%
53,79	68	48	48	29%	29%
57,60	75	50	51	32%	33%
61,42	80	53	53	34%	34%

от самих данных и алгоритма сжатия. Как показала практика, используя цепочку алгоритмов сжатия можно получить коэффициент сжатия вплоть до 1500 раз [123].

Организация хранения сжатых данных. В исходном виде данные могут занимать значительный объем, который необходимо хранить, читать и передавать. Сжатие данных позволяет хранить их в большем объеме и быстрее передавать. Но для проведения над ними каких-либо работ их следует сначала восстановить. Восстановление данных вносит некоторую задержку в обработку данных, которая может нивелировать выигрыш в скорости передачи. Поэтому алгоритмы восстановления должны быть быстрыми и параллельными, пригодными для работы с графическими ускорителями (как, например, алгоритм RLE, приведенный выше).

Различают СУБД, ориентированные на хранение данных по строкам и по столбцам. Лучшие показатели по сжатию имеют СУБД второго типа, т.к. в них сжатие применяется для заранее определенного типа данных и для каждого столбца можно подобрать оптимальный алгоритм сжатия. Данный вопрос уже исследован в [123], где рассмотрено 5 алгоритмов сжатия (NS, NSV, DICT, BITMAP, RLE) и 4 вспомогательные схемы (FOR, DELTA, SEP, SCALE). Вспомогательные схемы

позволяют добиться более высокого коэффициента сжатия при применении их перед применением алгоритма сжатия. Например, пусть дана последовательность 1,2,3,4,5. Тогда при ее сжатии алгоритмом RLE получим удвоение исходного количества данных ([1,1], [2,1], [3,1], [4,1], [5,1]). Однако, если сначала применить вспомогательную схему DELTA (разность текущего и предыдущего элемента последовательности), то получим последовательность 1,1,1,1,1, которая успешно сжимается алгоритмом RLE: [1,5]. Поэтому для последовательности 1,2,3,4,5 выгодно применять цепочку алгоритмов: DELTA, RLE.

Выбор алгоритма сжатия в [123] для той или иной колонки осуществляется с помощью планировщика по ее параметрам: наличие сортировки, средняя длина повторяющихся подряд значений, количество уникальных значений, максимальная длина значения в байтах, возможность дробления значений столбца на более простые значения (например, для поля даты хранить день, месяц, год отдельно). При этом осуществляется выбор сразу цепочки алгоритмов сжатия, применение которой дает достижение максимального коэффициента сжатия.

Помимо ограничений, связанных с относительно медленным интерфейсом передачи данных (PCI-e), современные графические ускорители имеют еще один недостаток: они снабжаются сравнительно небольшим объемом памяти. Поэтому не всегда возможно загрузить в графический ускоритель сразу все данные. Это ограничение возможно обойти с помощью разбиения данных на блоки, что позволяет итерационно (блок за блоком) обрабатывать объем информации, превышающий объем памяти графического ускорителя согласно следующему алгоритму.

АЛГОРИТМ БЛОЧНОЙ ОБРАБОТКИ СЖАТЫХ ДАННЫХ НА GPU. Обозначим: $B = \{B_i\}$, $i = \{\overline{0, n-1}\}$ – множество i-блоков данных, где n – общее количество блоков данных; B^D – восстановленный блок данных из сжатого.

1. $i := 0$
2. Передать блок B_i в GPU
3. Восстановить B_i в B^D
4. Удалить блок B_i из памяти GPU
5. Выполнить обработку B^D

6. Передать результаты в память хоста
7. Удалить B^D из памяти GPU
8. $i := 0$
9. Если $i = n$, то завершить обработку, иначе перейти к шагу 2.

При этом необходимо учитывать объем востановлленых данных и объем сжатых данных, т.к. они будут находится в памяти графического ускорителя одновременно. Поскольку в худшем случае данные не будут сжаты (т.е. их объем относительно исходного не изменится), то верным предполагается разделение памяти графического ускорителя на 3 равные части: часть сжатых данных, часть распакованных данных и часть результата. В случае такого разделения данные загружаются в первую область, распаковываются во вторую. После распаковки данные в первой области уничтожаются и область готова к приему нового блока данных. Такое разделение памяти позволяет загружать порцию сжатых данных во время обработки распакованных данных и организовать конвейерную обработку.

Организация работы со сжатыми данными на GPU. Графические ускорители обладают большим количеством вычислительных ядер и быстрой памятью. Это позволяет быстро выполнять параллельные операции. В работе с БД представляется возможным параллельное выполнение всех операций *select-project-join*. Но выполнение операции *join* не всегда возможно. Сжатие данных позволяет минимизировать время передачи информации в графический ускоритель и увеличить объем, обрабатываемых данных. Выполнение запроса происходит по регулярному плану. Операции *select-project* выполняются на графических ускорителях. Операция *join* выполняется на CPU.

Запрос поступает в систему, где транслируется и разбивается на подзапросы согласно регулярному плану обработки. Для каждой таблицы, участвующей в обработке, определяется набор колонок, необходимых для выполнения операций *select-project*. В графический укоритель поблочно передаются колонки одной таблицы, над ними производятся необходимые операции. Результат возвращается с графического ускорителя и размещается в памяти для последующего выполнения операции *join*. Процесс передачи данных и выполнения операций *select-project* выполняются для всех таблиц согласно плану обработки. Результат работы сохраняется в памяти для последующего выполнения операции *join*.

Конвейер выполнения операции *select-project* представлен на рис. 5.1.

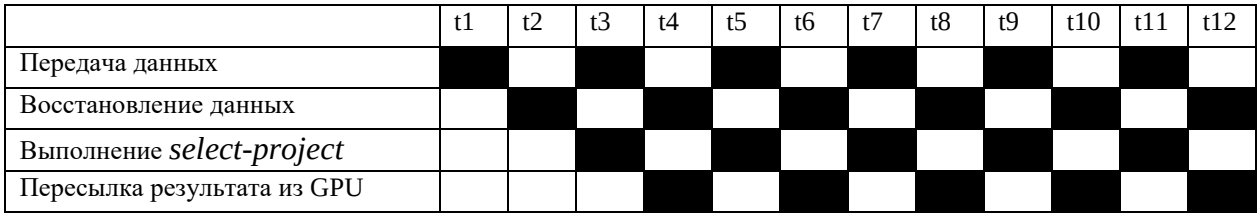


Рис. 5.1. Конвейер выполнения операции *select-project* на GPU

Здесь:

- t1 – поступление блока в GPU,
- t2 – восстановление данных блока в GPU,
- t3 – поступление следующего блока в GPU и выполнение операции *select-project*,
- t4 – восстановление данных нового блока в GPU и передача результата операции *select-project* в хост,
- t5 – поступление следующего блока в GPU и выполнение операции *select-project*,
- t6 – восстановление данных нового блока в GPU и передача результата операции *select-project* в хост,
- t7 – поступление следующего блока в GPU и выполнение операции *select-project*.

И т.д.

В идеальном случае графический ускоритель должен всегда выполнять параллельно 2 операции: операцию передачи данных в/из хоста и выполнять вычисления (операции восстановления или *select-project*).

Согласно рис. 5.1 на графическом ускорителе одновременного происходит обработка 2-х блоков. Поскольку данные блоков не должны пересекаться в памяти графического ускорителя, то память должна разделяться на 3 равные секции:

1. Буфер сжатых данных – используется для пересылки сжатых данных с хоста.

2. Буфер восстановленных данных – используется для выполнения операций *select-project*.

3. Буфер результата – используется для хранения результата *select-project* и передачи результата в хост.

Тогда получим следующую картину. Когда первый блок проходит через выполнение операций *select-project* (t3 на рис. 5.1), им заняты области памяти 2 и 3, а область 1 свободна. Туда начинается загрузка сжатых данных из блока 2. На следующем этапе (этап t4 на рис. 5.1) освобождается область 2, и она начинает запол-

няться восстановленными данными блока, в то время как область 3 занята данными результата, а область 1 занята сжатыми данными блока. На этапе t5 рис. 5.1 освобождается 1 область памяти и процесс повторяется. В случае использования нескольких графических ускорителей последовательность блоков обрабатывается параллельно. Распределение блоков происходит по круговому алгоритму. Так при обработке 5 блоков на 2-х видеокартах получим следующее распределение: блоки под номерами 1,3,5 обрабатываются на первом графическом ускорителе, а блоки под номерами 2,4 – на втором.

Под блоком данных здесь понимается набор сжатых столбцов с общим объемом восстановленных данных равным буферу восстановленных данных.

АЛГОРИТМ ПОДГОТОВКИ ДАННЫХ ДЛЯ СЖАТИЯ. Обозначим: BS – объем памяти, отведенной для восстановленных данных в GPU; RS – максимальная длина записи в отношении; RC – количество записей для в одном блоке.

1. Найти самую длинную запись в обрабатываемом отношении и записать ее длину в RS.
2. Найти количество записей в отношении, которое гарантированно умещается в отведенной памяти, по формуле: RC=BS/RS с округлением вниз.
3. Выдавать данные для сжатия из отношения с шагом RC.

Например, требуется сжать отношение длинной 5 млн. строк с 4-я колонками (1 – `int`, 2 – `char`, 3 – `string(50)`, 4 – `text`) для буфера размером 100 MB. Согласно алгоритму, в начале будет выполнен поиск по отношению с целью выявления строки максимальной длины. Этот шаг необходим при наличии поля с типом `text` (т.к. оно хранит данные произвольной дины), во всех остальных случаях длина строки заранее известна. Пусть максимальная длина оказалась равной 100 байт (4 байта – `int`, 1 байт – `char`, 50 байт – `string`, 55 байт – `text`), тогда, согласно формуле второго пункта алгоритма, количество строк в блоке будет равно: (100·1024·1024) / 100 ≈ 1 млн. Количество сжатых блоков 5 млн. / 1 млн. = 5. Объемы данных в одном блоке для каждого столбца равны: 1 – 4 MB, 2 – 1 MB, 3 – 50 MB, 4 – до 55 MB. Каждый столбец в таком блоке сжимается отдельно. Размер блока варьируется в зависимости от необходимого количества столбцов для обработки, но он никогда не превысит объем буфера после восстановления.

После операции *select-project* сжатие данных для передачи не происходит. Это связано с тем, что в результате выполнения операции *select-project* объем данных существенно уменьшается [97].

Предложенная организация работы с сжатыми данными не только позволяет хранить больше данных. Она может позволить ускорить передачу данных между узлами путем передачи их в сжатом виде, организуя аналогичный рис. 5.1 конвейер с дополнительным шагом – сжатие результата.

Сравнительные теоретической оценки эффективности. Прежде чем выполнить теоретическую оценку производительности Clusterix-N, необходимо собрать исходные данные по обработке запросов без операции записи теста TPC-H [11]:

1. Время выполнения каждого запроса в целом.
2. Время выполнения всех *select-project* для получения отношений R_i.
3. Объемы полученных отношений R_i.
4. Время индексации отношений R_i.
5. Время выполнения операций *join*.

Сбор исходных данных предполагает ряд экспериментов над тестовой БД из состава теста TPC-H. В качестве инструментальной СУБД была выбрана MySQL 5.7. Для запросов теста были сформированы загрузочные модули *select-project* (по каждому R_i) и *join* (по каждому запросу в целом). При этом все операции *project* были «опущены вниз». Объем тестовой БД составил 1GB. БД предварительно загружалась в оперативную память. Сравнивались времена, затрачиваемые на реализацию всех операций *select - project* и единой процедуры *join* по каждому запросу ПТ, с временами выполнения этих запросов в целом.

Эта процедура выполнялась по следующему алгоритму:

1. Создание промежуточных отношений в памяти MySQL для хранения результата *select - project.*
2. Индексация промежуточных отношений.
3. Выполнение единой процедуры *join:* $R_1' join (join\ R_2'(join\ R_3'(\dots))) \dots)$.

Полученные результаты представлены в таблице 5.3.

Как следует из таблицы 5.3, значения $t_\Sigma^{\sigma,\pi}$(кроме запросов № 1, 13) и t_Σ^{join} (кроме запросов № 12, 14) превышают $t_{общ}$. Это объясняется тем, что MySQL 5.7

Таблица 5.3. Сравнение времен на реализацию по-отдельности операций *select – project, join* и запроса в целом

№ запроса	**Время обработки запроса, сек**					**Отношение времен обработки**	
	Оригинального запроса	**Выполнение *«select-project»***	**Индексация рез-та *«select-project»***	**Выполнение *«join»***	**t_{Σ}^{join}**		
	$t_{общ}$	$t_{\Sigma}^{\sigma,\pi}$	$t_{инд}$	t^{join}	$t_{инд}+ t^{join}$	$t_{общ} / t_{\Sigma}^{\sigma,\pi}$	$t_{общ} /t_{\Sigma}^{join}$
1	21,579	9,808	0,000	0,000	0,000	2,20	нет *join*
2	0,145	2,050	3,633	0,011	3,644	0,07	0,040
3	2,142	7,939	3,270	0,163	3,433	0,27	0,624
4	1,035	6,668	1,623	0,470	2,093	0,16	0,495
5	1,123	8,721	11,892	0,154	12,046	0,13	0,093
6	2,846	5,054	0,000	0,000	0,000	0,56	нет *join*
7	0,901	7,624	5,506	0,261	5,767	0,12	0,156
8	3,335	9,837	22,394	0,081	22,475	0,34	0,148
9	5,315	11,355	26,805	2,234	29,039	0,47	0,183
10	1,850	6,707	1,614	0,944	2,558	0,28	0,723
11	0,229	1,896	1,209	0,069	1,278	0,12	0,179
12	3,760	6,561	0,809	0,075	0,884	0,57	4,253
13	3,302	1,833	0,714	4,703	5,417	1,80	0,610
14	3,270	5,339	0,198	0,264	0,462	0,61	7,078
Итого	50,830	91,392	79,667	9,428	89,096	0,556	0,571

оптимизирует исполнение запросов, объединяя выполнение некоторых операций. Усредненные значения $t_{общ}/t_{\Sigma}^{\sigma,\pi}$= 0,556 и $t_{общ}/t_{\Sigma}^{join}$ = 0,571 примерно одинаковы: $t_{\Sigma}^{join}/t_{\Sigma}^{\sigma,\pi}$ =0,975.

В результате *«select-project»*-обработки формируются отношения R_i, объем которых существенно меньше объема исходных отношений. Чтобы определить насколько уменьшается объем данных необходимо найти сумму объемов всех исходных отношений каждого запроса и сумма объемов отношений R_i. Подсчет объемов выполнялся путем получения суммы длин всех кортежей. Длина кортежа вычислялась как сумма длин его полей. Подсчет производился средствами MySQL с помощью модифицированных запросов.

Пример модификации для запроса №3:

```
1.   -- O'
2.   SELECT O_ORDERDATE, O_SHIPPRIORITY, O_ORDERKEY, O_CUSTKEY
3.   FROM ORDERS
4.   WHERE O_ORDERDATE < DATE '1995-03-31';
5.    -- O' LENGTH
6.   SELECT SUM(LENGTH(O_ORDERDATE), LENGTH(O_SHIPPRIORITY), LENGTH(O_ORDERKEY),
     LENGTH(O_CUSTKEY))
7.   FROM ORDERS
8.   WHERE O_ORDERDATE < DATE '1995-03-31';
```

Здесь `O'` – один из подзапросов *select-project*, полученных после претрансляции этого запроса; `O' LENGTH` – запрос для подсчета объема возвращаемых данных. Результаты измерений объемов информации необходимой для обработки запросов приведены в таблице 5.4, согласно которой имеем, в среднем, примерно 7-кратное уменьшение объема данных.

Таблица 5.4. Объемы данных для обработки запросов TPC-H

№ запроса	**Объем данных для обработки, байт**		**Коэффициент уменьшения объема данных**
	До выборки по условиям	**После выборки по условиям**	
	$V_{общ}$	$(\sigma, \pi)_{\Sigma}$	$V_{общ}/(\sigma, \pi)_{\Sigma}$
1	675 846 277	134 255 929	5,03
2	137 651 393	13 526 703	10,18
3	855 794 582	76 991 966	11,12
4	832 798 438	428 049 230	1,95
5	857 126 234	139 324 211	6,15
6	675 846 277	1 339 256	504,64
7	857 125 865	78 759 670	10,88
8	879 261 359	179 338 247	4,90
9	970 449 462	234 598 226	4,14
10	855 796 681	49 964 804	17,13
11	342 555 947	27 528 586	12,44
12	832 798 438	23 140 549	35,99
13	179 948 305	18 706 950	9,62
14	697 981 402	6 548 108	106,59
Итого	9 695 098 066	1 412 072 435	6,87

В таблице 5.4: $V_{общ}/(\sigma, \pi)_{\Sigma}$ – полный объем отношений, необходимый для выполнения запроса; $(\sigma, \pi)_{\Sigma}$ – сумма объемов тех же отношений после уменьшения объема данных с использованием условий выборки из запроса (секции `where`).

Теоретические оценки для простоты будем строить для начального состояния ***IS***-модели на основе гипотетического кластера из 7 узлов: 3 узла – IO, 3 узла JOIN и 1 узел управляющий (MGM). Модуль динамической сегментации отношений отсутствует. Равенство количества узлов IO и JOIN вытекает из таблицы 5.3, где отношение времен обработки подзапросов *select-project* и *join* к общему времени обработки запроса примерно одинаково. Рассматривать будем случай $V_{БД}$ = 5GB с конкатенацией 3 ПТ; БД распределена (хеширована) по узлам IO, 3 ПТ обрабатываются последовательно, один за другим; на узлах JOIN по 1 ПТ обрабатывается на каждом узле.

Несмотря на то, что экспериментальные данные получены для $V_{БД}$ = 1GB, согласно исследованиям, проведенным в [95], имеем линейную зависимость времени выполнения запросов от объема БД. Поэтому в расчетах будем просто умножать время для $V_{БД}$ = 1GB на объем БД из рассматриваемого случая.

Времена параллельного функционирования блоков IO и JOIN определятся соответственно величинами $t_{\Sigma}^{\sigma,\pi}$ и $(t_{\Sigma}^{join})^{12}$ для самых «тяжелых» на ПТ запросов. В обоих случаях таковым является запрос №9 (таблица 5.3). Для него

$$t_{\Sigma}^{\sigma,\pi} \approx 11{,}4 \cdot 5 \cdot 3/3 = 57 \text{ сек},$$

где 11,4 – $t_{\Sigma}^{\sigma,\pi}$ при $V_{БД}$ = 1GB, 5 – $V_{БД}$, 3 – количество перестановок ПТ, 3 – количество узлов IO.

Время передачи по сети GigabitEthernet пакета объемом $V_{\Sigma}^{Ri'}$ (сумма объемов всех отношений R_i для $V_{БД}$ = 1GB) составит

$$t_n'' \approx 1{,}4 \cdot 5 \cdot 3/0{,}1 = 210 \text{ сек},$$

где 1,4 – $V_{\Sigma}^{Ri'}$, 5 – $V_{БД}$, 3 – количество ПТ, 0,1 – скорость передачи данных по сети GigabitEthernet в GB/сек.

Так что суммарное время *«select–project»*-обработки

$$(t_{\Sigma}^{\sigma,\pi})' = t_{\Sigma}^{\sigma,\pi} + t_n'' \approx 267 \text{ сек}.$$

Значение

$$(t_{\Sigma}^{join})^{12} \approx 29{,}04 \cdot 5 = 145{,}2 \text{ сек},$$

где 29,04 – время t_{Σ}^{join} для 9 запроса.

К этому следует плюсовать время $t_{уд}$ на удаление средствами MySQL поступивших в узел JOIN промежуточных отношений. Как показал предварительный эксперимент, это время не превышает 20% времени выполнения единых операций *join* для всех запросов, если удаление проводить по окончании обработки каждого запроса.

$$t_{уд} \approx 0{,}2(t_{\Sigma}^{join})^{12} = 29{,}04 \text{ сек}.$$

Время передачи по сети GigabitEthernet от MGM к JOIN пакета объемом $(V_{\Sigma}^{Ri'})_{Join}$

$$t_n''' \approx 1{,}4 \cdot 5 \cdot 3/(0{,}1 \text{ GB/s}) = 210 \text{ сек},$$

где 1,37 – $V_{\Sigma}^{Ri'}$, 5 – VБД, 3 – количество ПТ, 0.1 – скорость передачи данных.

Общее время *«join»*-обработки

$$(t_{\Sigma}^{join})' = (t_{\Sigma}^{join})^{12} + t_{уд} + t_n''' \approx 145{,}2 + 29{,}04 + 210 = 384 \text{ сек}.$$

Суммарное время обработки теста в целом

$$T = (t_{\Sigma}^{\sigma,\pi})' + (t_{\Sigma}^{join})' \approx 267 + 384 = 651 \text{ сек}.$$

Эксперимент в разделе 2.6 показал 506 с. в среднем. Это говорит о том, что используемая методика получения теоретической оценки производительности не завышает ее реальную оценку. Сравнение с данными, полученными для Clusterix-M, показывает, что применение рассмотренного подхода на 7 узлах должно позволить в данном случае получить то же быстродействие, что и на 12 узлах мультикластера. Это – существенное улучшение.

