ЮЖНО-УРАЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

На правах рукописи

ЛЕПИХОВ Андрей Валерьевич

МЕТОДЫ ОБРАБОТКИ ЗАПРОСОВ В СИСТЕМАХ УПРАВЛЕНИЯ БАЗАМИ ДАННЫХ ДЛЯ МНОГОПРОЦЕССОРНЫХ СИСТЕМ С ИЕРАРХИЧЕСКОЙ АРХИТЕКТУРОЙ

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

Диссертация на соискание ученой степени кандидата физико-математических наук

Научный руководитель:
СОКОЛИНСКИЙ Леонид Борисович,
доктор физ.-мат. наук, профессор

Челябинск – 2008

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

АКТУАЛЬНОСТЬ ТЕМЫ

В настоящее время все большее распространение получают параллельные системы баз данных, ориентированные на мультипроцессоры с иерархической архитектурой [15]. Это связано с тем, что современные многопроцессорные системы в большинстве случаев организуются по иерархическому принципу. Большая часть суперкомпьютеров сегодня имеют двухуровневую кластерную архитектуру. В соответствии с данной архитектурой многопроцессорная система строится как набор однородных вычислительных модулей, соединенных высокоскоростной сетью. При этом каждый вычислительный модуль является в свою очередь многопроцессорной системой с разделяемой памятью. Если в системе используются еще и многоядерные процессоры, то получаем третий уровень иерархии.

Другим источником многопроцессорных иерархий являются Grid-технологии [48], позволяющие объединять несколько различных суперкомпьютеров в единую вычислительную систему. Подобная Grid-система будет иметь многоуровневую иерархическую структуру. На нижних уровнях иерархии располагаются процессоры отдельных кластерных систем, соединенные высокоскоростной внутренней сетью. На верхних уровнях располагаются вычислительные системы, объединенные корпоративной сетью. Высший уровень иерархии может представлять сеть Интернет.

Иерархические многопроцессорные системы обладают рядом особенностей. В рамках одного уровня иерархии скорость обмена сообщениями между вычислительными узлами практически одинакова. При переходе на более высокие уровни иерархии скорость межпроцессорных обменов может уменьшаться на несколько порядков. Следовательно, системы управления базами данных (СУБД) для многопроцессорных иерархий, или кратко – *иерархические СУБД*, занимают промежуточное положение между парал-

лельными СУБД, для которых характерна одинаковая скорость межпроцессорных обменов, и распределенными СУБД, в которых скорость обмена сообщениями может значительно отличаться для разных пар вычислительных узлов. Детально сходства и различия этих трех классов СУБД будут обсуждаться в главе 1. Анализ показывает, что не все алгоритмы и методы, используемые в параллельных и распределенных СУБД, могут быть перенесены в иерархические СУБД, а те алгоритмы и методы, которые такой перенос допускают, как правило, нуждаются в существенной адаптации. В соответствии с этим *актуальной* является задача разработки эффективных алгоритмов и методов обработки запросов, ориентированных на применение в системах баз данных с иерархической многопроцессорной архитектурой, масштабируемой до десятков тысяч процессорных узлов.

ОБЗОР РАБОТ ПО ТЕМАТИКЕ ИССЛЕДОВАНИЯ

Современные исследования в области иерархических СУБД опираются на алгоритмы и методы, применяемые в распределенных и параллельных системах баз данных.

Основы технологии распределенной обработки запросов в реляционных базах данных были впервые разработаны и реализованы в СУБД System R* [27]. Дальнейшее развитие технологии распределенной обработки запросов получили в ряде исследовательских прототипов (см., например, [21, 35, 105, 106]). Однако до появления успешных коммерческих проектов распределенных СУБД пришлось решить ряд проблем, таких как синхронизация доступа к распределенным данным [29, 62, 92], управление распределенными транзакциями [55, 109], поддержка репликации данных [34, 103, 108], многоуровневая защита данных [44, 90] и др. Более полный обзор результатов и исследовательских проблем в области распределенных СУБД можно найти в работах [53, 72, 107].

Базовые принципы построения параллельных СУБД были разработаны и реализованы в девяностых годах прошлого века в ряде прототипов, из которых наиболее известными являются Volcano [60], GAMMA [41], BUBBA [22] и GRACE [51]. В прототипе GRACE (архитектура с общей памятью) были предложены параллельные алгоритмы соединения, основанные на сортировке и хешировании [71]. В прототипе параллельной СУБД Volcano (архитектура с общей памятью) реализована операторная модель параллельного выполнения запросов [58]. В рамках проекта GAMMA (архитектура без совместного использования ресурсов) реализована технология горизонтальной фрагментации базы данных [77] и предложены параллельные алгоритмы обработки запросов, основанные на хешировании [42]. Среди ключевых направлений проекта BUBBA (архитектура без совместного использования ресурсов) можно выделить следующие: разработка стратегии оптимального размещения данных [37], управление параллельными потоками данных [17] и автоматическое распараллеливание запросов [23].

Проведенные исследования показали, что наилучшей базовой архитектурой для параллельных систем баз данных является архитектура без совместного использования ресурсов [104]. Поэтому в дальнейшем основные усилия были сконцентрированы в области поиска эффективных методов и алгоритмов параллельной обработки запросов в системах без совместного использования ресурсов. В рамках этого направления решались задачи оптимизации запросов для параллельной обработки [52, 65], адаптивной обработки запросов [40, 56] и управления вычислительными ресурсами [32, 57]. Результатом данных научных исследований стало появление ряда коммерческих параллельных СУБД с архитектурой без совместного использования ресурсов, среди которых наиболее известными являются Non Stop SQL [28], Teradata [87] и DB2 Parallel Edition [5].

Для параллельных систем баз данных без совместного использования ресурсов особое значение имеют проблема балансировки загрузки [66, 113],

и связанная с ней проблема размещения данных [83, 112]. В работе [75] было показано, что перекосы [82], возникающие при обработке запросов в параллельных системах баз данных без совместного использования ресурсов, могут приводить к практически полной деградации производительности системы. Поэтому теме балансировки загрузки посвящено значительное количество работ (см., например, [63, 66, 91, 99, 102, 113]). Исследования проблемы балансировки загрузки осуществлялись в рамках следующих двух основных подходов. Первый подход состоит в предупреждении перекосов и предполагает разработку стратегии размещения данных, при которой нагрузка будет равномерно распределяться между процессорными узлами системы [67, 78, 99]. Наибольший интерес здесь представляет подход, называемый адаптивным распределением [68, 79]. В соответствии с этим подходом способ размещения данных может меняться, когда СУБД находит лучшую стратегию размещения данных. Второй подход заключается в динамическом перераспределении данных между вычислительными узлами в процессе обработки запроса [64, 66]. Однако следует отметить, что в общем виде проблема балансировки загрузки для параллельных систем баз данных без совместного использования ресурсов не решена до сих пор.

В середине девяностых годов начались исследования в области иерархических СУБД. В работе [25] предложен метод динамической балансировки загрузки при обработке запросов в двухуровневых иерархических многопроцессорных системах. Первый (локальный) уровень представляет собой SMP–систему, второй (глобальный) – набор SMP-узлов, объединенных коммуникационной сетью. Главной идеей метода является использование межоператорного параллелизма на первом уровне иерархии и фрагментного параллелизма на втором уровне. При этом эффективная динамическая балансировка происходит только на первом уровне. Данный метод не может быть расширен для использования в многопроцессорных иерархиях с большим количеством уровней.

В работе [45] была предложена методика оптимизации операций параллельного ввода/вывода, ключевым элементом которой является использование репликации для уменьшения времени простоя вычислительных узлов. Развитие данной методики в работе [46] привело к разработке стратегии распределения данных в параллельных СУБД, использующих репликацию для оптимизации параллельного ввода/вывода. Итогом этих исследований стала работа [47], в которой предложено решение проблемы балансировки загрузки для архитектур вычислительных систем без совместного использования ресурсов, основанное на репликации. Данное решение позволяет уменьшить накладные расходы на передачу данных по сети в процессе балансировки загрузки. Однако метод балансировки загрузки предлагается в весьма узком контексте пространственных баз данных в специфическом сегменте диапазонных запросов.

Еще одним актуальным направлением исследований является параллельная обработка запросов в среде грид [57, 97]. В работе [81] выполнен анализ операции соединения методом хеширования и выделены наиболее значимые параметры, влияющие на скорость выполнения оператора соединения. В рамках проекта GParGRES [73] разработано программное обеспечение промежуточного слоя для параллельной обработки OLAP-запросов в грид средах. Проект GParGRES реализован на базе СУБД PostgreSQL и предназначен для использования в двухуровневых многопроцессорных иерархиях. Нижний уровень представляет собой вычислительный кластер. Верхний уровень представляет собой грид-систему, являющуюся набором вычислительных кластеров, объединенных некоторой коммуникационной сетью. В целях балансировки загрузки системы при обработке сложных OLAP-запросов база данных дублируется на всех сайтах грид-системы. Следует, отметить, что методы параллельной обработки запросов, предложенные в данной работе, не могут использоваться в многопроцессорных иерархиях с большим числом уровней. Вместе с тем, размещение полной

копии базы данных на каждом из сайтов грид-системы может порождать значительные накладные расходы при распространении обновлений.

Проведенный анализ литературы показывает, что на сегодняшний день отсутствуют эффективные методы и алгоритмы обработки запросов, размещения данных и балансировки загрузки, ориентированные на многопроцессорные системы с иерархической архитектурой. В связи с этим, актуальной является задача разработки новых методов и алгоритмов обработки запросов, которые позволят эффективно использовать потенциал многопроцессорных иерархий для создания высокопроизводительных систем баз данных, масштабируемых до десятков тысяч узлов.

ЦЕЛЬ И ЗАДАЧИ ИССЛЕДОВАНИЯ

Целью диссертационного исследования является разработка эффективных методов и алгоритмов обработки запросов, размещения данных и балансировки загрузки, ориентированных многопроцессорные системы с иерархической архитектурой и их реализация в прототипе *иерархической* СУБД. Для достижения этой цели необходимо было решить следующие ***задачи***:

1. Разработать и аналитически исследовать стратегию размещения и репликации базы данных для многопроцессорных иерархических систем.
2. Разработать эффективный алгоритм динамической балансировки загрузки на основе предложенной стратегии размещения данных.
3. Разработать метод параллельной обработки запросов для многопроцессорных иерархий, использующий предложенные стратегию размещения данных и алгоритм балансировки загрузки.
4. Реализовать разработанные методы и алгоритмы в прототипе иерархической СУБД «Омега».
5. Провести вычислительные эксперименты для оценки эффективности предложенных решений.

МЕТОДЫ ИССЛЕДОВАНИЯ

Проведенные в работе исследования базируются на реляционной модели данных и используют методы системного программирования. Для решения поставленных задач применялся математический аппарат, в основе которого лежит теория графов, предоставляющая возможность изучения и моделирования многопроцессорных иерархических конфигураций систем баз данных.

НАУЧНАЯ НОВИЗНА

Научная новизна работы заключается в следующем:

1. Предложена модель симметричной многопроцессорной иерархической системы.
2. Предложен метод частичного зеркалирования, использующий функцию репликации, которая сопоставляет каждому уровню иерархии определенный коэффициент репликации.
3. Получены аналитические оценки трудоемкости формирования и обновления реплик в методе частичного зеркалирования.
4. Разработан новый алгоритм балансировки загрузки для параллельных СУБД с иерархической архитектурой.
5. Разработан метод обработки запросов, допускающий эффективную балансировку загрузки для иерархических систем баз данных.