Расчетная эффективность Clusterix-G. Применив сжатие данных и обработку запросов *select-project* на GPU к Clusterix-N получим новую систему Clusterix-G (G от GPU). Расчет эффективности Clusterix-G произведем в тех же условиях, что и Clusterix-N: 7 узлов (3 – IO, 3 – JOIN и 1 – управляющий); $V_{БД}$ = 5GB; конкатенация 3 ПТ; на узлах IO БД хеширована, 3 ПТ обрабатываются последовательно, один за другим; на узлах JOIN 1 ПТ обрабатывается на каждом узле.

Теоретическая скорость передачи по шине PCI-e равна 6,4GB/s [121]. Получение на практике 3,2GB/s можно гарантировать. Поэтому для несжатой БД суммарное время передач информации объемом $V_{\sum}^{Ri}$ в каждом IO-узле от CPU параллельно к двум GPU составит

$$t_{n1} = 9{,}7 \cdot 5 \cdot 3/(3 \cdot 2 \cdot 3{,}2) \approx 7{,}58 \text{ сек},$$

а от GPU к CPU –

$$t_{n2} = 1{,}4 \cdot 5 \cdot 3/(3 \cdot 2 \cdot 3{,}2) \approx 1{,}1 \text{ сек}.$$

Хранение исходной БД в сжатом виде с коэффициентом сжатия K=5 уменьшает t_{n1} примерно на 34% (таблица 5.2). Соответственно для 14 запросов ПТ получаем

$$t_n' = t_{n1}' + t_{n2}' \approx 7{,}58 \cdot 0{,}76 + 1{,}1 \cdot 0{,}76 \approx 6{,}6 \text{ сек}.$$

Дополнительно необходимо учесть время на передачу сжатых данных по сети GigabitEthernet от IO к Mgm

$$t_n'' = 1{,}4 \cdot 5 \cdot 3/(5 \cdot 0{,}1) = 42 \text{ сек}.$$

GPU селектирует ПТ последовательно запрос за запросом. Суммарное время такого селектирования одним ядром CPU примерно равно 91,4·5 = 457 сек. Полагая (как это принято считать) десятикратным ускорение от использования одного GPU, для двух GPU получаем

$$t_{\Sigma}^{\sigma,\pi} \approx 457/20 = 22{,}85 \text{ сек}.$$

Итоговое время *«select–project»*-обработки

$$(t_{\Sigma}^{\sigma,\pi})' = t_n' + t_{\Sigma}^{\sigma,\pi} + t_n'' \approx 71{,}5 \text{ сек.}$$

Время передачи сжатых данных по сети GigabitEthernet от Mgm к узлам JOIN

$$t_n''' \approx 1{,}4 \cdot 5 \cdot 3/(5 \cdot 0{,}1) = 42 \text{ сек.}$$

Времена собственно *«join»*-обработки $(t_{\Sigma}^{join})^{12}$ и удаления $t_{уд}$ остаются прежними (как и в *Clusterix-N*). По условию в JOIN-узлах передача принятых сжатых данных в GPU и их восстановление в ускорителях выполняются «на проходе». Тогда параллельная передача восстановленных данных из двух GPU в CPU вносит дополнительную несущественную задержку, равную

$$t_n'''' = 1{,}37 \cdot 5 /(2 \cdot 3{,}2\text{GB/s}) \approx 1{,}1 \text{ сек.}$$

Общее время *«join»*-обработки

$$(t_{\Sigma}^{join})' = (t_{\Sigma}^{join})^{12} + t_{уд} + t_n''' + t_n'''' \approx 145{,}2 + 29{,}04 + 42 + 1{,}1 = 217{,}3 \text{ сек.}$$

Суммарное время обработки теста в целом

$$T = (t_{\Sigma}^{\sigma,\pi})' + (t_{\Sigma}^{join})' \approx 71{,}5 + 217{,}3 = 288{,}8 \text{ сек.}$$

Таким образом, выигрыш в производительности по сравнению с СУБД *Clusterix-N* – примерно в 2,3 раз, т.е. на 57%. Это более чем в 1,4 раза превышает достигнутый в [20] эффект от использования графических акселераторов. Понятие производительности ассоциируется со временем обработки. Выигрыш в производительности в 2,7 раз трактуется как величина отношения $T_{обр1}/T_{обр2}=2{,}7$. Поэтому $T_{обр2} = T_{обр1}/2{,}3 = 0{,}43\, T_{обр1}$, т.е. на 57% меньше (рост производительности – на 57%). Именно такова трактовка производительности в работе [20], в которой достигнутый рост производительности GPU-сервера БД составил 40%.

5.2. Кросс-моделирование территориально распределенных СУБД

Вычислительные сети очень широко распространены в мире. Поэтому очень остро встает вопрос о предоставлении доступа к ресурсам сети огромной массы пользователей. Данный вопрос осложняется географической распределенностью пользователей и неоднородностью каналов связи. Географическая распределенность подразумевает не только удаленность пользователей от сетевых центров, но и указывает на их различные времена активности. Например, в Москве вечер и наблюдается пик активности пользователей, а в Хабаровске ночь и активности

пользователей сети почти нет. Это означает, что вычислительные мощности в Хабаровске простаивают. Нагрузить их можно передав часть запросов из другого региона (например, из Москвы).

Распределением нагрузки занимаются специально разработанные балансировщики со своими алгоритмами [124]. Существуют множество алгоритмов балансировки. У каждого алгоритма есть свои плюсы и минусы, своя сложность реализации. Поэтому балансировщики нагрузки могут оказаться как самыми простыми, так и самыми сложными программными компонентами глобальной сети. Их главной задачей является распределение нагрузки между географически удаленными кластерами/фермами серверов.

Вне зависимости от того, к какой категории относится та или иная система балансировки нагрузки, она выполняет следующие задачи: контроль за нагрузкой и состоянием ферм/кластеров, правильный выбор фермы/кластера, который будет отвечать на запрос клиента, и управление трафиком между клиентом и фермой/кластером.

Серверная ферма (кластер) – это ассоциация серверов, соединенных сетью передачи данных и работающих как единое целое. Система балансировки нагрузки рассматривается как инструментальное средство, предназначенное для переадресации клиентских запросов на наименее загруженную или наиболее подходящую ферму группы кластеров, на которых хранятся зеркальные копии информационного ресурса. Клиент не подозревает о том, что обращается к целой группе ферм: все они представляются ему в виде некоего единого виртуального сервера.

Вопросы балансировки нагрузки между Web-серверами хорошо изучены и рассмотрены в печатных изданиях [125, 126] и в сети интернет [124, 127] и др. Но до сих пор специалисты уделяли мало внимания вопросам распределения запросов пользователей между фермами серверов баз данных, установленными в регионах, относящихся к разным часовым поясам, с использованием ресурсов существующих глобальных сетей. Разработка соответствующих методов может существенно повысить занятость названных сетей и пропускную способность распределенной системы баз данных [116].

Кросс-моделирование территориально распределенных СУБД и демонстрацию методов балансировки нагрузки к ним покажем на примере фрагмента Российской федеральной сети RUNNet [128].

RUNNet (Russian UNiversity Network) – федеральная университетская компьютерная сеть России является основой телекоммуникационной инфраструктуры единой образовательной информационной среды и обеспечивает образовательным учреждениям доступ к российским и мировым научно-образовательным ресурсам Интернета. Сеть RUNNet была создана в 1994 году в рамках государственной программы «Университеты России» как IP-сеть, объединяющая региональные сети, а также сети крупных научно-образовательных учреждений. Основная задача RUNNet – формирование единого информационного пространства сферы образования России и его интеграция в мировое информационное сообщество, реализация международной кооперации в области науки и образования.

В настоящее время сеть RUNNet предоставляет услуги более чем 400 университетам и другим крупным образовательным и научно-исследовательским учреждениям. Инфраструктура сети RUNNet (рис. 5.2) состоит из опорной инфраструктуры (отмечено красным), используемой всеми клиентами, и инфраструктуры доступа к опорной сети, используемой отдельными университетами.

Можно выделить 3 региональных центра, которые соединены наземными коммуникациями и через которые проходит наиболее интенсивный трафик: Москва; Новосибирск; Хабаровск. Корректность выбора в качестве центров именно этих городов подтверждает рис. 5.2.

Рис. 5.2. Инфраструктура сети RUNNet

Выбранные центры подключены к магистральному каналу передачи данных, а ближайшие города получают доступ в сеть через них. Помимо своих каналов связи, сеть RUNNet дополнительно использует сети обмена трафиком (Internet Exchange) [129]. Выделенным региональным центрам соответствуют 2 точки обмена: Москва – Московский Internet Exchange (MSK-IX) (рис. 5.3а), Новосибирск – Новосибирский Internet Exchange (NSK-IX) (рис. 5.3б). Наиболее близкая точка обмена трафиком для Хабаровска – Владивостокский Internet Exchange (VLV-IX) (рис. 5.3в).

При проведении модельного кросс-эксперимента считается, что уровень трафика меньше 1/3 максимального, означает что регион в данный момент пассивен. Соответственно средний период «сна» по всем регионам ~ 8 час. Степень загрузки точек обмена трафиком между соседними регионами сдвигается ~ на 4 час. по часовым поясам.

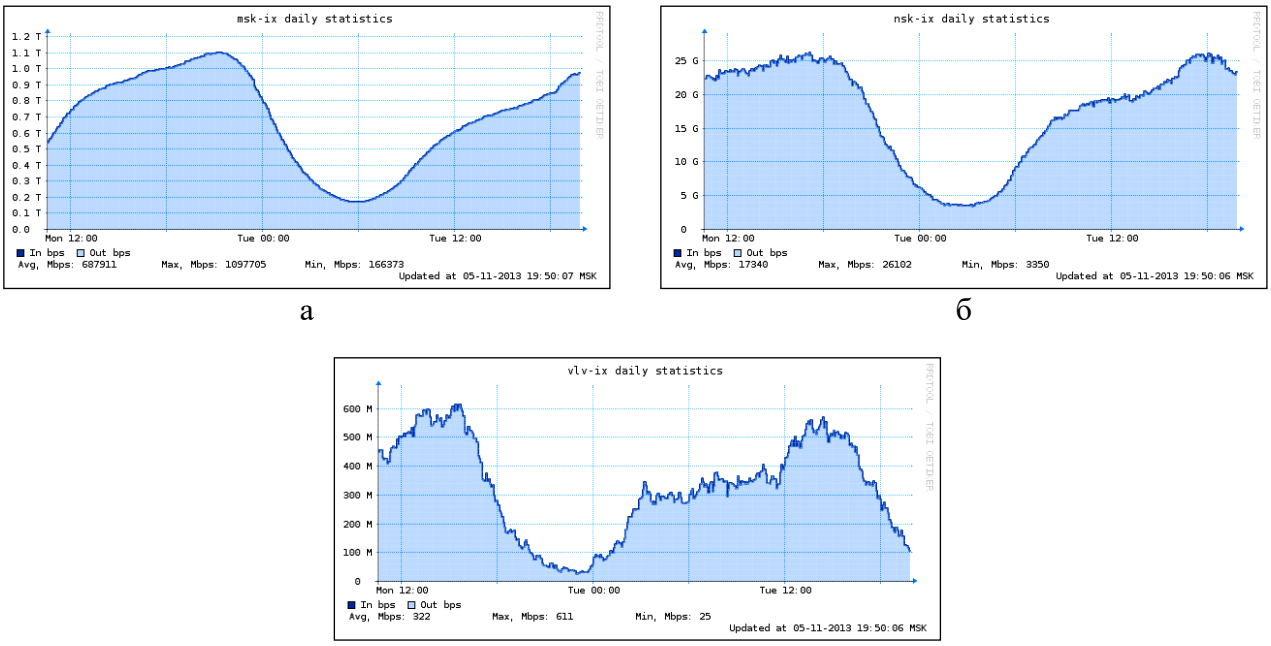


Рис. 5.3. Нагрузка на Московский (а), Новосибирский (б) и Владивостокский (в) Internet Exchange за день

Постановка задачи. Задержки передачи данных в сети RUNNet не превышают 200 миллисекунд, что много меньше времен выполнения запросов, представленных в таблице 5.5. Поэтому в процессе моделирования сетевые задержки не учитываются.

Таблица 5.5 содержит средние времена исполнения запросов теста TPC-H [11] без операций записи для БД объемов $V_{БД}$ = 5, 10, 15 GB. Времена получены пятикратным исполнением каждого запроса в СУБД MySQL на одном узле GPU-кластера и вычислением среднего значения по всем запускам.

Система в целом однородна (не гетерогенна): во всех r-регионах, $r \in \{\overline{1,q}\}$, где q – количество регионов, используются одинаковые $СУБД_r$ и серверы баз данных IO_{rl}, $l \in \{\overline{1,n_r}\}$. Число серверов n_r и объем баз данных $V_{БД_r}$ зависит от r. Любой IO_{rl} хранит копии всех $БД_r$.

Таблица 5.5. Времена выполнения запросов, сек

№ запроса / $V_{БД}$, GB	1	2	3	4	5	6	7	8	9	10	11	12	13	14
5	132,4	1,2	13,2	7,2	9,3	16,1	8,7	23,6	47,4	16,2	1,6	22,7	28,3	18,3
10	256,8	2,1	25	14,3	17,3	31,8	16,8	47,8	89,5	32,6	3,3	44,3	55	36,3
15	385,6	3,1	38,3	21,7	27,5	47,7	24,4	72,7	136	64,6	5	66,5	85,7	54,8

В течение астрономических суток серверы всех регионов работают непрерывно. Но степень временной активности пользователей каждого региона зависит от часового пояса, которому этот регион принадлежит.

Запросы пользователей r-региона поступают в балансировщик нагрузки $РБН_r$ между серверами этого региона. Если все r-серверы заняты обработкой, то запрос помещается в очередь $РБН_r$. Продвижение очереди – по завершению обработки одним из r-серверов. Способ пополнения r-очереди новым запросом зависит от времени суток и принятого метода межрегиональной балансировки. В случае централизованной балансировки перераспределением запросов между регионами по сети обмена трафиком ведает межрегиональный балансировщик нагрузки МРБН. Он выполняет балансировку в динамике работы системы с учетом текущего состояния очередей всех $РБН_r$. Но возможна и децентрализованная балансировка, когда МРБН отсутствует.

Модельный эксперимент проводится для выделенных ранее трех регионов на специально разработанной кросс-системе моделирования [130], реализованной на одном ПК.

Принятые ограничения:

1. В соответствии с предыдущим, пользователи каждого региона активны в течение 2/3 суток (16 час. реального времени). Время «сна» – 1/3 суток (8 час.). Сдвиг часовых поясов между соседними регионами равен 1/6 суток (4 час. реального времени). Сутки делятся на 6 одинаковых 4-часовых временных интервалов t_i, $i \in (\overline{1,6})$. Активность пользователей по регионам отвечает рис. 5.4. Где интервалы времени:

t_1 – запросы поступают только от пользователей региона 1;

t_2 – запросы поступают от пользователей регионов 1 и 2;

$t_{3,4}$ – запросы поступают от пользователей всех регионов;

t_5 – запросы поступают от пользователей регионов 2 и 3;

t_6 – запросы поступают только от пользователей региона 3.

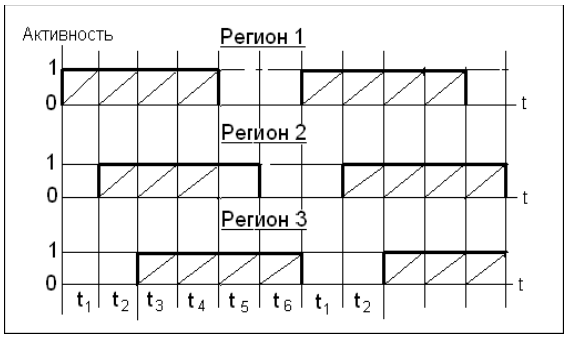


Рис. 5.4. Состояния активности пользователей по регионам

2. В r-регионе, $r \in \{\overline{1,3}\}$: число серверов n_r=r; число пользователей N_r=30·n_r; объем базы данных $V_{БД_r} = (5\text{GB}) \cdot n_r$.

3. В момент активизации любого r-региона его очередь пополняется N_r запросами своего региона. Балансировка нагрузки между отдельными IO_{rl}-серверами r-региона выполняется по круговому алгоритму [91]. Очередь запросов в каждом сервере – единичная. Поэтому, если к моменту активизации r-региона очередь $РБН_r$ нулевая и все его серверы завершили обработку, то начальная длина его очереди в этот момент $(L_r)_{нач} = N_r - 2n_r$. Первоочередной запрос из очереди $РБН_r$ сразу передается в завершивший обработку rl-сервер.

4. Моделирование начинается в момент активизации региона 1 (t = 0). Длины очередей всех регионов к этому моменту – нулевые, и все серверы – «свободны». Время «разгона» модели, по истечении которого начинаются измерения – 2 «модельных суток».

5. Исследования проводятся на подмножестве запросов теста TPC H [11] без операций записи (14 запросов в таблице 5.5). По завершении обработки запроса активного r-региона, $РБН_r$ случайным образом выбирает новый запрос из подмножества запросов своего региона и пополняет им свою очередь. Если же r-регион пассивен, то указанное действие не производится. Дальнейшее пополнение оче-

реди $РБН_r$ зависит от принятого способа балансировки. Сравниваются два метода балансировки – при наличии и отсутствии МРБН.

Для проведения модельного эксперимента были предложено два метода балансировки нагрузки в территориально распределенных СУБД: централизованный и децентрализованный. В обоих методах в процессе балансировки ведется подсчет весов очередей запросов в отдельных серверах. При определении этих весов учитывается следующее [80]. «Реально на функционирование системы влияет множество факторов, внешних и внутренних. Раскрыть их взаимосвязь затруднительно. Само множество внутренних факторов остается неизвестным. Учитываемые далее *внешние параметры* считаются первичными в том смысле, что они определяют степень влияния на эффективность системы всех факторов в совокупности.»

Централизованный метод реализуется согласно схеме, показанной на рис. 5.5.

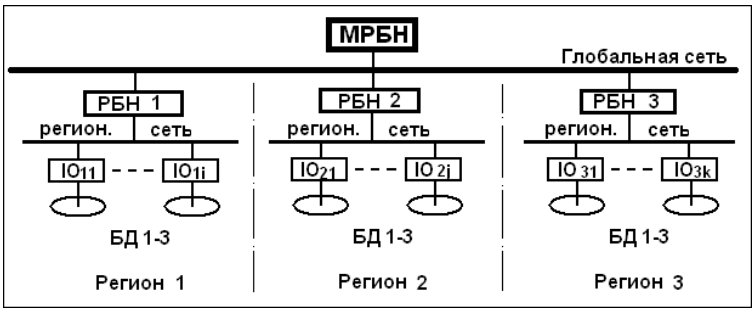


Рис. 5.5. Схема модели для централизованного метода

В динамике работы в каждой очереди могут находиться запросы из разных регионов. Результат обработки запроса передается любым IO_{rl} своему $РБН_r$, который отсылает ответ по истинному адресу с указанием региона. Затем продвигает свою очередь и подсчитывает ее новый вес W_r, значение которого предлагается определять как

$$W_r = \left\{ \left[\sum_{i=1}^{q} \sum_{j=1}^{L_i} (V_\sigma)_{ij} \right] \Big/ \left[L_r \cdot \left(V_{БД_i} \right)_{\max} \right] \right\} \cdot \left\{ (n_i)_{\max} / n_r \right\};\ r, i \in \{\overline{1,q}\}$$

Внешние параметры модели в данном случае: L_i – число запросов из i-региона в текущей r-очереди длиной L_r; $V_{БД_i}$ – объем $БД_i$ (базы данных i-региона); $(V_\sigma)_{ij}$ – суммарный объем отношений j-запроса ($j = \overline{1, L_i}$) к $БД_i$, подлежащих обработке в r-очереди (таблица 5.6).

Вычисленное значение передается МРБН, который обеспечивает равенство весов всех очередей. Перераспределение запросов между r-очередями реализуется

Таблица 5.6. Найденные значения V_σ,GB для $V_{БД}$ = 5, 10, 15GB теста TPC-H

№ за- $V_{БД}$, проса GB	1	2	3	4	5	6	7	8	9	10	11	12	13	14
5	3,21	0,65	4,05	3,95	4,06	3,21	4,06	4,16	4,59	4,05	1,61	3,95	0,85	3,31
10	6,46	1,30	8,16	7,95	8,18	6,46	8,18	8,38	9,25	8,16	3,23	7,95	1,70	6,67
15	9,74	1,95	12,31	11,98	12,33	9,74	12,33	12,64	13,94	12,31	4,87	11,98	2,56	10,05

следующим образом. Всякий раз по получении новой информации МРБН вычисляет средний вес $M(W_r)=\sum_{r=1}^{q} W_r/q$. Затем он связывается с $РБН_r$, для которых $\Delta_r = W_r - M(\mathrm{W}_r) > 0$, с требованием передачи ему одного запроса из конца их r-очереди. Полученный запрос он передает в $РБН_r$ с минимальным W_r. Цикл повторяется с перерывами на пересылку ответов от $РБН_r$ на запросы из других регионов.

В ***децентрализованном методе*** МРБН отсутствует (рис. 5.6), каждый РБН ведает информацией о нагрузке в других регионах и может высылать требования на передачу ему запросов.