ТЕОРЕТИЧЕСКАЯ И ПРАКТИЧЕСКАЯ ЦЕННОСТЬ

Теоретическая ценность работы состоит в том, что дано формальное описание симметричной многопроцессорной системы с иерархической архитектурой. Представлены доказательства теорем об оценке размера реплик и оценке трудоемкости формирования реплик без учета помех для метода частичного зеркалирования.

Практическая ценность работы заключается в том, что предложенный метод частичного зеркалирования совместно с разработанным алгоритмом балансировки загрузки может использоваться для решения проблемы перекосов по данным в широком классе приложений систем баз данных для вычислительных кластеров и грид-систем.

СТРУКТУРА И ОБЪЕМ РАБОТЫ

Диссертация состоит из введения, четырех глав, заключения и библиографии. Объем диссертации составляет 102 страницы, объем библиографии – 113 наименований.

СОДЕРЖАНИЕ РАБОТЫ

Первая глава, «Многопроцессорные иерархии», посвящена описанию и исследованию многопроцессорных вычислительных систем с иерархической архитектурой. Рассматриваются предпосылки появления многопроцессорных иерархических систем, описываются уровни иерархии в такой системе, строится общая математическая модель многопроцессорной иерархии. Центральной частью главы является сравнительный анализ параллельных, распределенных и иерархических СУБД. В заключительной части главы приводится описание механизмов параллельной обработки запросов, описывается реализация операторной и скобочной моделей в исполнителе запросов прототипа иерархической СУБД «Омега». Приводится общая схема параллельной обработки запросов.

Во второй главе, «Размещение данных и балансировка загрузки», предлагается метод балансировки загрузки для описанных в предыдущей главе иерархических СУБД. Приводится описание методов распределения и репликации данных в многопроцессорных иерархиях. Вводится понятие коэффициента репликации и функции репликации. Описывается оригинальный метод частичного зеркалирования. Доказываются теоремы для оценки размера реплик и оценки трудоемкости формирования реплик без учета по-

мех, которые играют важную роль при решении проблемы минимизации накладных расходов на поддержку согласованности реплик в методе частичного зеркалирования. Приводится схема работы параллельного агента, описывается алгоритм балансировки загрузки и приводится стратегия выбора аутсайдера. В заключении главы предлагается формула для вычисления функции балансировки загрузки.

В третьей главе, «Иерархическая СУБД «Омега»», описывается процесс проектирования и реализации СУБД «Омега». В первой части главы приводятся общие требования к данной СУБД, форматы входных и выходных данных, описывается структура иерархической СУБД. Для описания структуры СУБД «Омега» используются диаграммы объектов и размещения подсистем. Для ключевых подсистем представлено описание вариантов использования. Во второй части главы детально описываются основные аспекты реализации СУБД «Омега». Приводится описание реализации оператора обмена **exchange**, в основе которой лежит механизм пакетирования. Описывается реализация механизма балансировки загрузки.

В четвертой главе, «Вычислительные эксперименты», приводится описание алгоритма соединения хешированием в оперативной памяти, использованного для проверки предложенных методов и алгоритмов. Описываются параметры среды выполнения экспериментов и виды исследуемых перекосов. Обсуждаются результаты экспериментов, проведенных на высокопроизводительном вычислительном кластере СКИФ Урал. Исследуется влияние интервала балансировки и размера сегмента на эффективность балансировки. Изучается влияние балансировки на время выполнения запроса и масштабируемость СУБД.

В заключении суммируются основные результаты диссертационной работы, выносимые на защиту, приводятся данные о публикациях и апробациях автора по теме диссертации, и рассматриваются направления дальнейших исследований в данной области.

ГЛАВА 1. МНОГОПРОЦЕССОРНЫЕ ИЕРАРХИИ

1.1. Предпосылки появления многопроцессорных иерархий

Появление и широкое распространение иерархических многопроцессорных систем обусловлено следующими основными тремя факторами:

– создание многоядерных процессоров;
– широкое распространение вычислительных систем с кластерной архитектурой;
– развитие грид-технологий.

Рассмотрим каждый из этих трех факторов более подробно.

1.1.1. Многоядерные процессоры

В настоящее время переход к многоядерным процессорам является магистральным направлением микропроцессорной индустрии [26]. Многоядерные процессоры сегодня повсеместно применяются как в высокопроизводительных вычислительных системах, так и в настольных компьютерах, включая ноутбуки. В последней редакции (июнь 2008 г.) списка TOP500 [84] самых мощных компьютеров мира более 95% процессоров имеют два и более ядра. Основная причина появления многоядерных процессоров состоит в следующем. В одноядерных процессорах повышение производительности происходило главным образом за счет увеличения тактовой частоты процессора. Однако при достижении некоторого уровня частот, дальнейшее повышение частоты работы процессора сопряжено со значительным ростом энергопотребления и, как следствие, с повышенным тепловыделением [24]. Вместе с этим, преимущества от дальнейшего повышения тактовой частоты в значительной мере теряются из-за задержек при обращении к памяти, поскольку время доступа к памяти не соответствует скорости работы процессора. Многоядерные процессоры сделали возможным дальнейшее увеличение производительности без существенного увеличения

энергопотребления и тепловыделения. При этом повышение производительности определяется не тактовой частотой, а количеством ядер на одном кристалле. Архитектура многоядерного процессора предусматривает размещение двух и более вычислительных ядер на одном кристалле [88]. Каждое вычислительное ядро воспринимается операционной системой как отдельный процессор с полным инструментарием (приватная кэш-память первого уровня, блок плавающей арифметики и др.). Ядра процессора взаимодействуют между собой через общую кэш-память второго уровня.

В современной микропроцессорной индустрии происходит быстрое наращивание количества ядер в процессоре. Сегодня в массовом порядке промышленно изготавливаются и поставляются на рынок четырехъядерные процессоры. Завершен этап проектирования 16-ядерного процессора Rock от компании Sun [110]. Начало поставок на рынок нового 8–ядерного процессора Nehalem, выполненного компанией Intel по технологии 45 нм, запланировано на конец 2008 года. Предложен прототип 80-ядерного процессора Polaris [111]. По прогнозам Intel в ближайшее десятилетие следует ожидать появления процессоров с сотнями ядер на одном кристалле [24].

Помимо традиционных процессоров, перспективным направлением развития многоядерных архитектур являются специализированные процессоры [9]. Среди наиболее известных специализированных процессоров следует отметить программируемые графические процессоры (Graphical Processor Unit) и процессоры Cell, совместно разработанные компаниями IBM, Sony и Toshiba [31]. В состав процессора Cell входит ядро PowerPC и восемь специальных процессорных элементов SPE. SPE взаимодействуют между собой и ядром PowerPC посредством высокоскоростной шины. Ядро PowerPC играет роль координатора, перераспределяющего нагрузку между SPE. В настоящее время пиковая производительность процессора Cell достигла 204 Gflops. Следует отметить, что самый мощный в мире компьютер

RoadRunner [39] представляет собой гибридную платформу, содержащую 12 240 процессоров Cell и 6 562 двухъядерных процессоров Opteron DC.

1.1.2. Вычислительные кластеры

На сегодняшний день наиболее популярной архитектурой для высокопроизводительных вычислительных систем является кластерная архитектура [2]. В TOP500 доля кластерных систем составляет 80%. Популярность кластерных систем обусловлена следующими основными факторами. Кластерные системы ориентируются на стандартные процессоры, выпускаемые в массовом порядке такими компаниями как AMD и Intel. Для организации сети передачи данных между вычислительными узлами используются технологии, являющиеся индустриальными стандартами, такие как Gigabit Ethernet или Infiniband [30]. Это обеспечивает кластерным системам чрезвычайно выгодное соотношение «цена/производительность». Контекстная независимость отдельного вычислительного узла кластера от системы в целом позволяет разрабатывать кластерные системы с высоким уровнем отказоустойчивости. Важным показателем кластерных систем является простота масштабирования, которая обеспечивается децентрализованным характером управления кластером и невысокой стоимостью приобретения и подключения дополнительных вычислительных ресурсов. В настоящее время успешно эксплуатируются кластерные системы с тысячами узлов.

Кластер представляет собой связанный единой коммуникационной сетью набор вычислительных узлов, используемый в качестве единого ресурса [89]. Вычислительный узел практически является полноценной компьютерной системой, обладающей всем необходимым для самостоятельного функционирования инструментарием, включая процессоры, память, подсистему ввода/вывода, блок питания и др. Вычислительные узлы взаимодействуют между собой посредством передачи сообщений по коммуникационной сети. Внутри вычислительного узла процессоры обмениваются со-

общениями через общую оперативную память. При этом скорость межпроцессорных обменов внутри вычислительного узла может превосходить скорость межузлового взаимодействия на два-три порядка [20].

Основной тенденцией современной индустрии кластерных систем является продолжение наращивания производительности за счет увеличения количества вычислительных узлов. Вместе с тем, идет поиск новых кластерных архитектур, появляются гибридные кластерные системы. Наиболее производительный гибридный суперкомпьютер Roadrunner занимает первое место в рейтинге TOP500. Одним из перспективных направлений являются реконфигурируемые кластеры, основанные на использовании технологии FPGA [85]. Такие кластеры способны изменять структуру межузловой коммуникационной сети под управлением программного обеспечения. Предложены иерархические кластерные конфигурации [1], в которых отдельные кластеры объединяются в единую кластерную конфигурацию на базе локальных или глобальных сетей передачи данных.

1.1.3. Грид

В последние годы наблюдается устойчивый рост пропускной способности каналов передачи данных, используемых в сети Интернет. Это связано с интенсивным развитием сетевых технологий. По оценкам специалистов, пропускная способность каналов передачи данных ежегодно возрастает на 50% [95]. Интенсивное развитие глобальных компьютерных сетей создало основу для появления грид-систем. Под грид-системой понимается согласованная, открытая и стандартизованная среда, которая обеспечивает гибкое, безопасное, скоординированное разделение ресурсов в рамках виртуальной организации [50]. Грид-системы ориентированы, прежде всего, на решение крупномасштабных вычислительных задач [3]. Существующие на сегодняшний день грид-системы объединяют десятки и сотни тысяч компьютеров и обладают значительными вычислительными ресурсами. Так,

например, одна из наиболее развитых грид-систем TeraGrid [93] имеет суммарную пиковую производительность более одного петафлопса.

В соответствии с масштабом коммуникационной сети, грид-системы можно разделить на три вида: корпоративный грид (intra-grid), кооперативный грид (extra-grid) и глобальный грид (inter-grid) [18]. *Корпоративные грид-системы* формируются путем объединения кластеров одной организации локальной соединительной сетью в целях оптимизации использования вычислительных ресурсов. Основой такой грид-системы являются протоколы управления вычислительными ресурсами. *Кооперативные грид-системы* формируются путем объединения корпоративных грид-систем различных организаций в целях объединения усилий для решения больших задач [48]. Данный вид грид-систем требует внедрения дополнительного слоя программного обеспечения, обеспечивающего политику управления безопасностью грид-системы. *Глобальные грид-системы* объединяют десятки кооперативных грид-систем в единую вычислительную среду посредством сети Интернет. Глобальные грид-системы обеспечивают использование вычислительных ресурсов миллионов компьютеров и реализуют наиболее масштабные вычислительные системы.