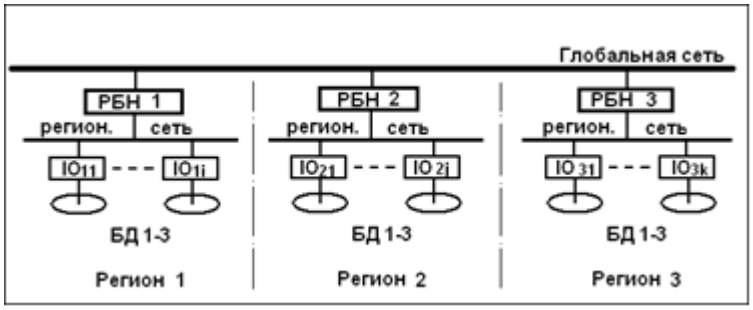


Рис. 5.6. Схема модели для децентрализованного метода

По-прежнему в каждом регионе обрабатываются запросы из разных регионов. Но «чужие» запросы могут поступать только по исчерпании очереди «своих» запросов. Как только некоторая L_r=0, $РБН_r$ высылает требование $РБН_k$, $k \neq r$, $r,k \in \{\overline{1,q}\}$), $L_k \neq 0$, передать ему значение веса W_k своей очереди. Этот вес определяется как

$$W_k = \left\{ \left[\sum_{j=1}^{L_k} \left(V_\sigma\right)_{rj} \right] \Big/ \left[L_r \cdot \left(V_{DB_i}\right)_{\max} \right] \right\} \cdot \left\{ \left(n_i\right)_{\max} / n_r \right\}$$

Получив нужные сведения, $РБН_r$ отправляет в адрес $РБН_k$ с максимальным весом W_k требование переслать его первоочередной запрос в $РБН_r$. Результат выполнения запроса он возвращает $РБН_k$. В этом методе веса очередей, исчерпавших запросы своего региона, не подсчитываются вплоть до момента новой инициализации региона (начала поступления новых запросов).

Результаты моделирования. Для каждого метода было выполнено по 3 запуска модели со случайным потоком запросов, генерируемых по равномерному закону. Каждый запуск проводился в течение 3 «модельных суток», из них 2 суток – на «разгон» модели. В таблицах 5.7 и 5.8 приведены усредненные по 3 запускам результаты кросс-моделирования, полученные по каждому методу и региону с использованием данных таблиц 5.5 и 5.6.

Таблица 5.7. Количество обработанных запросов за сутки

Метод	Регион №1	Регион №2	Регион №3	Итого
Без балансировки	7 152	7 297	7 212	21 661
Централизованный	13 130	10 369	9 387	32 886
Децентрализованный	16 812	9 314	8 845	34 971

Таблица 5.8. Увеличение пропускной способности по регионам, %

Метод	Регион №1	Регион №2	Регион №3	В среднем
Централизованный	84	42	30	52
Децентрализованный	135	28	23	61

Как следует из этих таблиц, оба рассмотренных метода достаточно эффективны. При этом централизованная балансировка обеспечивает более равномерное распределение числа обработанных запросов по регионам, а децентрализованная – большее увеличение пропускной способности системы в целом. На множестве взаимосвязанных регионов прирост пропускной способности для потока региональных запросов всегда повышается при переходе к региону с меньшей ресурсоемкостью и достигает максимума в регионе с минимальными IT-ресурсами.

5.3. Выводы по главе 5

Установленный в разделе 5.1 выигрыш не должен зависеть от объема базы данных, т.к. по условию все компоненты времени обработки теста, как для Clusterix-N, так и для Clusterix-G, прямо пропорциональны $V_{БД}$. Найденная теоретическая оценка повышения быстродействия Clusterix-G над Clusterix-N справедлива только для сравнительно медленной сети GigabitEthernet. Нетрудно подсчитать, что использование сети 10GigabitEthernet повысит быстродействие СУБД Clusterix-N примерно в 2,5 раза, а СУБД Clusterix-G – всего в 1,4 раза. Так что выигрыш по производительности при переходе от Clusterix-N к Clusterix-G снизится до 22%. А при использова-

нии сети Infiniband станет еще менее значительным. Поэтому основным эффектом от сжатия баз данных с использованием графических ускорителей следует считать возможность повышение объемов обрабатываемых баз данных при использовании тех же хранилищ без существенного снижения производительности. Это приведет к дополнительному повышению эффективности.

Полученные в разделе 5.2 результаты кросс-моделирования говорят о существенном повышении пропускной способности территориально распределенных СУБД консервативного типа при использовании ресурсов существующих глобальных сетей для целей балансировки нагрузки таких СУБД. Сделанный вывод должен сохранить силу и в реальных условиях функционирования. Для уточнения вопроса о влиянии введенных ограничений на достижимую степень эффективности требуется моделирование, более близкое к натурному. Провести его пока затруднительно.

Установлено, что централизованный метод балансировки позволяет загрузить вычислительную систему более равномерно, нежели децентрализованный метод. Последний проще в реализации, обладает бóльшей надежностью (выход из строя межрегионального балансировщика полностью нарушает балансировку) и нацелен на максимум помощи «слабейшему» региону, что не всегда приемлемо.

Эффективность работы регионального сервера можно существенно повысить, используя все процессорные ядра многоядерных узлов, как это представлено в технологии PerformSys (глава 4), либо используя технологии Clusterix-N (главы 2, 3) и Clusterix-G для обработки сложных аналитических запросов.

ЗАКЛЮЧЕНИЕ

В диссертационной работе была рассмотрена проблема создания высокоэффективных (по критерию «производительность/стоимость») открытых (open source) консервативных СУБД класса BigData с регулярным планом обработки запросов на кластерных платформах с GPU-ускорителями. Выполнена архитектурно-алгоритмическая разработка и программная реализация исследовательских проектов четырех эффективных версий системы Clusterix-N (с сосредоточенной и распределенной динамической сегментацией отношений) и PerformSys (как удачной альтернативы систем Clusterix-N при умеренных объемах баз данных и большом количестве обслуживаемых пользователей). Экспериментально найдены сравнительные оценки их эффективности.

Следуя методологии конструктивного моделирования систем, решение для Clusterix-N искалось итеративно из условия конкурентоспособности (по эффективности) с перспективной системой Spark. При этом переходы между итерациями ***S***-моделирования были определены найденной в самом процессе моделирования внешней (математической) моделью. Характерно, что результаты любой из итераций приемлемы для практического использования при тех или иных финансовых ограничениях. Поставленные цели работы достигнуты.

Основные результаты работы

1. Разработан способ претрансляции запросов к регулярному плану, основанный на таком дроблении исходного запроса на SQL-фрагменты, который, в отличие от использованного для Clusterix метода, позволяет использовать его с различными инструментальным СУБД.

2. Предложена интерпретация методологии конструктивного моделирования систем применительно к задаче моделирования процесса синтеза консервативных СУБД класса BigData, которая, в отличие от ранее использованной для Clusterix-подобных СУБД, позволяет добиться большей эффективности системы.

3. Предложен и реализован метод параллельной обработки селективных запросов в СУБД Clusterix-N на уровне IO, основанный на поблочной выборке из СУБД MySQL, что, в отличие от ранее реализованного метода в Clusterix-подобных системах, позволяет использовать все процессорные ядра всех узлов IO для обработки одного селективного запроса с полной загрузкой процессорных ядер.

4. Для СУБД Clusterix-N предложены и реализованы методы сосредоточенной и распределенной динамической сегментации промежуточных/временных отношений с применением GPU-ускорителей, основанные 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 и Clusterix-M, позволяет существенно ускорить операции хеширования, загрузить все процессорные ядра всех узлов уровня JOIN и более эффективно использовать сеть.

5. Предложен и реализован в PerformSys метод распределения потока запросов по процессорным ядрам кластерной платформы с их полной загрузкой, основанный на стратегии «запрос на ядро +1», что, в отличие от реализации в MySQL Router [19], позволяет передавать в узлы ровно столько запросов, сколько они могут эффективно обработать.

6. Выявлена возможность дальнейшего повышения эффективности Clusterix-подобных систем (переход от Clusterix-N к архитектуре Clusterix-G), основанного на работе со сжатыми БД, что, в отличие от применения GPU для выполнения SQL запросов, предложенного в [20, 21], позволяет увеличить объемы хранимых данных и ускорить их передачу по сети.

7. Предложены методы межрегиональной балансировки нагрузки для территориальной распределенных консервативных СУБД, основанные на подсчете веса очередей и времени активности каждого региона, что, в отличие от общепринятых методов балансировки нагрузки [22], позволяет более равномерно распределять нагрузку по регионам и увеличить эффективность эксплуатации территориально распределенных СУБД.

В процессе проведенных исследований содержательная теория консервативных СУБД кластерного типа была пополнена следующими элементами:

1. Инструментальная СУБД MySQL должна настраиваться на полную загрузку всех процессорных ядер, по-разному для различных функциональных узлов.

2. Близкой к оптимальной конфигурации монокластера является «совмещенная симметрия».

3. При выполнении динамической сегментации отношений, во избежание перегрузки сети, информация должна передаваться блоками, формируемыми на множестве процессоров IO и JOIN.

4. Операции *select-project, join* и агрегирования должны выполняться параллельно на полном собственном множестве процессорных ядер.

Соответствие работы специальности ВАК 05.13.11. Работа отвечает следующим пунктам 1, 4, 8 и 9 паспорта специальности ВАК 05.13.11.

Квалификационный признак работы. В работе изложены новые научно обоснованные технические решения и разработки, имеющие существенное значение для развития отечественных технологий параллельных СУБД.

СПИСОК СОКРАЩЕНИЙ И УСЛОВНЫХ ОБОЗНАЧЕНИЙ

MB Megabyte, мегабайт

Gb Gigabit, гигабит

GB Gigabyte, гигабайт

TB Terabyte, терабайт

PB Petabyte, петабайт

CPU Central Processing Unit

GPU Graphics Processing Unit

RAM Random Access Memory

ПК Персональный компьютер

XML eXtensible Markup Language

TPC Transaction Processing Performance Council

ОС Операционная система

ПО Программное обеспечение

СУБД Система управления базами данных

БД База данных

ПТ Представительский тест

IT Information Technology, Информационные технологии

КМС Конструктивное моделирование систем

СЛОВАРЬ ТЕРМИНОВ

ACID – Atomicity, Consistency, Isolation, Durability – описывает требования к транзакционной системе (например, к СУБД), обеспечивающие наиболее надёжную и предсказуемую её работу.

AWS – Amazon Web Services – инфраструктура платформ облачных веб-сервисов, представленная компанией Amazon. В AWS представлены сервисы аренды виртуальных серверов, предоставления вычислительных мощностей, хранения данных (файловый хостинг, распределённых хранилищ данных) и т.п.

OLAP – Online Analytical Processing – интерактивная аналитическая обработка – технология обработки данных, заключающаяся в подготовке суммарной (агрегированной) информации на основе больших массивов данных, структурированных по многомерному принципу.

OLTP – Online Transaction Processing – транзакционная система – обработка транзакций в реальном времени. Способ организации БД, при котором система работает с небольшими по размерам транзакциями, но идущими большим потоком, и при этом клиенту требуется от системы минимальное время отклика.

TPC-H – эталонный тест производительности задач поддержки принятия решений, характеризующихся большим объемом данных и сложными запросами.

БД консервативного типа – БД с редкими обновлениями в специально отведенное время.

PerformSys, Clusterix-N – предлагаемые в диссертации технологии построения консервативных СУБД повышенных объемов как более эффективные альтернативы Clusterix-M.

Clusterix, Clusterix-M – предыдущие разработки в коллективе КНИТУ-КАИ консервативных СУБД на платформе вычислительных кластеров с регулярным планом обработки запросов и равномерным распределением БД по узлам кластера.

СПИСОК ЛИТЕРАТУРЫ

1. Егоров Д. Переписать базу сообщений с нуля и выжить // Блог vk.com. 2017. URL: https://vk.com/blog/messages-database (дата обращения: 10.08.2018).
2. Яндекс. Поиск Яндекса с инженерной точки зрения. Лекция в Яндексе // Хабр. 2016. URL: https://habr.com/company/yandex/blog/312716/ (дата обращения: 10.08.2018).
3. Wikipedia contributors. EOSDIS // Wikipedia, The Free Encyclopedia. 2017. URL: https://en.wikipedia.org/w/index.php?title=EOSDIS&oldid=808578344 (дата обращения: 10.08.2018).
4. Wikipedia contributors. Online transaction processing // Wikipedia, The Free Encyclopedia. 2018. URL: https://en.wikipedia.org/w/index.php?title=Online_transaction_processing&oldid=846851521 (дата обращения: 10.08.2018).
5. Xu Y., Kostamaa P., Zhou X., Chen L. Handling data skew in parallel joins in shared-nothing systems // ACM SIGMOD international Conference on Management of Data. Canada. 2008. pp. 1043-1052.
6. Лепихов А.В. Параллельная обработка запросов в СУБД для кластерных вычислительных систем, Отчет в рамках гранта МК-3535.2009.9.
7. Wikipedia contributors. Online analytical processing // Wikipedia, The Free Encyclopedia. 2018. URL: https://en.wikipedia.org/w/index.php?title=Online_analytical_processing&oldid=850545800 (дата обращения: 08.10.2018).
8. Microsoft. Parallel Query Processing // Resources and Tools for IT Professionals | TechNet. 2018. URL: https://technet.microsoft.com/en-us/library/ms178065(v=sql.105).aspx (дата обращения: 05.04.2018).
9. Lenovo. System x3950 X6 Rack Server // Официальный сайт Lenovo в России. 2017. URL: https://www3.lenovo.com/ru/ru/data-center/servers/mission-critical/System-x3950-X6/p/WMD00000002 (дата обращения: 15.07.2018).
10. Lenovo System x3950 X6 // TPC-H Result Highlights. 2016. URL: http://www.tpc.org/3321 (дата обращения: 10.08.2018).
11. Transaction Processing Performance Council (TPC). TPC Benchmark H Standard Specification Revision 2.17.3 // TPC. 2017. URL: http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.3.pdf (дата обращения: 10.10.2017).
12. Oracle Exadata Database Machine X7 // Oracle Россия и СНГ. 2018. URL: https://www.oracle.com/ru/engineered-systems/exadata/database-machine-x7/index.html (дата обращения: 10.08.2018).
13. Benchmarking Postgres-XL // 2ndQuadrant. 2015. URL: https://blog.2ndquadrant.com/benchmarking-postgres-xl/ (дата обращения: 23.04.2018).
14. Российская отрасль СУБД продвигается на «слонах» // CONNECT, № 5-6, 2017. С. 34-38.
15. Oracle. The MySQL Plugin API // MySQL Documentation. 2018. URL: https://dev.mysql.com/doc/refman/5.7/en/plugin-api.html (дата обращения: 09.04.2018).
16. Абрамов Е.В. Параллельная СУБД Clusterix. Разработка прототипа и его натурное исследование // Вестник КГТУ им. А.Н. Туполева., № 2, 2006. С. 50-55.
17. Классен Р.К. Повышение эффективности параллельной СУБД консервативного типа на кластерной платформе с многоядерными узлами // Вестник КГТУ им. А.Н.Туполева, № 1, 2015. С. 115-121.
18. Райхлин В.А., Минязев Р.Ш. Мультикластеризация распределенных СУБД консервативного типа // Нелинейный мир, Т. 9, № 8, 2011. С. 473-481.
19. Oracle Corporation. MySQL Router 8.0 // MySQL Documentation. 2018. URL: https://dev.mysql.com/doc/mysql-router/8.0/en/ (дата обращения: 11.08.2018).
20. Rauhe H. Finding the Right Processor for the Job Co-Processors in a DBMS, Ilmenau University of Technology, Ilmenau, Dissertation urn:nbn:de:gbv:ilm1-2014000240, 2014.
21. Bres S. Efficient Query Processing in Co-Processor-accelerated Databases, University of Magdeburg, Magdeburg, Dissertation 2015.

22. Load Balancing Methods for Every Application // Peplink SD-WAN. Protecting Business Continuity. 2018. URL: https://www.peplink.com/technology/load-balancing-algorithms/ (дата обращения: 11.08.2018).
23. TOP500 // TOP500 Supercomputer Sites. 2018. URL: https://www.top500.org/ (дата обращения: 09.03.2018).
24. ASCI Red | TOP500 Supercomputer Sites // TOP500 Supercomputer Sites. URL: https://www.top500.org/system/168753 (дата обращения: 09.03.2018).
25. ASCI Red // Википедия. 2013-2017. URL: http://ru.wikipedia.org/?oldid=87421680
26. GeForce 700 series // Википедия. 2012-2018. URL: https://en.wikipedia.org/wiki/GeForce_700_series#GeForce_700_(7xx)_series (дата обращения: 09.03.2018).
27. Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway | TOP500 Supercomputer Sites // TOP500 Supercomputer Sites. URL: https://www.top500.org/system/178764 (дата обращения: 09.03.2018).
28. Amazon. Amazon Web Services 2018. URL: https://aws.amazon.com (дата обращения: 30.04.2018).
29. Лаборатория Параллельных Информационных Технологий, НИВЦ МГУ. Задачи для суперкомпьютеров, суперкомпьютерные приложения // PARALLEL.RU - Информационно-аналитический центр по параллельным вычислениям. 2018. URL: https://parallel.ru/research/apps.html (дата обращения: 22.03.2018).
30. Большие данные // Википедия. 2011-2017. URL: http://ru.wikipedia.org/?oldid=89762670 (дата обращения: 09.03.2018).
31. Turck M. Firing on All Cylinders: The 2017 Big Data Landscape // Matt Turck. 2017. URL: http://mattturck.com/bigdata2017/ (дата обращения: 09.03.2018).
32. Paradigm4: Creators of SciDB a computational DBMS // Paradigm4. 2018. URL: https://www.paradigm4.com/ (дата обращения: 25.02.2018).
33. In-Memory Database VoltDB // VoltDB. 2018. URL: https://www.voltdb.com/ (дата обращения: 25.02.2018).
34. Postgres-XL | Open Source Scalable SQL Database Cluster // Postgres-XL. 2018. URL: https://www.postgres-xl.org/ (дата обращения: 25.02.2018).
35. SQL Server pricing // Microsoft. 2018. URL: https://www.microsoft.com/en-US/sql-server/sql-server-2017-pricing (дата обращения: 04.02.2018).
36. Hellerstein J.M., Stonebraker M., Hamilton J. Architecture of a Database System // Foundations and Trends in Databases. 2007. Vol. 1. No. 2. pp. 141-259.
37. Stonebraker M. The case for shared nothing // Database Engineering Bulletin, Vol. 9, No. 1, 1986. pp. 4-9.
38. Соколинский Л.Б. Методы организации параллельных систем баз данных на вычислительных системах с массовым параллелизмом, Челябинский государственный университет, Челябинск, Диссертация 2003.
39. Copeland G.P., Keller T.A. Comparison Of High-Availability Media Recovery Techniques // Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data. Portland, Oregon. 1989. pp. 98-109.
40. Sokolinsky L., Axenov O., Gutova S. Omega: The Highly Parallel Database System Project // Proceedings of the First East-European Symposium on Advances in Database and Information Systems (ADBIS'97), St.-Petersburg, Vol. 2, September 2-5 1997. pp. 88-90.
41. Rahm E. A Framework for Workload Allocation in Distributed Transaction Processing Systems // Journal of Systems and Software, Vol. 18, 1992. pp. 171-190.
42. Григорьев А.Ю., Плутенко А.Д., Плужников Л.В., Ермаков Е.Ю., Цвященко Е.В., Пролетарская А.В. Теория и практика анализа параллельных систем баз данных. Владивосток: Дальнаука, 2015. 336 с.
43. Российская СУБД Postgres Pro // Postgres Professional. 2018. URL: https://postgrespro.ru/products/postgrespro (дата обращения: 03.05.2018).