Основным стандартом, описывающим общую архитектуру программного обеспечения грид-систем, является стандарт OGSA [49]. В соответствии с данным стандартом грид-система имеет четыре программных слоя. На первом (нижнем) слое располагается программное обеспечение, управляющее аппаратными ресурсами системы. Основной задачей данного слоя является унификация ресурсов и представление их в виде абстрактных типов со стандартизованным множеством операций. Второй слой архитектуры OGSA обеспечивает передачу данных между вычислительными узлами и маршрутизацию, решает задачи аутентификации, защиты сообщений и авторизации. Третий слой определяет ряд протоколов и программных интерфейсов, которые предоставляют возможность удаленного использования

ресурсов грид. Данный слой отвечает за поиск ресурсов, мониторинг и управление операциями. Верхний уровень иерархии предоставляет сервисы для управления совокупностью грид ресурсов.

Развитие грид-систем связано с повышением пропускной способности и уменьшением неоднородности коммуникационных сетей. Так, в проекте TeraGrid пропускная способность сети на самом верхнем уровне грид-системы составляет 40 Гбит/с, что превосходит пропускную способность современных локальных сетей, построенных по технологии Gigabit Ethernet. Одним из перспективных направлений развития крупномасштабных грид-систем считается использование метода агентных вычислений [61], основой которого является представление приложения в виде набора параллельных агентов [96]. Данный подход позволяет приложениям эффективно адаптироваться к изменяющимся характеристикам вычислительной среды, что характерно для грид-систем.

В настоящее время в процесс развития грид-технологий во все большей степени вовлекаются коммерческие компании [19]. Компания Sun Microsystems предлагает программное обеспечение для создания грид-систем [54]. Продукция фирм Entropia и United Devices предназначена для построения грид-систем на основе сетей персональных компьютеров [33]. Компания IBM предлагает целый комплекс программных инструментов для создания грид-систем [74]. Фирма Platform Computing выпускает средства для построения неоднородных корпоративных грид-систем [38].

1.2. Структура многопроцессорной иерархической системы

В общем виде структура многопроцессорной иерархической системы представлена на рис. 1. Первый уровень иерархии представлен многоядерными процессорами. Обмен данными между процессорными ядрами наиболее эффективно может осуществляться через разделяемую кэш-память,

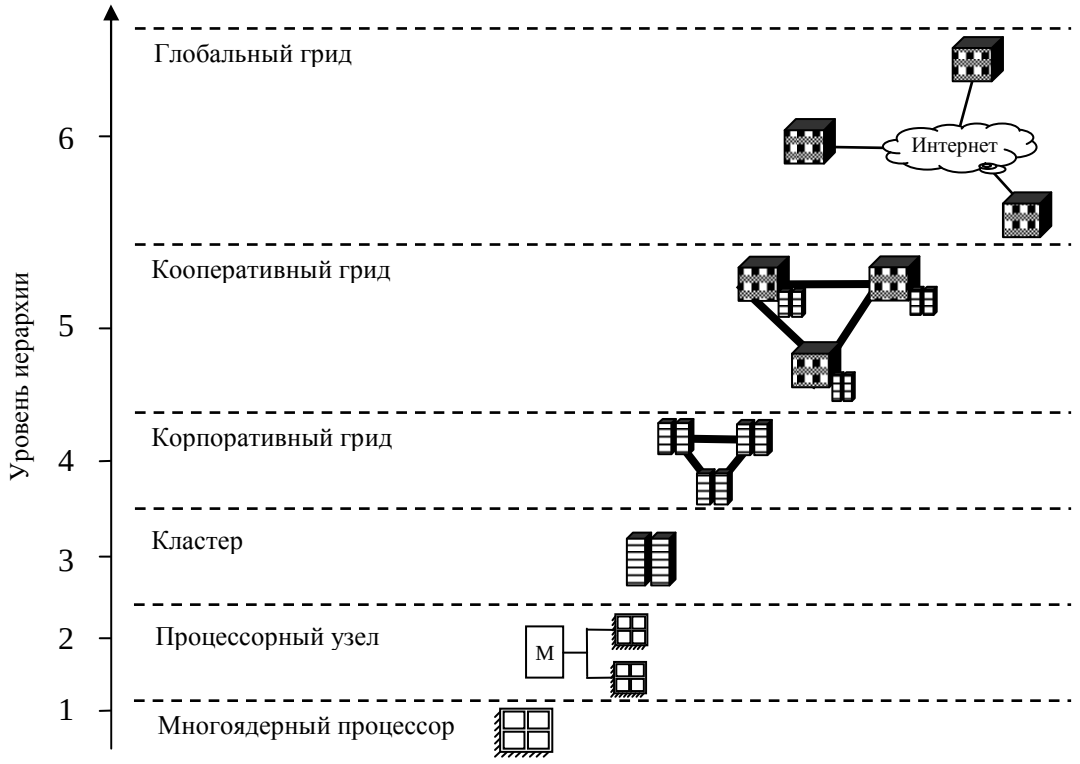

Рис. 1. Структура многопроцессорной иерархии.

расположенную на том же кристалле, на котором находятся процессорные ядра. Это наиболее быстрый способ организации межпроцессорных коммуникаций. Современные двухъядерные процессоры Intel 2 Core позволяют обеспечить скорость доступа к кэш-памяти на уровне 60 – 120 ГБайт/с [100].

На втором уровне иерархии многоядерные процессоры объединяются в процессорные узлы с SMP-архитектурой. Обмен данными между различными процессорами в SMP-системе наиболее эффективно организуется через общую область в оперативной памяти. В настоящее время, скорость доступа к оперативной памяти SMP-систем лежит в пределах 5-15 Гбайт/с [70].

На третьем уровне иерархии процессорные узлы, объединяются в кластер. Обмен данными между вычислительными узлами кластера организуется при помощи высокоскоростной коммуникационной сети. В настоящее время в качестве основной сети кластерных систем в большинстве

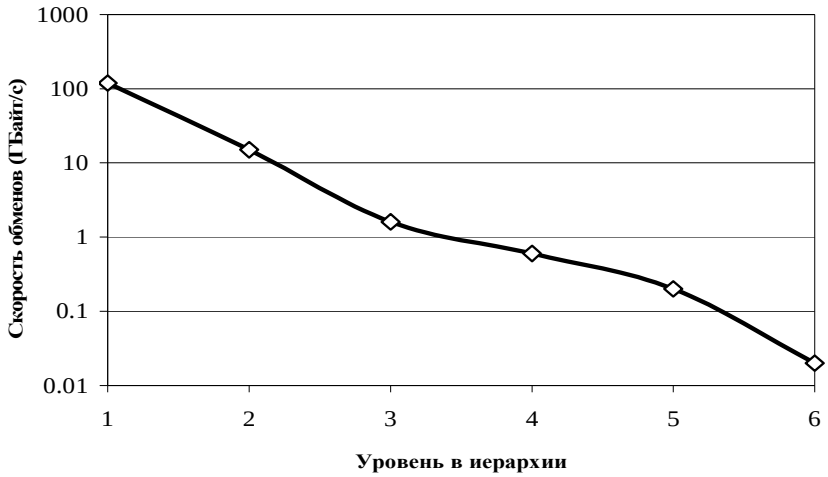

Рис. 2. Скорость межпроцессорных коммуникаций
в многопроцессорной иерархии.

случаев фигурирует сеть Infiniband, обеспечивающая пропускную способность в диапазоне 2 – 40 Гбит/с [30].

На четвертом уровне кластеры объединяются в корпоративную грид-систему. Для этой цели можно использовать сеть Gigabit Ethernet.

На пятом уровне корпоративные грид-системы объединяются в ассоциацию, образующую кооперативную грид-систему. Здесь в качестве соединительной сети могут быть использованы оптоволоконные линии муниципального уровня. Подобные каналы обычно обеспечивают передачу данных по схеме точка-точка со скоростью порядка 100 Мбит/с.

Шестой уровень представлен глобальными грид-сетями, которые используют в качестве коммуникационной сети сеть Интернет. На этом уровне можно говорить о скорости передачи данных по схеме точка-точка на уровне 10 Мбит/с.

На рис. 2 изображена зависимость скорости межпроцессорных коммуникаций от номера уровня в многопроцессорной иерархии. Мы видим, что скорость обменов уменьшается примерно на порядок при переходе на следующий уровень. Вычислительные узлы внутри каждого уровня могут

различаться по скорости межпроцессорных коммуникаций. Например, процессор на уровне 1 (в контексте рис. 2) AMD Opteron 285 обеспечивает скорость доступа к кэш памяти на уровне 50-80 Гбайт/с [100], что в среднем на 20 Гбайт/с меньше, чем скорость доступа к кэш памяти в процессорах Intel 2 Core. Однако величина этих различий незначительна по сравнению с разницей в скорости коммуникаций между узлами разных уровней и мы можем ее не учитывать. Таким образом, можно сформулировать следующее определение многопроцессорной иерархии.

Иерархическая многопроцессорная система – это многопроцессорная система, в которой процессоры объединяются в единую систему с помощью соединительной сети, имеющей иерархическую многоуровневую структуру и обладающей свойствами однородности по горизонтали и неоднородности по вертикали. *Однородность по горизонтали* означает, что в пределах одного уровня иерархии скорость обменов между двумя процессорами является постоянной величиной, независимо от того в каком поддереве иерархии эти процессоры находятся. *Неоднородность по вертикали означает*, что скорость обменов на разных уровнях иерархии существенно различается и этот факт должен учитываться в алгоритмах обработки запросов.

В следующем разделе будет построена формальная модель многопроцессорной иерархии применительно к системам баз данных.

1.3. Формальная модель многопроцессорной иерархии

В этом разделе описывается модель симметричной иерархической многопроцессорной системы баз данных [12]. Симметричная модель задает достаточно широкий класс реальных систем и является математическим фундаментом для описания стратегии распределения данных, предлагаемой в главе 2.

В основе симметричной модели лежит понятие *DM-дерева* [13], представляющего собой абстракцию иерархической многопроцессорной систе-

мы. Для определения DM-дерева используется понятие ориентированного дерева [6], которое представляет собой ориентированный граф с выделенной вершиной R, такой, что:

1) каждая вершина $V \neq R$ является начальной вершиной в точности одной дуги, обозначаемой через $e(V)$;
2) вершина R не является начальной вершиной ни одной из дуг;
3) вершина R является корнем, то есть для каждой вершины $V \neq R$ существует ориентированный путь от V до R.

Дадим определение DM-дерева.

DM-дерево – это ориентированное дерево, узлы которого относятся к одному из трех классов:

$\mathfrak{P}(T)$ – класс «процессорные модули»;

$\mathfrak{D}(T)$ – класс «дисковые модули»;

$\mathfrak{N}(T)$ – класс «модули сетевых концентраторов».

Для произвольного *DM*–дерева T будем обозначать множество всех его узлов как $\mathfrak{M}(T)$, множество всех дуг как $\mathfrak{E}(T)$.