44. DB-Engines Ranking - popularity ranking of database management systems // DB-Engines. 2018. URL: https://db-engines.com/en/ranking (дата обращения: 29.03.2018).
45. PostgreSQL 10 Released // PostgreSQL. 2017. URL: https://www.postgresql.org/about/news/1786/ (дата обращения: 06.02.2018).
46. Parallel Execution with Oracle Database 18c Fundamentals // ORACLE WHITE PAPER. 2018. URL: http://www.oracle.com/technetwork/database/bi-datawarehousing/twp-parallel-execution-fundamentals-133639.pdf
47. Oracle. Oracle Technology Global Price List 2018. URL: http://www.oracle.com/us/corporate/pricing/technology-price-list-070617.pdf (дата обращения: 06.04.2018).
48. Chapter 21 MySQL NDB Cluster 7.5 and NDB Cluster 7.6 2018. URL: https://dev.mysql.com/doc/refman/5.7/en/mysql-cluster.html (дата обращения: 06.03.2018).
49. Pgpool-II // PostgreSQL wiki. 2018. URL: https://wiki.postgresql.org/index.php?title=Pgpool-II&oldid=20968 (дата обращения: 12.04.2018).
50. Slony // PostgreSQL wiki. 2018. URL: https://wiki.postgresql.org/index.php?title=Slony&oldid=28088 (дата обращения: 12.04.2018).
51. 2ndQuadrant. PostgreSQL vs MySQL 2018. URL: https://www.2ndquadrant.com/en/postgresql/postgresql-vs-mysql/ (дата обращения: 11.04.2018).
52. INTERNATIONAL STANDARD. ISO/IEC 9075-1 Information technology - Database languages - SQL - Part 1: Framework (SQL/Framework), 2008.
53. Gray , Reuter A. Transaction Processing: Concepts and Techniques. 1st ed. Morgan Kaufmann Publishers Inc., 1992.
54. Postgres-XL // 2ndQuadrant. 2018. URL: https://www.2ndquadrant.com/en/resources/postgres-xl/ (дата обращения: 16.04.2018).
55. Sharp M. Scaling PostgreSQL on Postgres-XL // PGConf.ru. 2015. URL: https://pgconf.ru/media2015c/sharp.pdf (дата обращения: 16.04.2018).
56. Amazon EC2 Instance Types // Amazon Web Services (AWS). 2018. URL: https://aws.amazon.com/ec2/instance-types/?nc1=h_ls (дата обращения: 23.04.2018).
57. Пан К.С. Методы внедрения фрагментного параллелизма в последовательную СУБД с открытым исходным кодом, Южно-уральский государственный университет, Челябинск, Диссертация 2013.
58. Соколинский Л.Б., Цымблер М.Л. Проект создания параллельной СУБД Омега на базе суперкомпьютера МВС-100/1000 // Телематика'98: Тез. докл. Всероссийск. науч.-метод. конф. (7-10 июня 1998 г., Санкт-Петербург). -СПб: Вузтелекомцентр. 1998. С. 154-155.
59. Соколинский Л.Б. Организация параллельного выполнения запросов в многопроцессорной машине баз данных с иерархической архитектурой // Программирование, № 6, 2001. С. 13-29.
60. СУБД Postgres Pro // Официальный сайт оператора единого реестра российских программ для электронных вычислительных машин и баз данных в информационно-телекоммуникационной сети "Интернет". 2016. URL: https://reestr.minsvyaz.ru/reestr/65273/ (дата обращения: 06.05.2018).
61. Российская СУБД Postgres Pro // Компания Postgres Professional. 2018. URL: https://postgrespro.ru/products/postgrespro (дата обращения: 06.05.2018).
62. Сопоставление возможностей PostgreSQL и PostgresPro версии 9.5 // Компания Postgres Professional. 2018. URL: https://postgrespro.ru/products/postgrespro/9.5/comparison (дата обращения: 06.05.2018).
63. Raikhlin V.A. Simulation of Distributed Database Machines // Programming and Computer Software, Vol. 22, No. 2, 1996. pp. 68-74.
64. Martin J. Computer database organization. 2nd ed. New Jersey 07632: Prentice-Hall, Inc., Englewood Cliffs, 1977. 713 pp.
65. MapReduce // Википедия. 2018. URL: https://ru.wikipedia.org/?oldid=94040380 (дата обращения: 18.07.2018).

66. Shankland S. Google spotlights data center inner workings // CNet. 2008. URL: https://www.cnet.com/news/google-spotlights-data-center-inner-workings/ (дата обращения: 2018.07.2018).
67. Hadoop // Википедия. 2018. URL: https://ru.wikipedia.org/?oldid=95122995 (дата обращения: 17.07.2018).
68. Shvachko K. Apache Hadoop. The Scalability Update // File Systems, Vol. 36, No. 3, 2011. pp. 7-13.
69. Apache Spark // Википедия. 2018. URL: https://ru.wikipedia.org/?oldid= 94436956 (дата обращения: 09.08.2018).
70. Apache Spark - Unified Analytics Engine for Big Data // Apache Spark. 2018. URL: https://spark.apache.org/ (дата обращения: 26.10.2018).
71. Oracle. MySQL // MySQL. 2017. URL: https://www.mysql.com/ (дата обращения: 19.05.2017).
72. Charvet F., Pande A. Database Performance Study 2003. URL: https://mis.umsl.edu/files/pdfs/TuningPaperV5.pdf (дата обращения: 13.07.2018).
73. Oracle Corporation and/or its affiliates. The MyISAM Storage Engine // MySQL Documentation. 2018. URL: https://dev.mysql.com/doc/refman/5.6/en/myisam-storage-engine.html (дата обращения: 13.07.2018).
74. Oracle Corporation and/or its affiliates. The InnoDB Storage Engine // MySQL Documentation. 2018. URL: https://dev.mysql.com/doc/refman/5.7/en/innodb-storage-engine.html (дата обращения: 13.07.2018).
75. Oracle Corporation and/or its affiliates. Table Locking Issues // MySQL Documentation. 2018. URL: https://dev.mysql.com/doc/refman/5.6/en/table-locking.html (дата обращения: 13.07.2018).
76. Oracle. MySQL 5.7 Reference Manual. Buffer Pool // MySQL Documentation. 2017. URL: https://dev.mysql.com/doc/refman/5.7/en/innodb-buffer-pool.html (дата обращения: 24.03.2017).
77. Oracle Corporation and/or its affiliates. Server Option, System Variable, and Status Variable Reference // MySQL Documentation. 2018. URL: https://dev.mysql.com/doc/refman/5.7/en/server-option-variable-reference.html (дата обращения: 13.07.2018).
78. Oracle Corporation and/or its affiliates. Using Option Files // MySQL Documentation. 2018. URL: https://dev.mysql.com/doc/refman/5.7/en/option-files.html (дата обращения: 13.07.2018).
79. Райхлин В.А. Конструктивное моделирование систем. Казань: Изд-во «Фән» («Наука»), 2005.
80. Райхлин В.А., Вершини И.С., Минязев Р.Ш., Гибадуллин Р.Ф. Конструктивное моделирование систем информатики. Казань: Изд-во «Фән» («Наука»), 2016. 312 с.
81. Никтин Е.П. Объяснение – функция науки. Москва: Наука, 1970.
82. Конторов Д.С. Внимание – системотехника. Москва: Радио и связь, 1993.
83. Хакинг Ян. Представление и вмешательство. Начальные вопросы философии естественных наук. Москва: Логос, 1998.
84. Анохин П.К. Принципиальные вопросы общей теории функциональных систем // В кн.: Принципы системной организации функций / ред. Анохин П.К. Москва: Наука, 1973. С. 5-61.
85. Хакен Г. Основные понятия синергетики //Синергетическая парадигма. М. 2000.
86. Шрейдер Ю.А., Шаров А.А. Системы и модели. М.: Радио и связь, 1982.
87. Дружинин В.В., Конторов Д.С. Проблемы системологии (проблемы теории сложных систем). М.: Сов. Радио, 1976. 296 с.
88. Райхлин В.А., Абрамов Е.В. К теории моделей синтеза кластеров баз данных // Вестник КГТУ им. А.Н. Туполева, № 1, 2004. С. 44-49.
89. Райхлин В.А., Классен Р.К. Сравнительно недорогие гибридные технологии консервативных СУБД больших объемов // Информационные технологии и вычислительные системы, Т. 68, № 1, 2018. С. 46-59.
90. Mono Project. Cross platform, open source.NET framework // Mono. 2018. URL: https://www.mono-project.com/ (дата обращения: 12.07.2018).
91. Минязев Р.Ш. Распределение потока запросов в параллельных СУБД на платформе вычислительных кластеров // Нелинейный мир, Т. 10, № 3, 2012. С. 173-179.

92. GUID // Википедия. 2018. URL: https://ru.wikipedia.org/?oldid=92011990 (дата обращения: 25.07.2018).
93. A Universally Unique IDentifier (UUID) URN Namespace // Internet Engineering Task Force. 2005. URL: http://www.ietf.org/rfc/rfc4122.txt (дата обращения: 25.07.2018).
94. Классен Р.К. Clusterix-N Wiki // Bitbucket. 2018. URL: https://bitbucket.org/rozh/clusterixn/wiki/Home (дата обращения: 22.12.2018).
95. Райхлин В.А., Классен Р.К. Моделирование процессов балансировки нагрузки в распределенных СУБД, использующих ресурсы сети RUNNet // Научный вестник НГТУ, № 4, 2015. С. 90-100.
96. Райхлин В.А., Минязев Р.Ш., Классен Р.К., Садовин А.В. Анализ возможных путей повышения эффективности параллельных СУБД консервативного типа на платформе вычислительных кластеров // Материалы Международной конференции и молодежной школы «Информационные технологии и нанотехнологии» (Конференция ИТНТ-2016). Самара. 2016. С. 965-970.
97. Райхлин В.А., Минязев Р.Ш., Классен Р.К. Эффективность консервативных СУБД больших объемов на кластерной платформе // Кибернетика и программирование, № 5, Ноябрь 2018. С. 44-62.
98. Классен Р.К. Ускорение операций хеширования с применением графических ускорителей // Вестник КГТУ им. А.Н.Туполева, № 1, 2018. С. 134-141.
99. Karwin Software Solutions LLC. Load Data Fast! // SlideShare. 2017. URL: https://www.slideshare.net/billkarwin/load-data-fast (дата обращения: 15.05.2018).
100. Классен Р.К. Разработка и исследование консервативной СУБД Clusterix-N класса «BigData» на платформе GPU-кластера // Системы компьютерной математики и их приложения, № 19, 2018. С. 163-171.
101. Oracle Corporation and/or its affiliates. MySQL 8.0 Release Notes // MySQL Documentation. 2019. URL: https://dev.mysql.com/doc/relnotes/mysql/8.0/en/ (дата обращения: 25.01.2019).
102. Clusterix-N [Электронный ресурс] [2018]. URL: https://bitbucket.org/rozh/clusterixn
103. Классен Р.К. Программа региональной балансировки нагрузки к базе данных консервативного типа на кластерной платформе «PerformSys». Свидетельство о государвенной регистрации программы для ЭВМ №2017611785 от 09.02.2017.
104. Классен Р.К. Использование мощностей кластерной платформы с многоядерными узлами для работы с базой данных консервативного типа // Международной молодежной научной конференции XXII Туполевские чтения (школа молодых ученых). Казань. 2015. Т. 4. С. 263-269.
105. Дубовцев А.В. Microsoft.NET в подлиннике. БХВ-Петербург, 2004. 704 с.
106. Паркер Т., Сиян К. TCP/IP для профессионалов. Спб.: Питер, 2004. 859 с.
107. Классен Р.К. Документация PerformSys 2018. URL: https://github.com/rozh1/PerformSys/wiki/ (дата обращения: 31.12.2018).
108. Shafranovich Y., SolidMatrix Technologies Inc. RFC 4180 - Common Format and MIME Type for Comma-Separated Values (CSV) Files // IETF Tools. 2005. URL: https://tools.ietf.org/html/rfc4180 (дата обращения: 11.07.2018).
109. Base64 // Википедия. 2018. URL: https://ru.wikipedia.org/?oldid=91876119 (дата обращения: 12.07.2018).
110. W3C (MIT, ERCIM, Keio). Extensible Markup Language (XML) 1.0 // World Wide Web Consortium (W3C). 2008. URL: https://www.w3.org/TR/REC-xml/ (дата обращения: 12.07.2018).
111. Microsoft. Examples of XML Serialization // Microsoft Docs. 2017. URL: https://docs.microsoft.com/en-us/dotnet/standard/serialization/examples-of-xml-serialization (дата обращения: 12.07.2018).
112. Oracle Corporation and/or its affiliates. How MySQL Uses Threads for Client Connections // MySQL Documentation. 2018. URL: https://dev.mysql.com/doc/refman/5.7/en/connection-threads.html (дата обращения: 11.07.2018).

113. Oracle Corporation and/or its affiliates. MySQL Connector/NET Developer Guide // MySQL Documentation. 2018. URL: https://dev.mysql.com/doc/connector-net/en/ (дата обращения: 11.07.2018).
114. Классен Р.К. Анализ возможностей использования графических ускорителей в параллельных СУБД с полной активизацией процессорных ядер кластерной платформы // Вестник КГТУ им. А.Н.Туполева, № 4, 2016. С. 107-117.
115. Raikhlin V.A., Klassen R.K. 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. pp. 1-5.
116. Классен Р.К., Хисамиев Л.Р. Моделирование процессов балансировки нагрузки // Труды Междунар. конф. «XXI Туполевские чтения». Казань. 2013. Т. 1. С. 323-324.
117. Райхлин В.А., Классен Р.К. Моделирование процессов глобальной балансировки нагрузки в распределенных СУБД // Материалы XVI Международной научно-методической конференции «Информатика: проблемы, методология, технологии». Воронеж. 2016. Т. 4. С. 454-459.
118. CoGaDB – Column-oriented GPU-accelerated DBMS [Электронный ресурс] URL: http:// cogadb.cs.tu-dortmund.de/wordpress/ (дата обращения: 15.09.2016).
119. PGStrom 2016. URL: https://wiki.postgresql.org/index.php?title=PGStrom&oldid=25517 (дата обращения: 15.09.2016).
120. Беседин Д. Первый взгляд на DDR3. Изучаем новое поколение памяти DDR SDRAM, теоретически и практически // ixbt.com. 2007. URL: http://www.ixbt.com/mainboard/ddr3-rmma.shtml (дата обращения: 01.10.2016).
121. Петров С.В. Шины PCI, PCI Express. Архитектура, дизайн, принципы функционирования. СПб.: БХВ-Петербург, 2006. 321-322 с.
122. Blelloch G.E. Introduction to Data Compression. Pittsburgh: Carnegie Mellon University, 2013. 55 с.
123. Wenbin F., Bingsheng H., Qiong L. Database Compression on Graphics Processors // Proc. VLDB Endow., Vol. 3, No. 1-2, Sep 2010. pp. 670-680.
124. F5 Networks. Load Balancing 101: Nuts and Bolts // F5 Networks. 2017. URL: https://f5.com/Portals/ 1/Cache/Pdfs/2421/load-balancing-101-nuts-and-bolts.pdf (дата обращения: 16.07.2018).
125. Бершадский А.М., Курилов Л.С. Исследование стратегий балансировки нагрузки в системах распределённой обработки данных // Известия высших учебных заведений, № 4, 2009. С. 38-47.
126. Игнатенко Е.Г., Бессараб В.И. Алгоритм адаптивной балансировки нагрузки в кластерных системах // Моделювання та інформаційні технології, № 58, 2010. С. 142-150.
127. Mauro T. Choosing an NGINX Plus Load-Balancing Technique // NGINX. 2015. URL: https:// www.nginx.com/blog/choosing-nginx-plus-load-balancing-techniques/ (дата обращения: 16.07.2018).
128. Федеральная университетская компьютерная сеть России [Электронный ресурс] [2018]. URL: https://www.runnet.ru/ (дата обращения: 15.05.2018).
129. Российские точки обмена IP-трафиком // IX.RU. 2015. URL: http://www.ix.ru/ (дата обращения: 16.09.2015).
130. Классен Р.К. Кросс-система моделирования межрегиональной балансировки // GitHub. 2013. URL: https://github.com/rozh1/cluster_emul (дата обращения: 16.07.2018).

ПРИЛОЖЕНИЕ А. РАЗРАБОТАННЫЕ ДЛЯ НАЧАЛЬНОГО СОСТОЯНИЯ СЕРВИСНЫЕ СРЕДСТВА

А.1. Подсистема журналирования

Слежение за процессом работы системы, обработка ошибок, отладка системы требуют наличия подсистемы журналирования. Ее основная задача – ведение журнала работы системы для дальнейшего анализа.

Подсистема журналирования основана на свободной библиотеке NLog [5], которая позволяет гибко настраивать параметры [6]:

– Хранилище журнала (текстовый файл, БД и т.д.).
– Формат записи в журнале (дата, время, сообщение, уровень сообщения, название метода, из которого была вызвана запись в журнал, информация о исключении и т.д.).
– Способ записи сообщений в журнал (синхронный/асинхронный).
– Правила вывода сообщений на терминал (просто текст, цветной текст, уровень сообщения для вывода и т.д.) и др.

Библиотека NLog была выбрана в силу ряда причин. Во-первых, это возможность настройки вывода сообщений отдельно для файла и отдельно для терминала. Во-вторых, возможность использования асинхронных операций, что позволяет основной программе не ждать завершения записи в журнал, а продолжать работать остановок. В-третьих, возможность записи структурированных сообщений в БД. Все эти возможности позволили не разрабатывать свою систему журналирования, а использовать готовую.

Подсистема журналирования на основе NLog позволяет записывать в заданный файл и выводить на терминал события, происходящие в системе, с указанием даты/времени, уровня события и сообщения. Она ограничивает вывод сообщения в зависимости от его уровня. В модуле журналирования предусмотрено 6 уровней сообщений:

---

[5] NLog flexible & free open-source logging for.NET //NLog. 2017. URL: http://nlog-project.org/ (дата обращения: 15.01.2017).

[6] Configuration file //NLog Wiki. 2018. URL: https://github.com/nlog/nlog/wiki/Configuration-file (дата обращения: 23.07.2018).

1. TRACE – сообщения трассировки – служат для подробного журналирования хода работы программы.

2. DEBUG – сообщения отладки – подробные сообщения о том, что происходило в тот или иной момент работы программы.

3. INFO – информационные сообщения о основных событиях в работе программы. К таковым относятся, например, сообщения о инициализации, получения очередного запроса, завершения работы над запросом и т.д.

4. WARNING – предупреждения – записи о ошибках в работе, которые мало влияют на итоговый результат работы. Например, сообщение о недостатке оперативной памяти на одном из узлов.

5. ERROR – сообщения о ошибке во время работы программы. Например, ошибка передачи данных между узлами, ошибка соединения с СУБД и т.д.

6. FATAL – ошибка во время работы, возникает при неожиданном завершении программы в следствии непредвиденной ошибки.

Вывод сообщений можно ограничивать, изменяя параметр минимального уровня сообщения minlevel для каждого правила журналирования в отдельности. Так при минимальном minlevel=TRACE будут выведены и записаны все сообщения, а при уровне minlevel=ERROR будут выведены/записаны только сообщения уровней ERROR и FATAL.

А.2. Подсистема сбора статистики и визуализации

Оценка эффективности и корректности работы сложной системы невозможна без подсистемы сбора статистики и визуализации, которая могла бы дать ответ на вопрос: что происходит в тот или иной момент в системе?

Для этих целей и помощи в отладке был разработан модуль мониторинга Clusterix-N. Он позволяет: во-первых, определить, какие операции проводились над тем или иным запросом; время выполнения этих операций; узлы, на которых обрабатывался запрос; загрузку CPU и объем доступной RAM этих узлов в режиме реального времени; во-вторых, визуализировать работу системы.

Организация мониторинга. Мониторинг времени выполнения отдельных операций в Clusterix-N строится на системных таймерах. Системные таймеры вы-

браны в силу их точности (вплоть до тысячных долей миллисекунды). Их запуск и остановка встроены во все модули. Например, в модуле сетевого взаимодействия счетчик времени отправки данных запускается с вызовом предобработчика любого из пакетов, а останавливается и сохранятся в журнале после завершения отправки. Аналогично для всех операций с запросами. Перед передачей запроса драйверу СУБД запускается счетчик времени. Он останавливается с завершением выполнения запроса.

Журнал измерений формируется структурированными записями с полями:

- `Time` – отметка времени начала измерения.
- `Duration` – измеренное время выполнения операции в миллисекундах.
- `Operation` – измеряемая операция.
- `Module` – имя программного модуля.
- `QueryId` – идентификатор запроса, над которым выполняется операция.
- `SubQueryId` – идентификатор подзапроса, над которым выполнялась операция.
- `RelationId` – идентификатор отношения, над которым выполнялась операция.
- `From` – адрес источника.
- `To` – адрес приемника.

Из такой записи без труда можно получить информацию о времени запуска (`Time`) определенной операции (`Operation`), ее продолжительности (`Duration`) или времени окончания (`Time + Duration`), имя программного модуля (`Module`), где выполнялась эта операция, а также информацию о запросе (`QueryId`, `SubQueryId`, `RelationId`, если операция связана с обработкой запроса, иначе эти поля заполняются нулями) и информацию о сетевых адресах отправки и назначения (`From` и `To`, если операция не связана передачей по сети, иначе эти поля остаются пустым).

Чтобы автоматизировать заполнение всех полей записи был разработан помощник измерения времени операций. Он инициализируется во время запуска программного модуля: устанавливает имя модуля, выполняет синхронизацию времени с модулем MGM. Предоставляет два метода: метод генерация объекта состояния и метод завершения измерения. Объект состояния представляет собой запись журнала мониторинга с запущенным таймером. Для его создания в метод генерации передается как минимум один параметр – `Operation`, в общем случае в него может быть передано 6 параметров: `Operation`, `QueryId`, `SubQueryId`, `RelationId`, `From`

и `To`. Если какой-то из параметров отсутствует, то вместо него используется значение по умолчанию (пустая строка или нули).

Каждая записанная в журнал измерений запись обязательно содержит в себе идентификатор операции (поле `Operation`):

- `WaitStart` – время ожидания запуска запроса (время нахождения в очереди).
- `WaitJoin` – время ожидания начала *join* (от попадания запроса в очередь до старта *join*).
- `WaitSort` – время ожидания начала *sort* (от попадания запроса в очередь до старта *sort*).
- `ProcessingSelect` – время выполнения подзапроса *select-project*.
- `ProcessingJoin` – время выполнения подзапроса *join*.
- `ProcessingSort` – время выполнения подзапроса *sort*.
- `FileSave` – время подготовки данных к загрузке в БД.
- `LoadData` – время загрузки данных в БД.
- `Indexing` – время индексации отношения.
- `DataTransfer` – время передачи данных.
- `DeleteData` – время удаления временных отношений из БД.
- `WorkDuration` – общее время работы.
- `Pause` – время остановки/ожидания узла.