С каждым узлом $v \in \mathfrak{M}(T)$ в *DM*-дереве T связывается *коэффициент трудоемкости* $\eta(v)$, являющийся вещественным числом, большим либо равным единицы. Коэффициент трудоемкости определяет время, необходимое узлу для обработки некоторой порции данных. В качестве такой порции данных может фигурировать, например, кортеж.

Дадим определение изоморфизма двух *DM*-деревьев.

DM-деревья A и B называются *изоморфными*, если существуют взаимно однозначное отображение f множества $\mathfrak{M}(A)$ на множество $\mathfrak{M}(B)$ и взаимно однозначное отображение g множества $\mathfrak{E}(A)$ на множество $\mathfrak{E}(B)$ такие, что:

1) узел v является конечным узлом дуги e в дереве A тогда и только

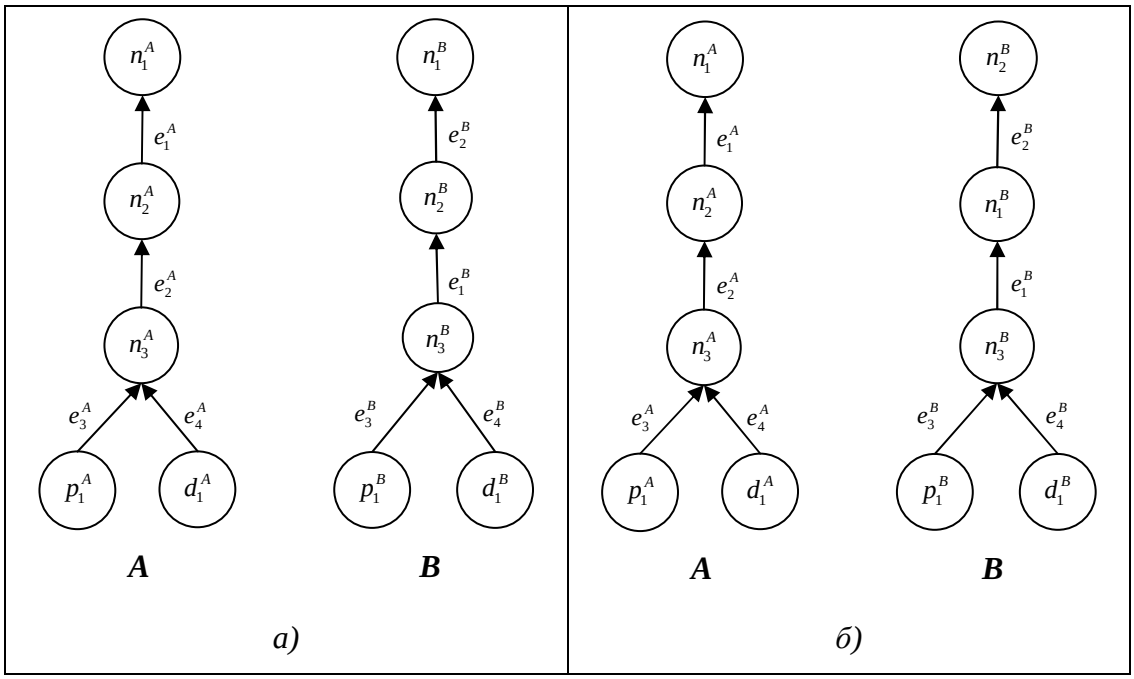

Рис. 3. Примеры отображений, не являющихся изоморфизмом:
$\mathrm{f}(n_i^A) = n_i^B$, $\mathrm{f}(p_i^A) = p_i^B$, $\mathrm{f}(d_i^A) = d_i^B$, $\mathrm{g}(e_i^A) = e_i^B$.

тогда, когда узел $\mathrm{f}(v)$ является конечным узлом дуги $\mathrm{g}(e)$ в дереве B;

2) узел w является начальным узлом дуги e в дереве A тогда и только тогда, когда узел $\mathrm{f}(w)$ является начальным узлом дуги $\mathrm{g}(e)$ в дереве B;

3) $p \in \mathfrak{P}(A) \quad \Leftrightarrow \quad \mathrm{f}(p) \in \mathfrak{P}(B)$;

4) $d \in \mathfrak{D}(A) \quad \Leftrightarrow \quad \mathrm{f}(d) \in \mathfrak{D}(B)$;

5) $n \in \mathfrak{N}(A) \quad \Leftrightarrow \quad \mathrm{f}(n) \in \mathfrak{N}(B)$;

6) $\eta(\mathrm{f}(v)) = \eta(v)$.

Упорядоченную пару отображений $\mathrm{q} = (\mathrm{f}, \mathrm{g})$ будем называть *изоморфизмом DM*-дерева A на *DM*-дерево B.

Необходимость условия 1 в определении изоморфизма следует из примера, показанного на рис. 3 (а). Здесь предполагаем, что коэффициент трудоемкости для всех узлов равен 1. Отображения **f** и **g** удовлетворяют всем требованиям изоморфизма, кроме условия 1. Однако нельзя признать $\mathrm{q} = (\mathrm{f}, \mathrm{g})$ изоморфизмом *DM*-дерева A на *DM*-дерево B, так как здесь на-

рушается отношение подчиненности узлов (узел *w* является *подчиненным* по отношению к узлу *v*, если существует дуга, направленная от *w* к *v*). Действительно, в *DM*-дереве A узел n_2^A подчинен узлу n_1^A, а в *DM*-дереве B узел $\mathrm{f}(n_2^A) = n_3^B$ имеет в подчинении узел $\mathrm{f}(n_3^A) = n_2^B$.