Где, `WaitStart`, `WaitJoin`, `WaitSort`, `ProcessingSelect`, `ProcessingJoin`, `ProcessingSort` – этапы выполнения запроса, `FileSave`, `LoadData`, `Indexing` – подготовка выполнению запроса, `DataTransfer` – сетевые передачи, `WorkDuration`, `Pause` – служебная информация, `DeleteData` – очистка.

Хранение собранных данных. Запись в журнал ведется асинхронно библиотекой журналирования NLog в специальной конфигурации (листинг Б.3), которая позволяет выполнять запись параметров мониторинга во встраиваемую СУБД SQLite [7]. Выбор именно этой СУБД обусловлен несколькими факторами. Во-первых, она не требует какой-либо специфической настройки или специального адми-

[7] Hipp R. About SQLite //SQLite. 2018. URL: https://www.sqlite.org/about.html (дата обращения: 29.07.2018).

нистрирования, не требует запуска себя как службы, а вместо этого управляется вызовами из хост-программы. Во-вторых, она хранит всю БД и информацию о ней в одном файле, что упрощает дальнейшую обработку данных и их переносимость между вычислительными системами. В-третьих, она обладает очень высокой производительностью на запись (в несколько раз быстрее MySQL и PostgreSQL).

Каждый модуль отчитывается модулю MGM о загрузке CPU и RAM с интервалом в 10 секунд и записывает эти значения в отдельный файл БД SQLite с именем `performance.db`, формируя таким образом локальный журнал производительности.

Журнал измерений (`times.db`), как и журнал производительности (`performance.db`) ведется на каждом узле отдельно в своей локальной БД. Это исключает мониторинг системы в реальном времени, но снимает нагрузку с сети, т.к. не требует передачи множества сообщений (до нескольких сотен в секунду) в MGM. Консолидация данных мониторинга на узле MGM запускается после окончания работы или по команде передачи файла.

Обработка данных мониторинга. В результате консолидации на MGM собирается множество файлов (количество узлов в составе Clusterix-N x2). Обрабатывать каждый файл в отдельности неудобно. Поэтому был разработан инструмент обработки данных мониторинга `LogProcessingTool`.

Задачи, выполняемые `LogProcessingTool`, следующие:

- Консолидация всех файлов журнала в один файл.
- Заполнение пропущенных данных (в модулях IO, JOIN, SORT нет информации о идентификаторе запроса, но есть информация об идентификаторе подзапроса и/или отношения, по которой отыскивается и заполняется недостающая информация).
- Составление отчета по полученным данным.
- Визуализация работы системы.

Задачи консолидации и заполнения пропущенных данных совмещены и выполняются за один проход. Сначала считывается информация о выполненных запросах: `QueryId`, `SubQueryId` и `RelationId`. Поочередно открываются все файлы БД журналов измерений, откуда считываются все записи с добавлением собранной ранее информации о запросах и записываются в общий файл журнала. Консолидированный файл журнала представляет собой БД SQLite со схемой, показанной на рис. А.1, где:

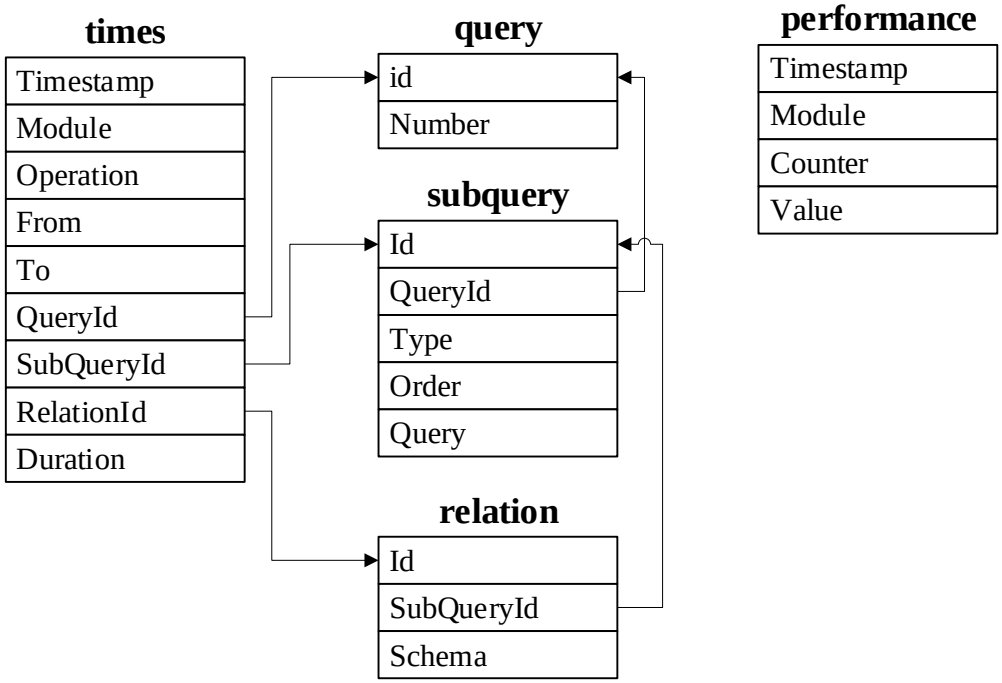


Рис. А.1. Схема консолидированной БД журнала измерений

- `query` – таблица с информацией о запросах, содержащая поля:
  - `Id` – идентификатор запроса;
  - `Number` – номер запроса;
- `subquery` – таблица с информацией о подзапросах, содержащая поля:
  - `Id` – идентификатор подзапроса;
  - `QueryId` – ссылка на родительских запрос;
  - `Type` – тип запроса (*select-project*, *join*, *sort*);
  - `Order` – порядковый номер выполнения в рамках запроса;
  - `Query` – выполненный запрос;
- `relation` – таблица с информацией о отношениях, содержащая поля:
  - `Id` – идентификатор отношения;
  - `SubQueryId` – ссылка на родительский подзапрос;
  - `Schema` – схема отношения;
- `times` – таблица журнала измерений, связанная с информацией о запросах;
- `performance` – таблица журнала измерений счетчиков производительности (CPU, RAM) с полями:
  - `Timestamp` – отметка времени.
  - `Module` – название модуля, с которого было получено значение.

- `Counter` – имя счетчика производительности (CPU или RAM).
- `Value` – значение счетчика производительности.

Все дальнейшие операции выполняются над подготовленным консолидированным файлом журнала измерений.

Отчет по работе (рис. А.2) включает в себя несколько разделов:

1. Количество выполненных запросов с группировкой по номеру запроса.
2. Суммарное время работы по каждой операции для каждого номера запроса.
3. Среднее временя работы по каждой операции для каждого номера запроса.

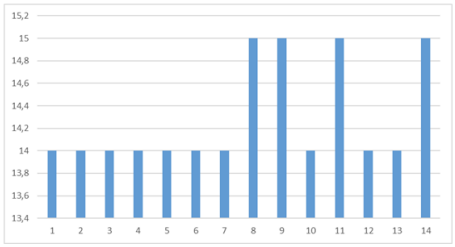
Суммарное время работы по каждой операции для каждого узла в системе. Отчет по работе помогает определить наиболее медленные операции, которые «тормозят» работу системы. Полученная информация может быть использована для определения наиболее эффективной конфигурации Clusterix-N для обработки потока запросов. Так, из рис. А.2 видно, что большая часть времени уходит на загрузку данных и 3 узла *join* оказываются перегруженными, в то время как узлы IO загружены в 3 раза меньше.

Визуализация работы (рис. А.3) строится на основании записей журнала измерений. Записи группируются по узлам и операциям, затем каждая запись отображается на диаграмме в виде прямоугольника заданного цвета. Цвет каждой операции заранее определен. Высота прямоугольника по умолчанию равна 5 пикселям, длина считывается по формуле: `Duration`/`PPMS`, где `Duration` – длительность операции, а `PPMS` – количество миллисекунд в одном пикселе итогового изображения. Если длина прямоугольника оказалась менее одного пикселя, то она устанавливается равной одному пикселю. Начальная точка прямоугольника рассчитывается по формуле: (`Time` – `StartTime`)/`PPMS`, где `Time` – отметка времени начала операции, `StartTime` – отметка времени самой первой операции. Отсчет времени в визуализации начинается с запуска запросов в момент времени 00:00:00.

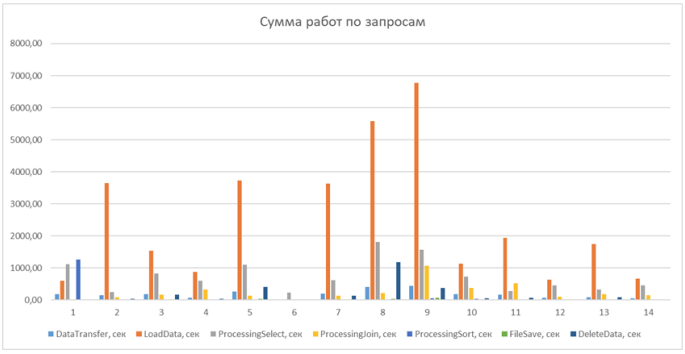
На рис. А.3 слева отображаются названия узлов и названия операций. Временная шкала представляет время в формате чч:мм:сс (часы:минуты:секунды). Отображение нескольких прямоугольников в несколько строк связано с тем, что некоторые операции выполняются параллельно. Соответственно, если операции накладываются друг на друга, то более поздняя их них рисуется ниже со смещением на 5 пикселей (высота прямоугольника операции).

Время выполнения 1413,18сек

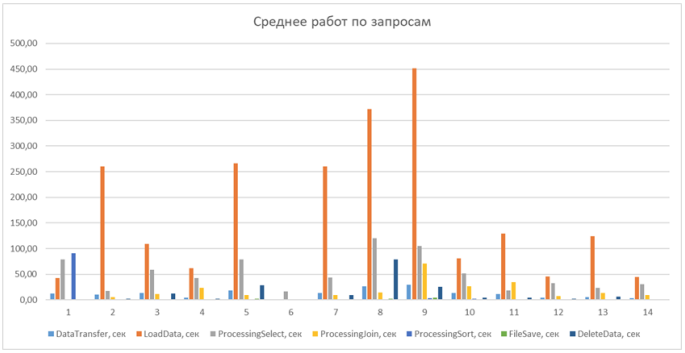
№ запроса	Количество
1	14
2	14
3	14
4	14
5	14
6	14
7	14
8	15
9	15
10	14
11	15
12	14
13	14
14	15



Сумма работ по запросам							
№ запроса	DataTransfer, сек	LoadData, сек	ProcessingSelect, сек	ProcessingJoin, сек	ProcessingSort, сек	FileSave, сек	DeleteData, сек
1	176,73	592,26	1111,13		1267,41	10,96	0,94
2	140,58	3640,09	249,82	81,01	0,20	4,21	35,72
3	185,71	1531,37	818,49	162,13	4,65	19,70	169,96
4	65,26	869,24	594,27	326,47	7,18	3,19	33,77
5	258,02	3725,27	1099,23	131,80	1,17	33,62	398,66
6	12,15	5,62	228,90		2,19	0,14	0,06
7	189,66	3637,33	608,13	138,88	1,68	13,96	138,05
8	398,29	5583,52	1805,75	210,31	0,28	37,87	1176,29
9	441,33	6777,23	1572,33	1069,05	49,49	63,92	380,79
10	184,96	1131,92	726,00	370,61	31,06	11,45	57,83
11	168,15	1937,13	279,69	526,34	4,35	5,35	71,09
12	63,23	633,97	451,11	100,61	3,56	4,31	27,00
13	76,28	1738,39	331,04	187,05	18,64	5,31	88,70
14	45,26	669,95	454,54	142,23	3,85	0,48	4,32



Среднее работ по запросам							
№ запроса	DataTransfer, сек	LoadData, сек	ProcessingSelect, сек	ProcessingJoin, сек	ProcessingSort, сек	FileSave, сек	DeleteData, сек
1	12,62	42,30	79,37		90,53	0,78	0,07
2	10,04	260,01	17,84	5,79	0,01	0,30	2,55
3	13,26	109,38	58,46	11,58	0,33	1,41	12,14
4	4,66	62,09	42,45	23,32	0,51	0,23	2,41
5	18,43	266,09	78,52	9,41	0,08	2,40	28,48
6	0,87	0,40	16,35		0,16	0,01	0,00
7	13,55	259,81	43,44	9,92	0,12	1,00	9,86
8	26,55	372,23	120,38	14,02	0,02	2,52	78,42
9	29,42	451,82	104,82	71,27	3,30	4,26	25,39
10	13,21	80,85	51,86	26,47	2,22	0,82	4,13
11	11,21	129,14	18,65	35,09	0,29	0,36	4,74
12	4,52	45,28	32,22	7,19	0,25	0,31	1,93
13	5,45	124,17	23,65	13,36	1,33	0,38	6,34
14	3,02	44,66	30,30	9,48	0,26	0,03	0,29



Работы узлов по стадиям							
Узел	DataTransfer, сек	LoadData, сек	ProcessingSelect, сек	ProcessingJoin, сек	ProcessingSort, сек	FileSave, сек	DeleteData, сек
IO_1	701,83		3848,74				
IO_2	781,77		3966,33				
JOIN_1	4,64	11374,80	524,19	1031,64		63,36	805,32
JOIN_2	31,80	11682,84	1039,84	1129,21		97,61	993,06
JOIN_3	2,44	8500,55	907,06	1285,64		37,96	775,60
MGM_1	881,55						
SORT_1	1,59	915,11	44,26		1395,70	15,55	9,18

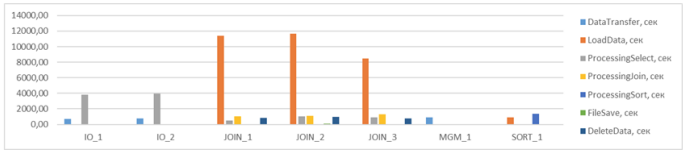


Рис. А.2. Отчет о работе системы

Рис. А.3. Пример визуализации эксперимента с $V_{БД}$ = 5 GB

Управление `LogProcessingTool` (указание действий и параметров) производится через интерфейс командной строки.

По журналу производительности (`performance`) можно определить объем доступной оперативной памяти и загрузку CPU при выполнении различных операций. Надо лишь наложить временные шкалы журнала измерений и журнала производительности. Делать это удобнее всего визуально. Поэтому в инструмент обработки данных мониторинга `LogProcessingTool` имеется возможность визуализации загрузки CPU (рис. А.4) и RAM (рис. А.5). Загрузка CPU визуализируется в про-

Рис. А.4. Визуализация загрузки CPU во время обработки запроса №9 при $V_{БД}$=70GB

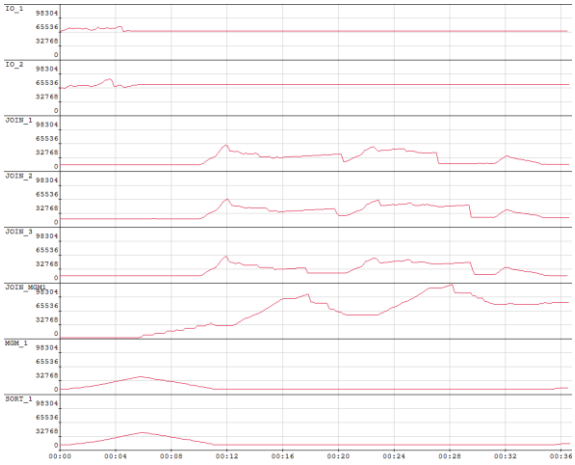


Рис. А.5. Визуализация загрузки RAM во время обработки запроса №9 при $V_{БД}$=70GB

центах, т.е. 100% соответствует полной загрузке всех процессорных ядер в узле. На визуализации RAM показан занятый объем оперативной памяти в MB. Для его правильного построения необходимо передать `LogProcessingTool` объем RAM узлов через параметр командной строки `--visualize -vram 131072`, где `-vram 131072` указывает на объем в 128 GB.

А.3. Модуль сетевого взаимодействия

Взаимодействие между компонентами системы выполняется через сеть. Поэтому модуль сетевого взаимодействия является одним из самых главных в Clusterix-N. Его основная задача: передавать и получать пакеты данных, производя их преобразование в поток байт во время передачи (*сериализацию*) и обратное преобразование потока байт в пакет данных при приеме (*десериализацию*). В основе модуля лежит Protocol Buffers [8] – протокол сериализации структурированных данных, предложенный Google как эффективная бинарная альтернатива текстовому формату XML. Он позволяет создавать пакеты на основе классов данных и напрямую работать с потоковой сетевой передачей, что существенно облегчает работу с сетью.

Всего в системе реализовано 19 пакетов для передачи данных, информации о запросах, их статусах и прочей служебной информации:

1. `CommandPacket` – пакет управления узлом, позволяет приостановить или возобновить его работу.
2. `DropQueryPacket` – пакет уничтожения всей информации связанной с переданными идентификаторами запроса, подзапроса и отношения.
3. `GetFileRequestPacket` – пакет запроса произвольного файла с узла, применяется для сбора данных подсистемы мониторинга со всех узлов на управляющем узле.
4. `GetFileResponcePacket` – пакет ответа на запрос файла.
5. `InfoRequestPacket` – пакет запроса сведений о узле (количество процессорных ядер и объем доступной оперативной памяти).
6. `InfoResponcePacket` – ответ на запрос сведений о узле.

[8] Protocol Buffers //Google Developers. 2018. URL: https://developers.google.com/protocol-buffers/ (дата обращения: 23.07.2018).

7. `IntegratedJoinCompletePacket` – пакет с сообщением о завершении выполнения интегрированного *join* над указанными отношениями и с указанием идентификатора нового отношения.

8. `IntegratedJoinStartPacket` – пакет запуска интегрированного *join* с указанием исходных и результирующего отношений.

9. `JoinCompletePacket` – пакет с сообщением о завершении выполнения *join* над двумя указанными отношениями и с указанием идентификатора нового отношения.

10. `JoinStartPacket` – пакет запуска *join* с двух указанием исходных и результирующего отношений.

11. `QueryPacket` – пакет запуска *select*, содержащий один из запросов *select-project* или запрос передачи результата в BUF MGM.

12. `RelationDataPacket` – пакет передачи данных для указанного отношения.

13. `RelationPreparePacket` – пакет подготовки отношения к загрузке данных, содержащий информацию о схеме отношения, его имени и индексах.

14. `RelationPreparedPacket` – пакет подтверждения готовности отношения к загрузке данных.

15. `SelectResult` – пакет с результатом обработки запроса.

16. `SortCompletePacket` – пакет с сообщением о завершении операции sort.

17. `SortStartPacket` – пакет запуска операции *sort*, содержащий необходимый запрос.

18. `StatusPacket` – пакет с информацией о текущем состоянии узла: нагрузка на процессор и объем свободной оперативной памяти. Отправляется со всех узлов в MGM с 10 секундным интервалом.

19. `TimeAdjustPacket` – пакет синхронизации времени между узлами. Отравляется один раз при подключении нового узла к MGM.

Коммуникация между программными модулями организуется в соответствии с рис. А.6. На нем можно выделить 9 универсальных пакета, используемых во всех программных модулях: `TimeAdjustPacket`, `StatusPacket`, `CommandPacket`, `GetFileRequestPacket`, `GetFileResponcePacket`, `InfoRequestPacket`, `InfoResponcePacket`, `QueryPacket` и `SelectResult`. Обработчики этих пакетов реализованы все всех модулях одинаково. Остальные же пакеты обрабатываются в программных моду-

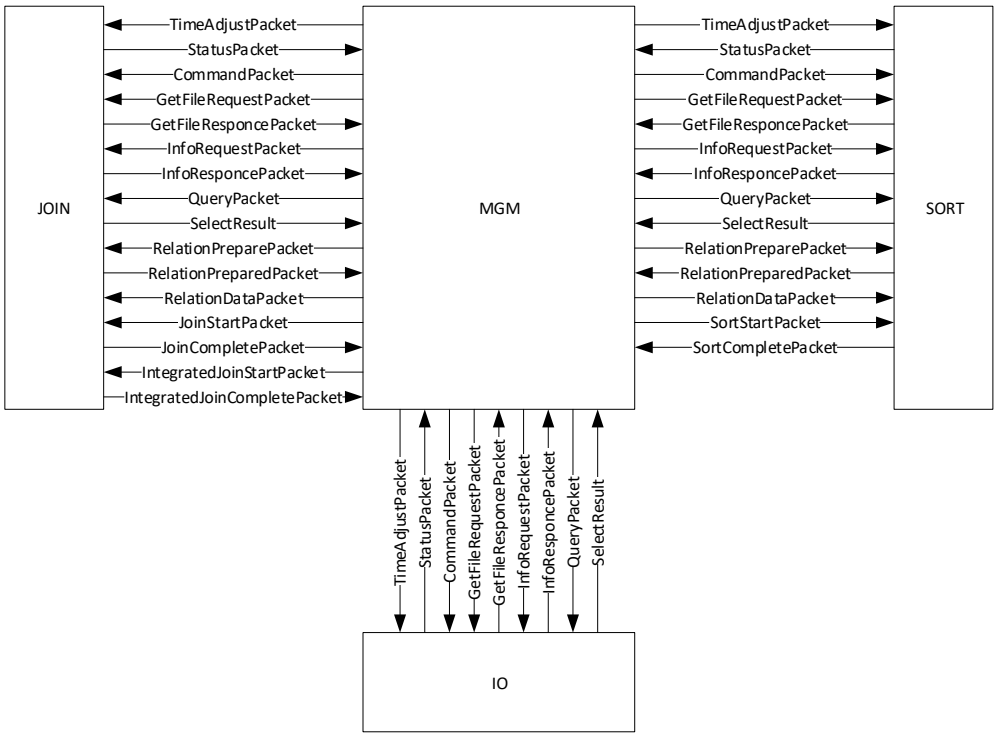


Рис. А.6. Схема обмена пакетами между программными модулями

лях индивидуально. Такой подход сделал необходимым разработку универсального клиент-сервера, который в зависимости от типа пакета вызывал бы требуемый обработчик или записывал сообщение о ошибке в случае, если обработчик не был найден.

Обработчик пакета представляет собой метод с единственным параметром – пакетом. Чтобы обработчик мог использоваться в работе его необходимо зарегистрировать в модуле сетевого взаимодействия следующим образом: `_server.SubscribeToPacket<SelectResult>(SelectResultHandler)`, где `_server` – модуль сетевого взаимодействия, `SelectResult` – тип пакета, который будет передан в обработчик, `SelectResultHandler` – название метода обработчика. Во время регистрации обработчика модуль сетевого взаимодействия добавляет его в коллекцию и вызывает в случае совпадения типа пакета.

Аналогичный механизм организован для передачи пакетов в сеть, но задача у него иная. Если при приеме пакета необходимо знать его тип для передачи соответствующему обработчику, то для передачи такой функционал не нужен. Но добавление обработчиков отправляемых пакетов позволяет организовать мониторинг сетевых передач без серьезного вмешательства в исходный код других модулей. Кроме того, такие обработчики существенно упрощают отладку сетевого взаимодействия. Важно отметить, что если модуль сетевого взаимодействия получает на

отправку или прием незарегистрированный пакет, то он его отбрасывает с занесением ошибки в журнал. Это тоже упрощает отладку.

Асинхронная природа модуля сетевого взаимодействия позволяет основной программе не ждать окончания сетевых передач, а продолжать работать и добавлять задания для передачи по сети. Очередь задач организуется на основе динамического массива. Блокировки очереди происходят только во время добавления и извлечения задач. В случае же, когда какой-либо программе необходимо удостовериться в передачи пакета, в модуле реализован синхронный режим работы.

Особого внимания заслуживают пакеты `RelationDataPacket` и `SelectResult`, т.к. они сопряжены с передачей большого объема данных. Выполнять передачу по кортежам, как это было сделано в Clusterix, нет никакого смысла, поскольку передача пакетов малого объема неблагоприятно влияет на производительность сети. Для достижения передачи по сети на максимальной скорости следует передавать крупные пакеты и использовать потоковую передачу. Оба условия можно удовлетворить передачей отношений целиком, но это потребует большого объема оперативной памяти для буфера данных. Поэтому в Clusterix-N используется передача данных блоками – массивами байт, в которые записаны множество строк отношения. Размер блока зависит от количества входящих в него строк и их длин. Так, размер блока, содержащего 1000 строк со средней длиной строки равной 1 KB, составит 1000·1KB ≈1 MB.

Чтобы избежать многократного разбиения отношений на блоки в процессе передачи, блоки формируются еще на этапе получения результата из инструментальной СУБД в драйвере СУБД. В процессе работы, полученные блоки, не подвергаются каким-либо преобразованиям и передаются в другие программные модули «как есть».

А.4. Драйвер СУБД

Работа с инструментальной СУБД может осуществляться несколькими способами: использовать ее как внешний сервис со своим интерфейсом, использовать исходный код СУБД непосредственно для создания новой СУБД, расширение функционала существующей СУБД. Первый вариант позволяет не ограничиваться

в выборе всего одной СУБД и предоставляет возможность использования различных СУБД на разных этапах обработки, но цена за универсальность – снижение производительности. Второй вариант требует досконального знания исходного кода выбранной инструментальной СУБД и огромных усилий по его переработке, но в результате можно получить оригинальную высокопроизводительную СУБД. Третий вариант, так же, как и второй, требует хорошего знания исходного кода СУБД, ограничивается в работе только с одной СУБД и в большей мере требует соответствия архитектуры инструментальной СУБД поставленным задачам. У этого варианта можно отметить существенный плюс: выполненные наработки могут быть приняты сообществом или организацией, которая осуществляет поддержку и выпуск СУБД. Тогда не возникает вопроса о внедрении разработанной системы в информационные системы.

Два последних варианта позволяют добиться высокой производительности с ограничением в изначальной архитектуре СУБД или с чрезвычайными трудозатратами по ее модификации. Так, в работе [57] выполнена интеграция параллелизма в открытую СУБД PostgreSQL. Поскольку PostgreSQL обладает очень зрелой и гибкой архитектурой, внедрение новых операторов не создало непреодолимых трудностей перед автором.

В настоящее время существует множество различных СУБД, которые ориентированы на ту или иную область. В таких условиях желательно иметь универсальный подход в работе с инструментальной СУБД, чтобы иметь возможность менять ее при необходимости и не зависеть от конкретной СУБД. Поэтому в Clusterix-N реализуется подход с использованием СУБД как внешнего сервиса со своим интерфейсом. При этом реализация такого подхода не отвергает двух оставшихся (создание собственного движка СУБД или тесной интеграции с существующей СУБД), т.к. их можно реализовать позже и использовать в уже готовой системе с реализацией нового драйвера СУБД.

Чтобы иметь возможность подключать различные СУБД к Clusterix-N, реализован интерфейс драйвера СУБД[9]. Интерфейс реализуется в динамически

[9] Интерфейс драйвера СУБД ClusterixN.Common.Interfaces.IDatabase URL: https://bitbucket.org/rozh/clusterixn/src/a1de9e102598f7a33cd096cf421229498616035c/ClusterixN.Common/Interfaces/IDatabase.cs?at=master&fileviewer=file-view-default

загружаемой библиотеке, которая занимается преобразованием команд Clusterix-N в SQL-запросы/команды конкретной СУБД.

В Clusterix-N в качестве инструментальной СУБД используется MySQL. Она была выбрана в силу наличия движка MEMORY, который позволяет выполнять все операции над БД в оперативной памяти без использования внешнего хранилища. Организация работы в оперативной памяти необходима для наиболее быстрого функционирования программных модулей JOIN и SORT, т.к. они производят загрузку данных и их индексирование. Выполнение этих операций с использованием внешнего хранилища привело бы к значительным задержкам.

Реализация драйвера только на .Net C# оказалась не эффективной: чтение данных из БД было очень медленным. Анализ работы драйвера показал, что реализация MySQL Connector/Net производит чтении результирующих данных из соединения побайтно и пытается сразу приводить их к типам данных, поддерживаемых .Net. В то же время реализация MySQL Connector для C только копирует результат из соединения в буфер без каких-либо преобразований. Перенос операций чтения из БД на C позволил значительно ускорить чтение данных из БД.

Чтение данных из БД организуется следующим образом. Исходный запрос модифицируется с целью реализации чтения данных по блокам. К нему добавляется строка вида: «*LIMIT P,N*», где *N* – размер блока данных, *P* – номер очередного блока умноженный на *N*. Модифицированный запрос передается в MySQL. Результат выполнения запроса, полученный из MySQL, содержит не более *N* строк. Полученный блок передается с помощью события чтения очередного блока (или как результат работы метода `Select`) в обработчик приложения.

Все методы интерфейса драйвера СУБД генерируют и выполняют соответствующие их назначению SQL запросы. Особого внимания заслуживают методы `QueryIntoRelation` и `LoadFile`. `QueryIntoRelation` выполняет модификацию исходного запроса путем подстановки параметров в строку "`INSERT INTO` `{relationName}` `{query}`", где `relationName` – имя отношения для вставки, `query` – исходный запрос. Затем модифицированный запрос исполняется, а в `relationName` оказывается результат запроса. Метод `LoadFile` выполняет загрузку

файлов в отношения БД, используя специальную инструкцию диалекта SQL для СУБД MySQL [10]: `LOAD DATA INFILE`.

В библиотеке драйвера, помимо интерфейса драйвера, должен быть реализован интерфейс сервиса СУБД (листинг А.1), включающий единственный метод, который должен возвращать объект драйвера СУБД, настроенный в соответствии с переданной строкой подключений. При этом, может быть возвращен новый экземпляр объекта драйвера (если `newInstance = true`). Это может быть полезно, если требуется создать несколько разных экземпляров драйвера. Параметр `connectionId` позволяет присвоить идентификатор экземпляру драйвера и получать его через метод `GetDatabase` с помощью сервиса.

Листинг А.1. Интерфейс сервиса СУБД

```
1.  namespace ClusterixN.Common.Interfaces
2.  {
3.    public interface IDatabaseService
4.    {
5.      IDatabase GetDatabase(string connectionString, string connectionId, bool
    newInstance = false);
6.    }
7.  }
```

При работе с MySQL все операции выполняются в новых соединениях. Поэтому два последних параметра из метода `GetDatabase` не используются. Но при работе, например, с PostgreSQL они оказываются полезны. PostgreSQL может хранить временные отношения только в установленном соединении. В случае разрыва соединения временные отношения уничтожаются. Создание нескольких экземпляров драйвера СУБД с присвоением им уникальных идентификаторов позволяет держать соединения открытыми ровно столько времени, сколько это необходимо. Поскольку PostgreSQL не поддерживает работу только в оперативной памяти и его производительность оказалась хуже MySQL (из-за использования внешнего хранилища), то его драйвер был удален из системы.

Реализованный драйвер для СУБД MySQL используется во всех программах. Выбор драйвера производится в конфигурационном файле [94].

[10] Oracle Corporation. LOAD DATA INFILE Syntax //MySQL Documentation. 2018. URL: https://dev.mysql.com/doc/refman/5.7/en/load-data.html (дата обращения: 24.07.2018).

ПРИЛОЖЕНИЕ Б. ЛИСТИНГИ

Листинг Б.1. Конфигурация MySQL

```
[mysqld]

user                        = mysql
default-storage-engine      = InnoDB
socket                      = mysql.sock
pid-file                    = mysql.pid
character-set-server        = utf8
collation-server            = utf8_general_ci
key-buffer-size             = 32M
max-allowed-packet          = 16M
max-connect-errors          = 1000000
datadir                     = data\

# CACHES AND LIMITS #
tmp-table-size              = 32M
max-heap-table-size         = 32M
query-cache-type            = 0
query-cache-size            = 0
max-connections             = 500
thread-cache-size           = 50
open-files-limit            = 65535
table-definition-cache      = 1024
table-open-cache            = 2048

# INNODB #
innodb-log-files-in-group       = 2
innodb-log-file-size            = 256M
innodb-flush-log-at-trx-commit  = 1
innodb-file-per-table           = 1
innodb-buffer-pool-size         = 16G

# LOGGING #
log-error                     = mysql-error.log
log-queries-not-using-indexes = 1
slow-query-log                = 1
slow-query-log-file           = mysql-slow.log

init-file = mysql_init.sql
```

Листинг Б.2. Файл инициализации mysql_init.sql

```
 USE tpch5;

 SELECT * FROM LINEITEM;
 SELECT * FROM PART;
 SELECT * FROM SUPPLIER;
 SELECT * FROM PARTSUPP;
 SELECT * FROM NATION;
 SELECT * FROM REGION;
 SELECT * FROM CUSTOMER;
 SELECT * FROM ORDERS;
```

Листинг Б.3. Конфигурацяи NLog для записи в SQLite

```
<target name="Database" xsi:type="Database" keepConnection="false"
    useTransactions="false"
    dbProvider="System.Data.SQLite.SQLiteConnection, System.Data.SQLite"
    connectionString="Data Source=timeLog.db;Version=3;"
```

```
5.       commandText="INSERT INTO times(Timestamp, Module, Operation, `From`,
     `To`, QueryId, SubQueryId, RelationId, Duration)
6.       VALUES(@Timestamp, @Module, @Operation, @From, @To, @QueryId,
     @SubQueryId, @RelationId, @Duration)">
7.    <parameter name="@Timestamp" layout="${event-properties:Time:format=yyyy-MM-
     dd HH\:mm\:ss.fff}"/>
8.    <parameter name="@Module" layout="${event-properties:Module}"/>
9.    <parameter name="@Operation" layout="${event-properties:Operation}"/>
10.   <parameter name="@From" layout="${event-properties:From}"/>
11.   <parameter name="@To" layout="${event-properties:To}"/>
12.   <parameter name="@QueryId" layout="${event-properties:QueryId}"/>
13.   <parameter name="@SubQueryId" layout="${event-properties:SubQueryId}"/>
14.   <parameter name="@RelationId" layout="${event-properties:RelationId}"/>
15.   <parameter name="@Duration" layout="${event-properties:Duration}"/>
16.  </target>
17.  <target name="PerfDatabase" xsi:type="Database" keepConnection="false"
18.       useTransactions="false"
19.       dbProvider="System.Data.SQLite.SQLiteConnection, System.Data.SQLite"
20.       connectionString="Data Source=performance.db;Version=3;"
21.       commandText="INSERT INTO performance(Timestamp, Module, Counter, `Value`)
22.       VALUES(@Timestamp, @Module, @Counter, @Value)">
23.   <parameter name="@Timestamp" layout="${event-properties:Time:format=yyyy-MM-
     dd HH\:mm\:ss.fff}"/>
24.   <parameter name="@Module" layout="${event-properties:Module}"/>
25.   <parameter name="@Counter" layout="${event-properties:Counter}"/>
26.   <parameter name="@Value" layout="${event-properties:Value}"/>
27.  </target>
```

Листинг Б.4. Пример вывода команды help в терминале MGM

```
1.   # help
2.   exit - Завершение работы терминала
3.   help - Вывод доступных команд
4.   startq - Запуск запроса
5.       -n - номер запроса
6.   startbench - Запуск 14 запросов на выполнение
7.       -c - количество повторений
8.   gettimelog - Запрос лога времени со всех узлов
9.   startstream - Запуск потока запросов на выполнение
10.      -c - количество запросов
11.      -l - длина очереди
12.  tpchseq - Запуск последовательности запросов для заданного числа перестановок
13.      -c - количество перестановок
14.      -l - длина очереди
15.  tpchstream - Запуск последовательности запросов для заданного числа
     перестановок
16.      -c - количество запросов
17.      -l - длина очереди
18.  pause - остановка IO
19.  resume – возобновление работы IO
```

Листинг Б.5. Базовый класс обработчика стадии

```
1. internal abstract class HandlerBase : IPipelineNode
2. {
3.   protected ICommunicator Server { get; }
4.   protected IQueryManager QueryManager { get; }
5.   protected NodesManager NodesManager { get; }
6.   protected PauseLogManager PauseLogManager { get; }
7.   protected QueryBufferManager QueryBufferManager { get; }
8.   protected QueueManager QueueManager { get; }
9.   protected readonly ILogger Logger;
10.
11.     protected HandlerBase(ICommunicator server, IQueryManager queryManager,
```

```
   NodesManager nodesManager,
12.       PauseLogManager pauseLogManager, QueryBufferManager queryBufferManager,
   QueueManager queueManager)
13.     {
14.       Server = server;
15.       QueryManager = queryManager;
16.       NodesManager = nodesManager;
17.       PauseLogManager = pauseLogManager;
18.       QueryBufferManager = queryBufferManager;
19.       QueueManager = queueManager;
20.       Logger = ServiceLocator.Instance.LogService.GetLogger("defaultLogger");
21.     }
22.
23.     public abstract void DoAction();
24.   }
```

Листинг Б.6. Отрывок из журнала времени выполнения запросов

```
1. [2017-08-11 16:58:18.8366] Завершен запрос 8147cd5d-dba7-4dfc-aec2-87285ce23793
   (12) за 46598.3227 мс
2.  Операция SELECT:
3.   id = bce45dd0-3772-432e-8423-6cd16e10a305
4.    8/11/2017 4:57:33 PM Wait -> SelectProcessing 2075.1746 мс
5.    8/11/2017 4:57:40 PM SelectProcessing -> TransferSelectResult 6949.5553 мс
6.    8/11/2017 4:57:43 PM TransferSelectResult -> SelectProcessed 2106.4613 мс
7.   id = 865fda1c-db06-459a-8a43-ba4f39b11d2a
8.    8/11/2017 4:57:34 PM Wait -> SelectProcessing 2207.8864 мс
9.    8/11/2017 4:57:43 PM SelectProcessing -> TransferSelectResult 9597.7256 мс
10.   8/11/2017 4:57:44 PM TransferSelectResult -> SelectProcessed 300.6252 мс
11. Операция JOIN:
12.  id = 7b2453e3-4de2-4d30-8028-cfb1bc304db6
13.   8/11/2017 4:57:52 PM Wait -> TransferToJoin 20350.5308 мс
14.   8/11/2017 4:58:02 PM TransferToJoin -> JoinProcessing 9806.9828 мс
15.   8/11/2017 4:58:09 PM JoinProcessing -> JoinProcessed 7888.4979 мс
16.   8/11/2017 4:58:09 PM JoinProcessed -> TransferJoinResult 8.1922 мс
17.   8/11/2017 4:58:14 PM TransferJoinResult -> JoinResultTransfered 4116.1946 мс
18. Операция SORT:
19.  id = ffa822d8-66bc-41cf-8df2-2f7949e55110
20.   8/11/2017 4:58:18 PM Wait -> TransferToSort 46150.3715 мс
21.   8/11/2017 4:58:18 PM TransferToSort -> TransferedToSort 115.159 мс
22.   8/11/2017 4:58:18 PM TransferedToSort -> ProcessingSort 0.6673 мс
23.   8/11/2017 4:58:18 PM ProcessingSort -> SortProcessed 558.503 мс
24.   8/11/2017 4:58:18 PM SortProcessed -> TransferSortResult 1.7732 мс
25.   8/11/2017 4:58:18 PM TransferSortResult -> SortResultTransfered 87.8549 мс
```

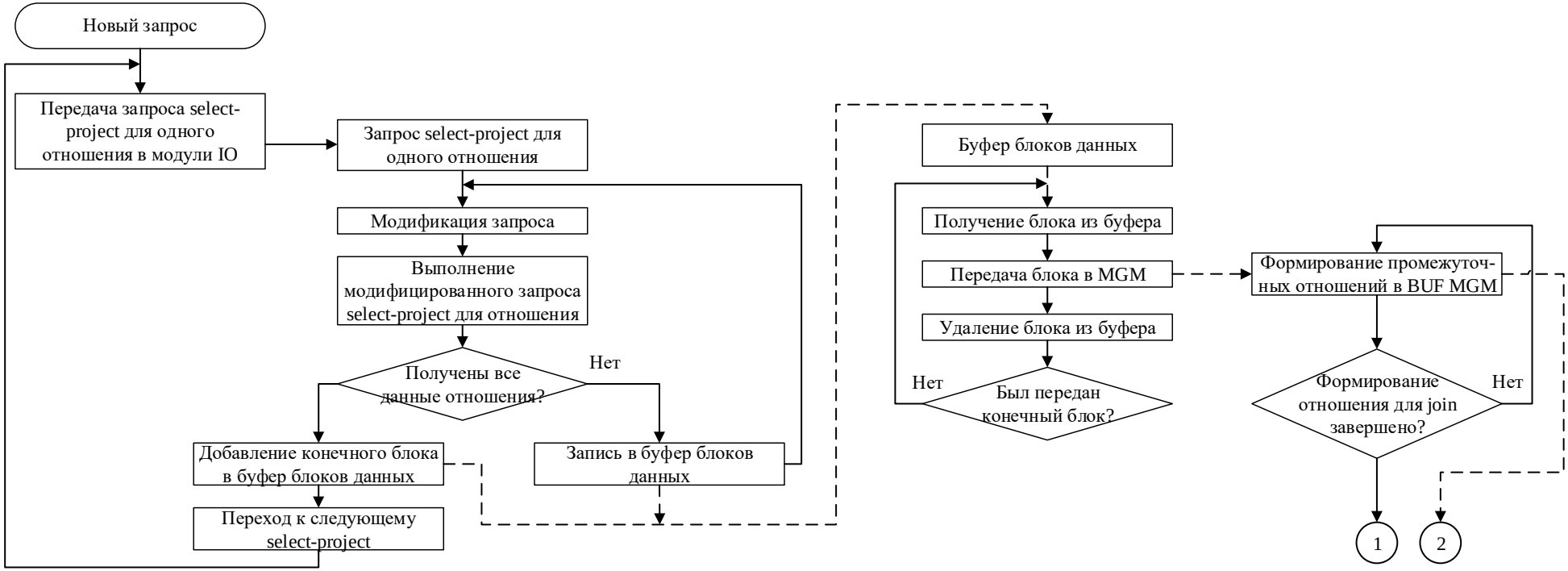
Листинг Б.7. Отрывок пакета команд для загрузки тестовых данных

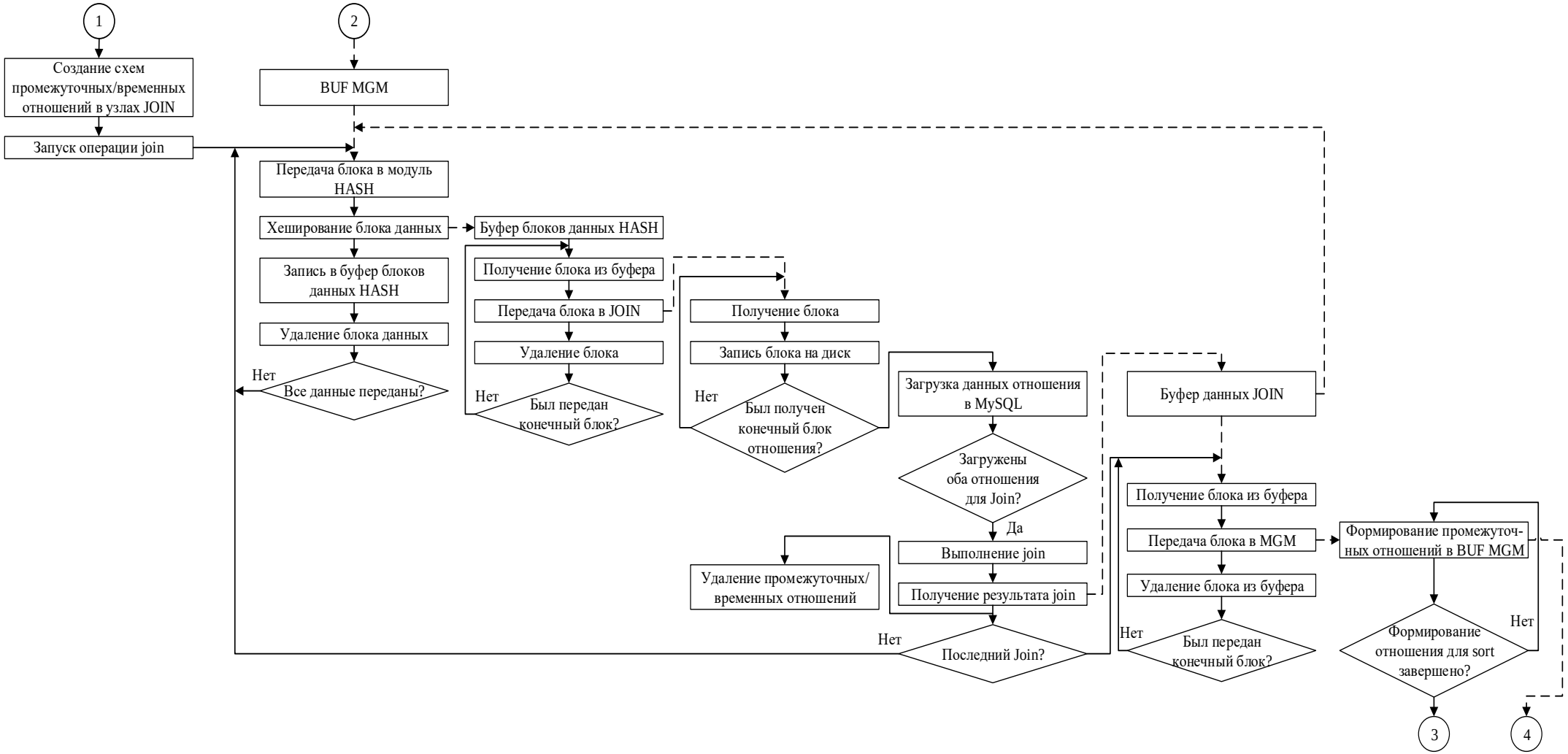
```
1.   -- создание новой БД с именем tpch5
2.   create database tpch5;
3.   -- использовать БД tpch5 (все дальнейшие действия будут применены к этой БД).
4.   use tpch5;
5.
6.   -- инструкции создания новых отношений с указанными именами и параметрами
     полей
7.   CREATE TABLE NATION ( N_NATIONKEY INTEGER NOT NULL,
8.                N_NAME    CHAR(25) NOT NULL,
9.                N_REGIONKEY INTEGER NOT NULL,
10.               N_COMMENT  VARCHAR(152));
11.
12.  CREATE TABLE REGION ( R_REGIONKEY INTEGER NOT NULL,
13.               R_NAME    CHAR(25) NOT NULL,
14.               R_COMMENT  VARCHAR(152));
15.  <... часть файла пропущена ...>
16.
```

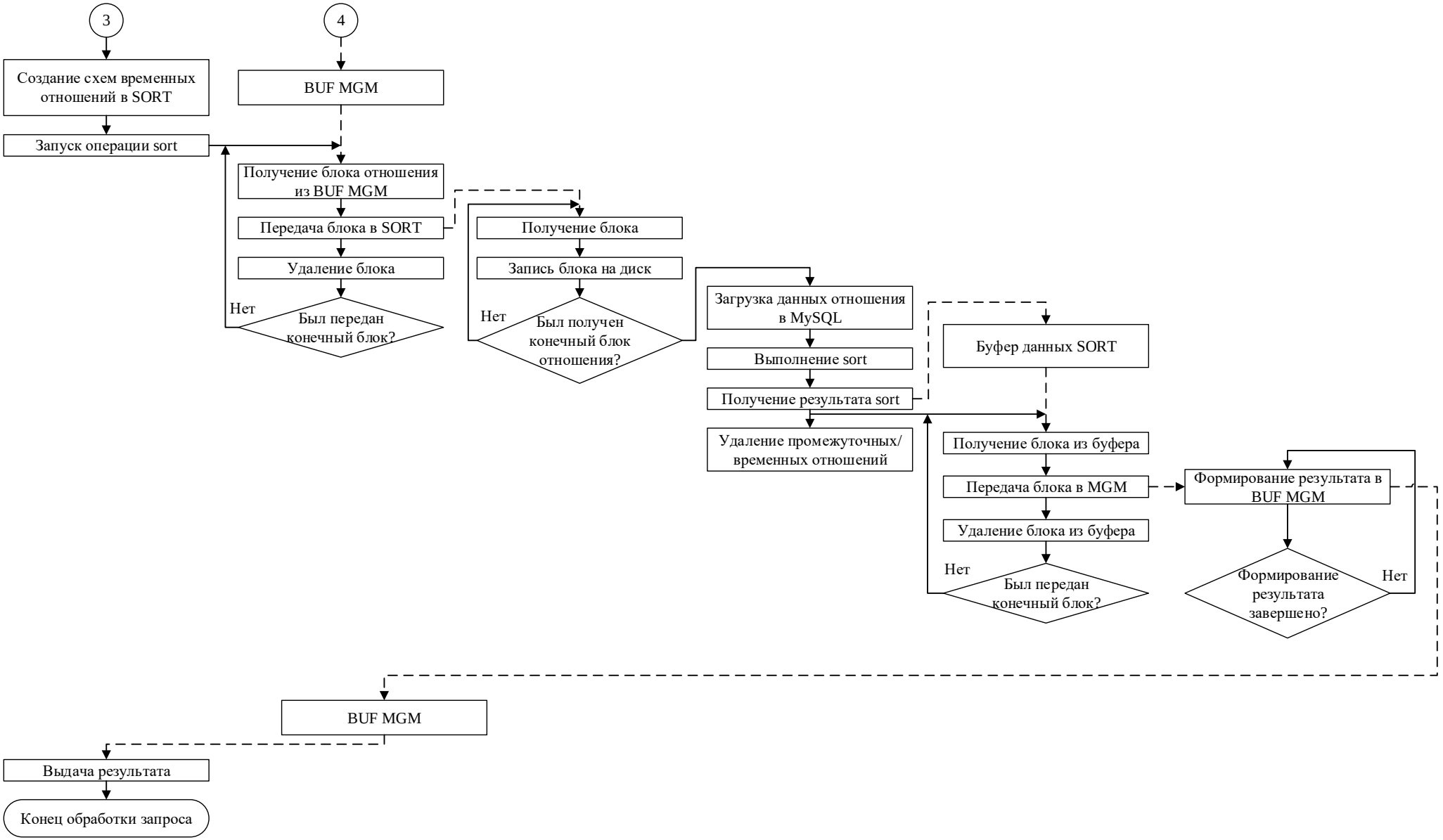
```
17. -- инструкция загрузки файла в определенное отношение с указанным форматом
18. load data infile "/mnt/serv/files/tmp/tpch_2_16_1/dbgen/3G/part.tbl" into
    table PART fields terminated by "|" lines terminated by "\n";
19. load data infile "/mnt/serv/files/tmp/tpch_2_16_1/dbgen/3G/partsupp.tbl" into
    table PARTSUPP fields terminated by "|" lines terminated by "\n";
20.
21. <... часть файла пропущена ...>
22.
23. -- добавление к отношениям первичных ключей.
24. ALTER TABLE REGION ADD PRIMARY KEY (R_REGIONKEY);
25. ALTER TABLE NATION ADD PRIMARY KEY (N_NATIONKEY);
26.
27. <... часть файла пропущена ...>
28.
29. -- добавление к отношениям индексов
30. ALTER TABLE `NATION` ADD INDEX ( `N_REGIONKEY` ) ;
31. ALTER TABLE `NATION` ADD INDEX ( `N_COMMENT` ) ;
32.
33. <... часть файла пропущена ...>
34.
35. -- добавление к отношениям внешних ключей (связей, связанных ключей).
36. ALTER TABLE `NATION` ADD FOREIGN KEY ( `N_REGIONKEY` ) REFERENCES `REGION` (
37. `R_REGIONKEY`
38. ) ON DELETE RESTRICT ON UPDATE RESTRICT ;
39.
40. <... часть файла пропущена ...>
```

ПРИЛОЖЕНИЕ В. ДИАГРАММЫ ФУНКЦИОНИРОВАНИЯ CLUSTERIX-N

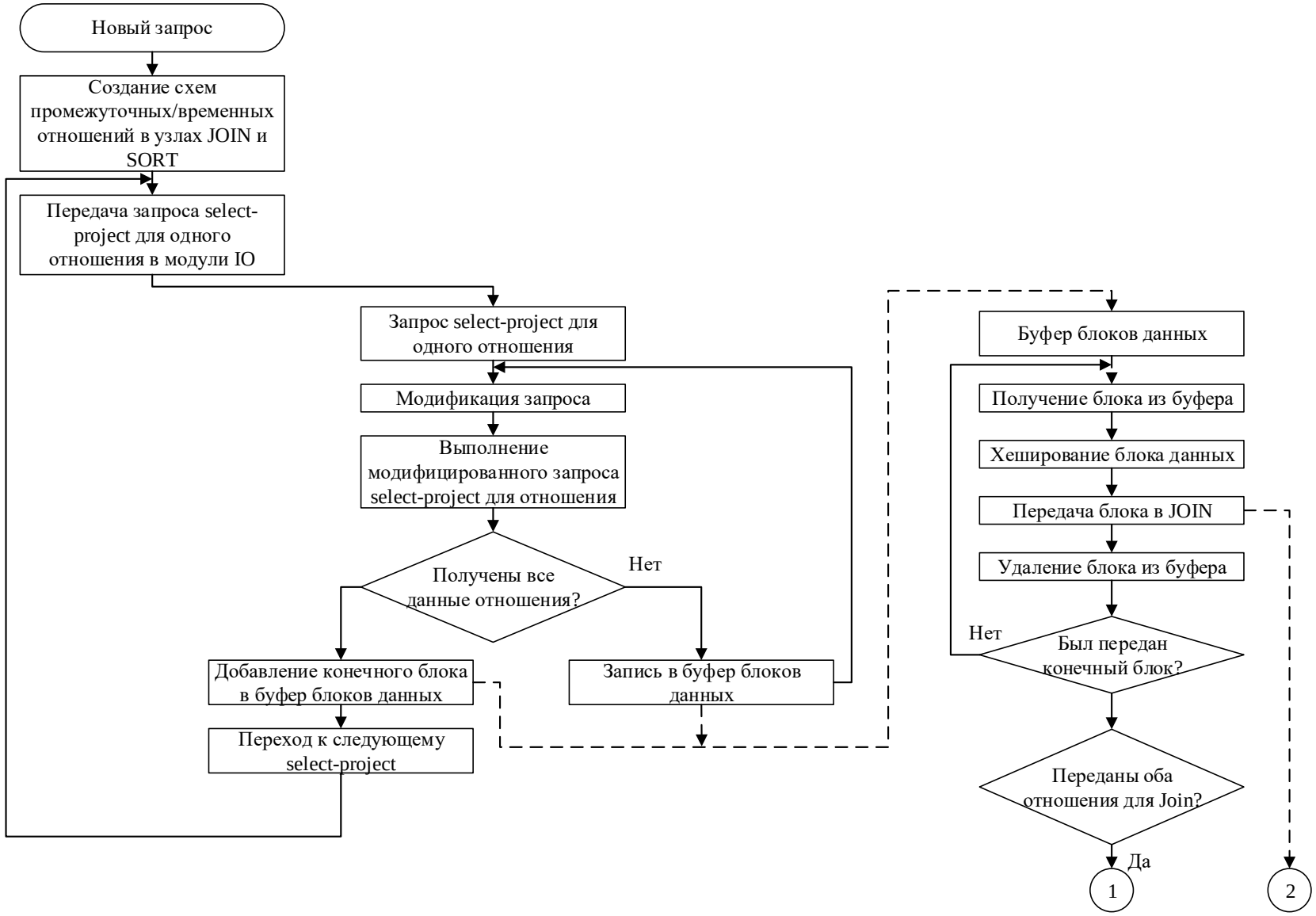
В.1. Итерация 1

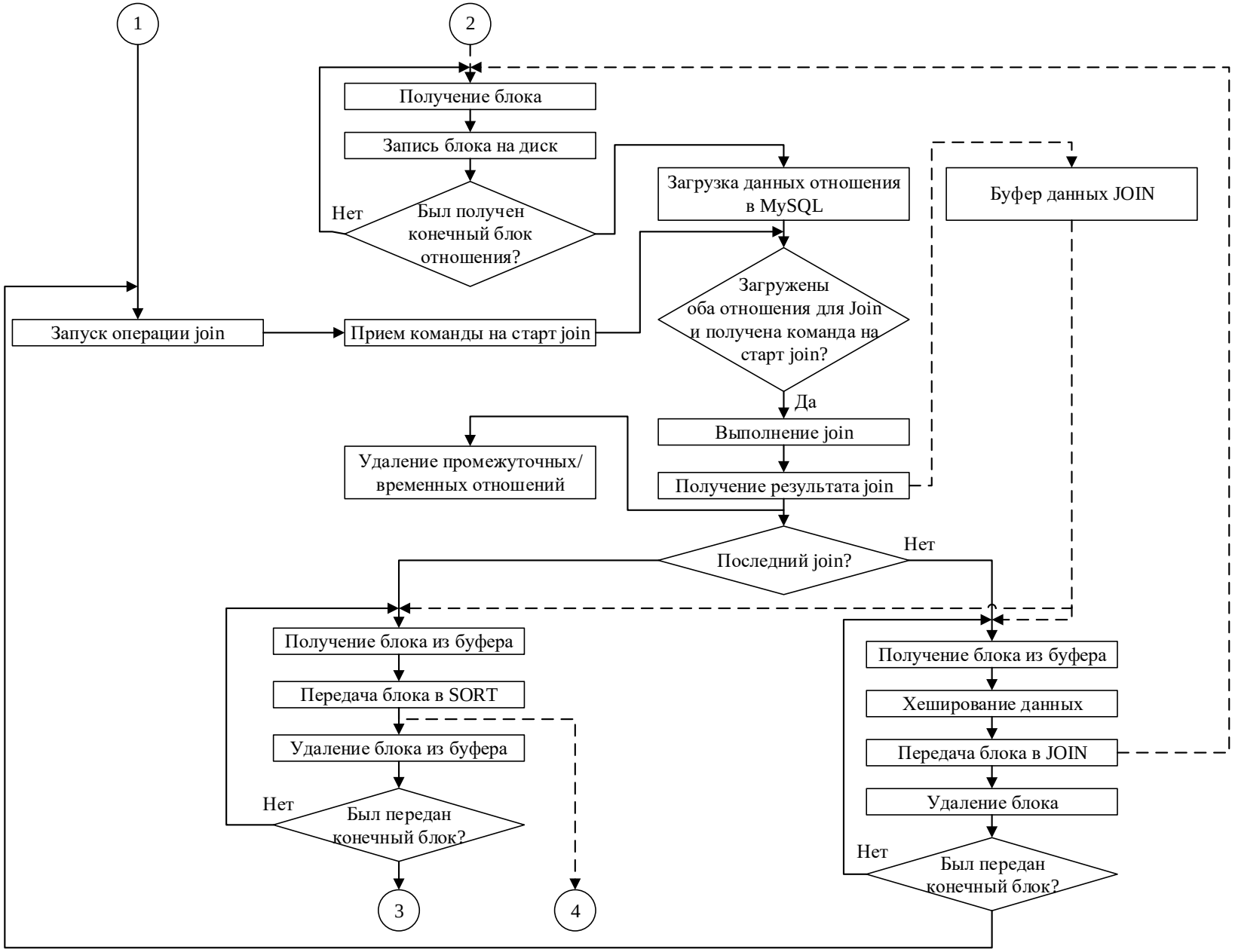


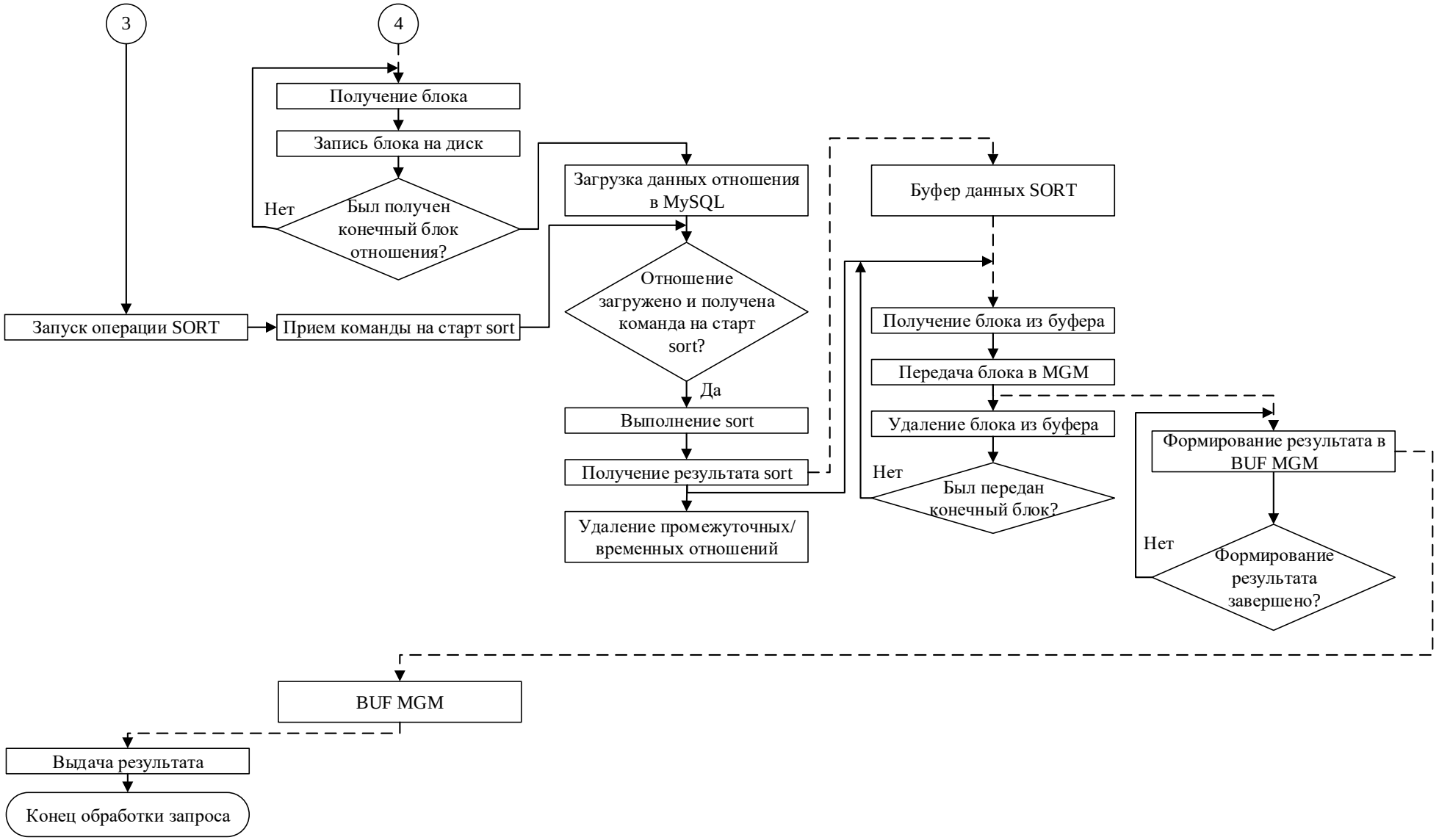

1
2
Создание схем промежуточных/временных отношений в узлах JOIN
Запуск операции join
BUF MGM
Передача блока в модуль HASH
Хеширование блока данных
Буфер блоков данных HASH
Запись в буфер блоков данных HASH
Удаление блока данных
Нет
Все данные переданы?
Получение блока из буфера
Передача блока в JOIN
Удаление блока
Нет
Был передан конечный блок?
Получение блока
Запись блока на диск
Нет
Был получен конечный блок отношения?
Загрузка данных отношения в MySQL
Загружены оба отношения для Join?
Да
Выполнение join
Получение результата join
Нет
Последний Join?
Удаление промежуточных/ временных отношений
Буфер данных JOIN
Получение блока из буфера
Передача блока в MGM
Удаление блока из буфера
Нет
Был передан конечный блок?
Формирование промежуточ-ных отношений в BUF MGM
Нет
Формирование отношения для sort завершено?
3
4


3
4
Создание схем временных отношений в SORT
Запуск операции sort
BUF MGM
Получение блока отношения из BUF MGM
Передача блока в SORT
Удаление блока
Нет
Был передан конечный блок?
Получение блока
Запись блока на диск
Нет
Был получен конечный блок отношения?
Загрузка данных отношения в MySQL
Выполнение sort
Получение результата sort
Удаление промежуточных/ временных отношений
Буфер данных SORT
Получение блока из буфера
Передача блока в MGM
Удаление блока из буфера
Нет
Был передан конечный блок?
Формирование результата в BUF MGM
Формирование результата завершено?
Нет
BUF MGM
Выдача результата
Конец обработки запроса

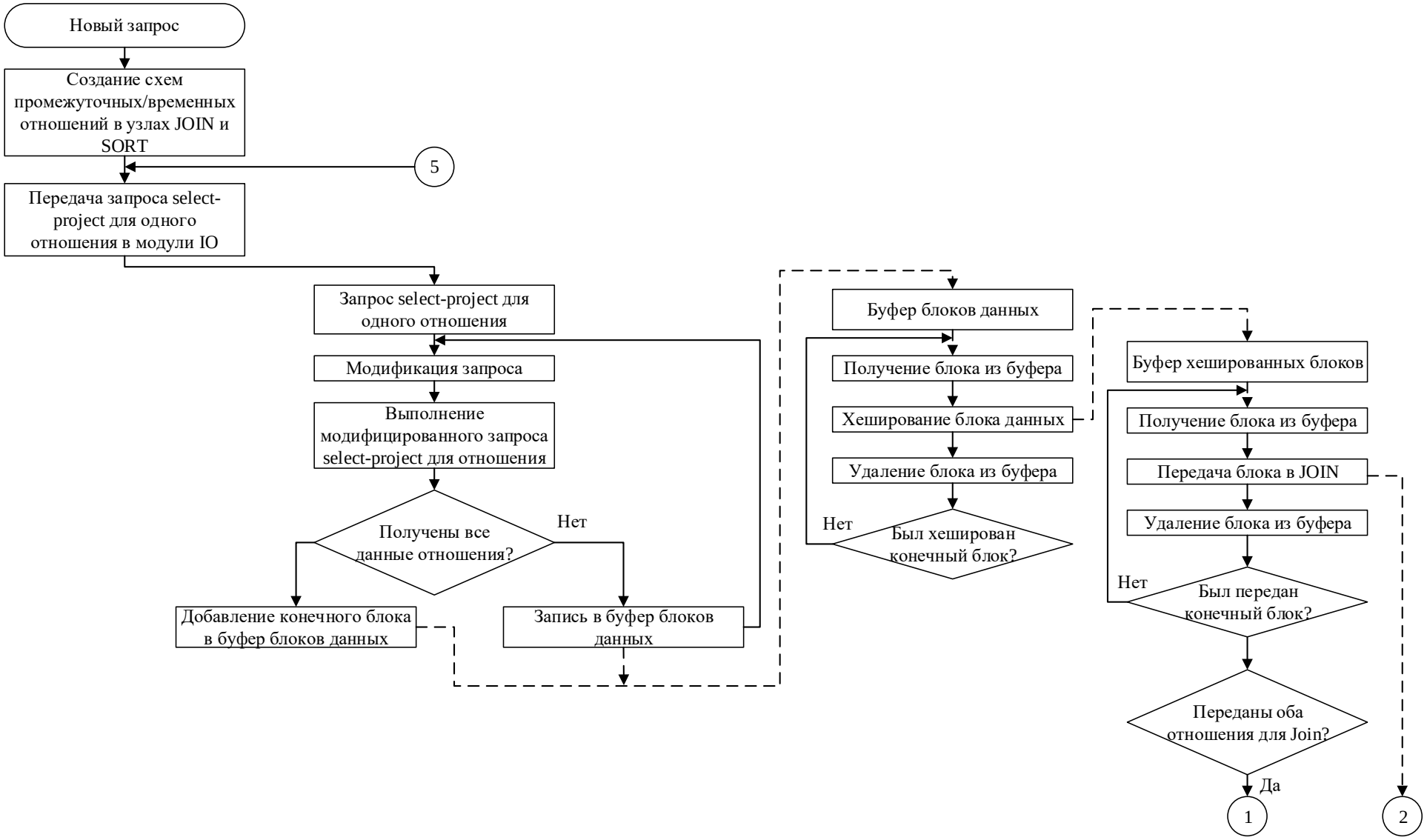
В.2. Итерация 2 и 3

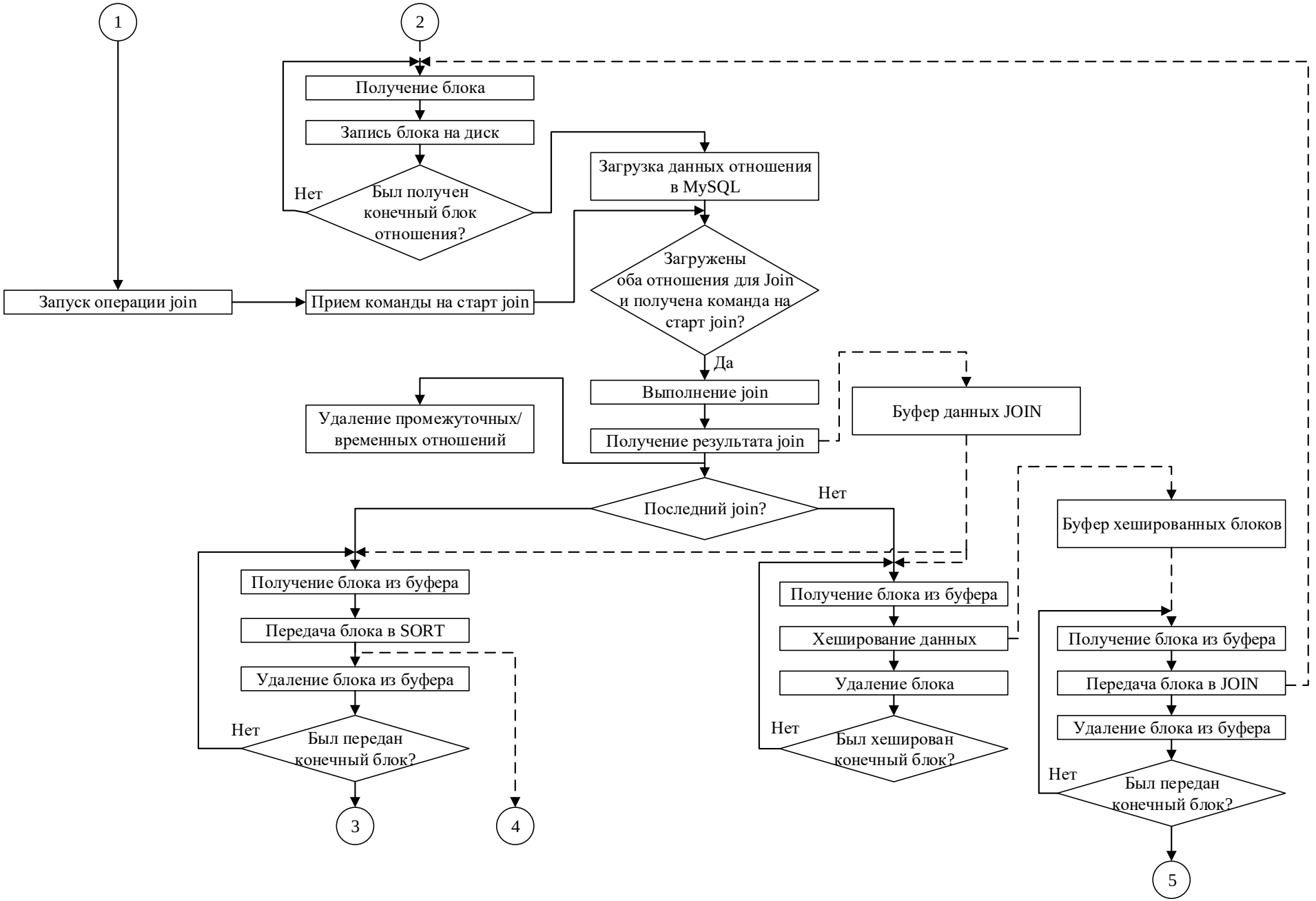


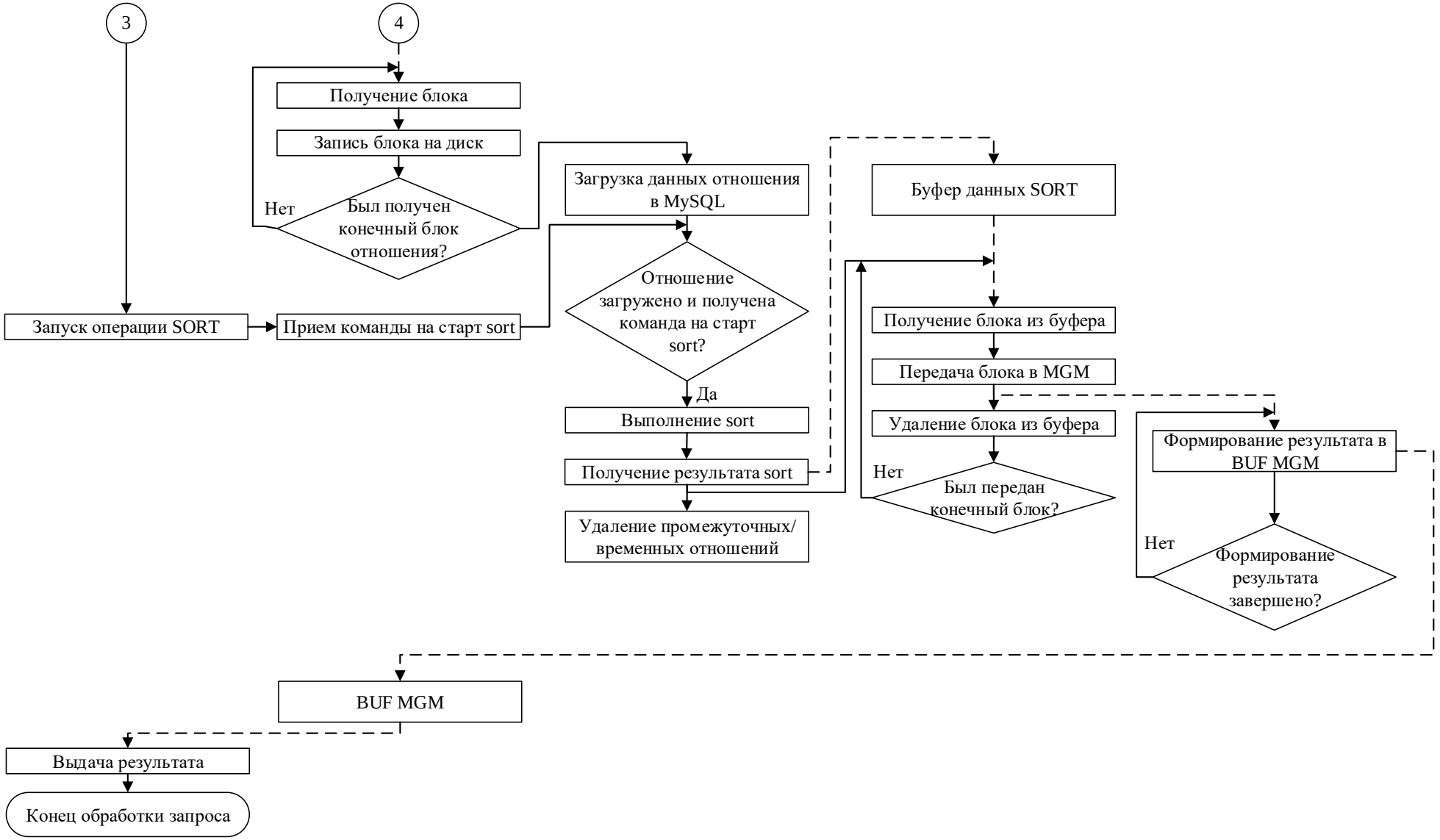

1
2
Получение блока
Запись блока на диск
Нет
Был получен конечный блок отношения?
Загрузка данных отношения в MySQL
Буфер данных JOIN
Запуск операции join
Прием команды на старт join
Загружены оба отношения для Join и получена команда на старт join?
Да
Выполнение join
Удаление промежуточных/ временных отношений
Получение результата join
Последний join?
Нет
Получение блока из буфера
Передача блока в SORT
Удаление блока из буфера
Нет
Был передан конечный блок?
3
4
Получение блока из буфера
Хеширование данных
Передача блока в JOIN
Удаление блока
Нет
Был передан конечный блок?


3
4
Получение блока
Запись блока на диск
Нет
Был получен конечный блок отношения?
Загрузка данных отношения в MySQL
Буфер данных SORT
Запуск операции SORT
Прием команды на старт sort
Отношение загружено и получена команда на старт sort?
Да
Получение блока из буфера
Передача блока в MGM
Выполнение sort
Удаление блока из буфера
Формирование результата в BUF MGM
Получение результата sort
Нет
Был передан конечный блок?
Удаление промежуточных/ временных отношений
Нет
Формирование результата завершено?
BUF MGM
Выдача результата
Конец обработки запроса

B.3. Итерация 4




1
2
Получение блока
Запись блока на диск
Нет
Был получен конечный блок отношения?
Загрузка данных отношения в MySQL
Запуск операции join
Прием команды на старт join
Загружены оба отношения для Join и получена команда на старт join?
Да
Выполнение join
Получение результата join
Буфер данных JOIN
Удаление промежуточных/ временных отношений
Последний join?
Нет
Буфер хешированных блоков
Получение блока из буфера
Передача блока в SORT
Удаление блока из буфера
Нет
Был передан конечный блок?
Получение блока из буфера
Хеширование данных
Удаление блока
Нет
Был хеширован конечный блок?
Получение блока из буфера
Передача блока в JOIN
Удаление блока из буфера
Нет
Был передан конечный блок?
3
4
5


3
4
Получение блока
Запись блока на диск
Нет
Был получен конечный блок отношения?
Загрузка данных отношения в MySQL
Буфер данных SORT
Запуск операции SORT
Прием команды на старт sort
Отношение загружено и получена команда на старт sort?
Да
Получение блока из буфера
Передача блока в MGM
Выполнение sort
Удаление блока из буфера
Формирование результата в BUF MGM
Получение результата sort
Нет
Был передан конечный блок?
Удаление промежуточных/ временных отношений
Нет
Формирование результата завершено?
BUF MGM
Выдача результата
Конец обработки запроса

ПРИЛОЖЕНИЕ Г. РЕГИСТРАЦИЯ ПРОГРАММ ДЛЯ ЭВМ

РОССИЙСКАЯ ФЕДЕРАЦИЯ

СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2017611785

Программа региональной балансировки нагрузки к базе данных консервативного типа на кластерной платформе «PerformSys»

Правообладатель: ***Федеральное государственное бюджетное образовательное учреждение высшего образования «Казанский национальный исследовательский технический университет им. А.Н. Туполева-КАИ» (КНИТУ-КАИ) (RU)***

Автор: ***Классен Роман Константинович (RU)***

Заявка № **2016663729**

Дата поступления **14 декабря 2016 г.**

Дата государственной регистрации в Реестре программ для ЭВМ ***09 февраля 2017 г.***

Руководитель Федеральной службы по интеллектуальной собственности

Г.П. Ивлиев

ПРИЛОЖЕНИЕ Д. АКТ О ВНЕДРЕНИИ РЕЗУЛЬТАТОВ ДИССЕРТАЦИИ В УЧЕБНЫЙ ПРОЦЕСС КНИТУ-КАИ

УТВЕРЖДАЮ

Проректор по образовательной
деятельности КНИТУ–КАИ
_____ А.А. Лопатин
«26» ____ 2019 г.

А К Т

о внедрении результатов диссертационной работы Классена Р.К.
«КОНСЕРВАТИВНЫЕ СУБД КЛАССА BIGDATA С РЕГУЛЯРНЫМ ПЛАНОМ ОБРАБОТКИ ЗАПРОСОВ НА КЛАСТЕРНОЙ ПЛАТФОРМЕ»
В учебный процесс КНИТУ–КАИ

Мы, ниже подписавшиеся,

- заведующий кафедрой компьютерных систем КНИТУ-КАИ, доцент И.С. Вершинин,
- научный руководитель лаборатории системотехники кафедры компьютерных систем, профессор В.А. Райхлин,

составили настоящий акт в том, что программная система Clusterix-N, разработанная Классеном Романом Константиновичем в рамках его кандидатской диссертации, внедрена в учебный процесс кафедры Компьютерных систем КНИТУ-КАИ.

Данная система используется при проведении лабораторных занятий для студентов направления «Информатика и вычислительная техника» по дисциплине «Параллельные вычисления» с 1 сентября 2018 г.

Формы внедрения:

1. Пакет программных модулей и исходные коды системы Clusterix-N.
2. Методические указания к выполнению лабораторной работы «Параллельные СУБД консервативного типа».

Заведующий кафедрой КС КНИТУ–КАИ, к.т.н., доцент _____ И.С. Вершинин

научный руководитель лаборатории системотехники кафедры КС, д.ф.-м.н., профессор _____ В.А. Райхлин

ПРИЛОЖЕНИЕ Е. АКТ О ВНЕДРЕНИИ РЕЗУЛЬТАТОВ ДИССЕРТАЦИИ В ООО «АМР СИСТЕМЫ»

УТВЕРЖДАЮ

Директор ООО «АМР Системы»
Сафонов Артем Алексеевич
«09» декабря 2018 г.

А К Т

о внедрении результатов диссертационной работы Классена Р.К.
«КОНСЕРВАТИВНЫЕ СУБД КЛАССА BIGDATA С РЕГУЛЯРНЫМ ПЛАНОМ ОБРАБОТКИ ЗАПРОСОВ НА КЛАСТЕРНОЙ ПЛАТФОРМЕ»

Комиссия ООО «АМР Системы» в составе директора Сафонова А.А. и ведущего инженера-программиста Костылева М.Ю. рассмотрела вопрос о внедрении результатов диссертационной работы Классена Романа Константиновича и установила следующее:

Внедрение разработанной в диссертации технологии PerformSys в ООО "АМР Системы" выполнено в коммерческий продукт "ПроВесы" (http://amr-systems.ru/avtomatizirovannaya-sistema-upravleniya-vzveshivaniya) как модификация подсистемы синхронизации данных между центральным сервером системы и клиентскими АРМ (автоматизированное рабочее место). Подсистема синхронизации обеспечивает консолидацию данных на центральном сервере. Управление процессом синхронизации производится с центрального сервера.

Процесс синхронизации выполняется по следующим шагам, выполняющихся по расписанию (каждые 5 минут):

1. Запрос журнала изменений со времени последнего получения изменений для каждого отношения в БД, развернутой на АРМ.
2. Сравнение полученных журналов со своим журналом.
3. Рассылка требований передачи новых/измененных данных в центральный сервер.
4. Передача данных новых/измененных данных в БД на АРМ.

Технология PerformSys была применена для наиболее эффективного использования вычислительных ресурсов и исключения перегрузки АРМ при реализации пунктов 1,3,4 процесса синхронизации.

В процессе внедрения PerformSys претерпел следующие изменения:

1. Инструментальная СУБД заменена на MS SQL.
2. Добавлена возможность работы не только с запросами, но и с заданиями (специальными пакетами, которые содержат инструкции для других модулей).
3. Программа *balancer* изменена для реализации передачи задания/запроса в указанный АРМ.

4. Количество задействованных ядер определяется конфигурацией системы ПроВесы.

Разработанная в рамках диссертационного исследования программа *balancer* внедрена в подсистему синхронизации центрального сервера, а программа *server* - в подсистему синхронизации АРМ. При этом была сохранена возможность масштабирования системы с помощью программы *router*.

Результаты внедрения использованы при автоматизации контроля рудоперевозок в ООО «Кинросс Дальний Восток» Филиал Корпорации Кинросс Голд, где используются по настоящее время.

Директор ООО «АМР Системы» Сафонов А.А.

Ведущий инженер-программист Костылев М.Ю.