Заметим, что условие 2 из определения изоморфизма также не является избыточным, что подтверждается примером, изображенным на рис. 3 (б). Отображения **f** и **g** удовлетворяют всем требованиям изоморфизма, кроме условия 2. Однако нельзя признать $\mathrm{q} = (\mathrm{f}, \mathrm{g})$ изоморфизмом *DM*-дерева A на *DM*-дерево B, поскольку снова нарушается отношение подчиненности узлов. Действительно, в *DM*-дереве A узел n_2^A подчинен узлу n_1^A, а в *DM*-дереве B узел $\mathrm{f}(n_2^A) = n_2^B$ имеет в подчинении узел $\mathrm{f}(n_1^A) = n_1^B$.

Определим *уровень узла* по отношению к дереву *T* рекурсивно следующим образом [6]. Уровень корня дерева *T* равен нулю, а уровень любого другого узла на единицу больше, чем уровень корня минимального поддерева дерева *T*, содержащего данный узел. Отметим, что порядок такой нумерации будет противоположен порядку нумерации уровней на рис. 1.

Под *уровнем поддерева* дерева *T* будем понимать уровень корня этого поддерева в дереве *T*.

Два поддерева одного уровня называются *смежными*, если их корни являются братьями, то есть в дереве существует узел, являющийся общим родителем по отношению к корневым узлам данных поддеревьев.

Определим *высоту* ориентированного дерева *T* как максимальную длину пути в этом дереве.

Мы будем называть *DM*-дерево *T* высоты *H* *симметричным*, если выполняются следующие условия:

1) любые два смежных поддерева уровня $l < H$ являются изоморфными;
2) любое поддерево уровня $H - 1$ содержит в точности один диск и один процессор.

Условие 2 в определении симметричности представляет собой абстрактную модель SMP-системы в том смысле, что в контексте многопроцессорных иерархий все процессоры SMP-системы могут рассматриваться как один «мегапроцессор», а все диски – как один «мегадиск». Очевидно, что балансировка загрузки на уровне SMP-системы должна решаться специальными методами, так как SMP-система имеет общую память и все диски в равной мере доступны всем процессорам (см. по этому вопросу работы [80, 86]).

Узлом в симметричном дереве *T* будем называть поддерево высоты один. Таким образом, узел включает в себя один дисковый модуль, один процессорный модуль и один модуль сетевого концентратора.

Определим *степень узла* как количество дуг, входящих в этот узел. В симметричном дереве все узлы одного уровня имеют одинаковую степень, называемую *степенью данного уровня*.

1.4. СУБД для многопроцессорных иерархий

Под *иерархической СУБД* будем понимать систему управления базами данных, ориентированную на работу в многопроцессорных иерархиях. Иерархическая СУБД совмещает в себе черты параллельной и распределенной СУБД. Эта двойственная природа иерархической СУБД вытекает из базовых свойств многопроцессорной иерархии, сформулированных в разделе 1.2. Неоднородность по вертикали сближает иерархические СУБД с распределенными, а однородность по горизонтали – с параллельными. В данном разделе проводится сравнительный анализ этих трех видов систем управления базами данных по критериям, суммированным в табл. 1.

Контекстная независимость узла означает, что метод обработки запроса, эффективный для данного узла, остается эффективным для любого другого узла многопроцессорной системы. Это свойство, очевидно, имеет место для параллельных систем баз данных. Поэтому для

Табл. 1. Сравнительный анализ трех видов СУБД.

	Параллельная СУБД	**Распределенная СУБД**	**Иерархическая СУБД**
Контекстная независимость узла	+	–	+
Одноранговость соединительной сети	+	–	–
Фрагментный параллелизм	+	–	+
Репликация данных	–	+	+
Балансировка загрузки	+	–	+

параллельных СУБД помечаем соответствующую графу таблицы 1 знаком «плюс». В случае распределенных СУБД эффективность того или иного метода обработки запроса зависит от географического положения узла, удаленности и доступности данных, используемых в запросе, наличия или отсутствия необходимых реплик и др. Поэтому для распределенных СУБД в графе «Контекстная независимость узла» ставим «минус». Для иерархических СУБД в разделе 1.3 была предложена модель симметричной иерархической многопроцессорной системы баз данных, в которой любые два смежных поддерева являются изоморфными. В соответствии с этой моделью метод обработки запроса, эффективный в данном поддереве, остается эффективным для любого смежного поддерева. Причем это правило рекурсивно применимо для более высокого уровня. Следовательно, для иерархических СУБД в этой графе можно поставить «плюс».

Одноранговость соединительной сети означает, что скорость обмена данными между любыми двумя процессорными узлами является постоянной величиной. Очевидно, что этим свойством обладает только параллельная СУБД.

Фрагментный параллелизм подразумевает параллельную обработку запроса, основанную на фрагментации отношений (см. раздел 1.5). Подоб-

ный вид параллелизма является основным для параллельных СУБД и на практике никогда не применяется в распределенных СУБД. В иерархических СУБД фрагментный параллелизм может быть эффективно использован на нижних уровнях иерархии, поэтому в соответствующей графе таблицы ставим «плюс».

Репликация данных выражается в дублировании одних и тех же частей базы данных на различных узлах. Репликация данных интенсивно используется в распределенных СУБД для повышения доступности данных и для увеличения скорости обработки распределенных запросов. В классических параллельных СУБД при обработке запросов репликация, как правило, не используется. В иерархических СУБД репликация может быть эффективно использована для балансировки загрузки. Этот аспект будет детально рассмотрен в последующих главах диссертации.

Балансировка загрузки подразумевает наличие в СУБД эффективных механизмов перераспределения работ между процессорными узлами, задействованными в параллельной обработке одного запроса. Как уже отмечалось во введении, перекосы, возникающие при обработке запросов в параллельных системах баз данных без совместного использования ресурсов, могут приводить к существенной деградации производительности системы. Поэтому наличие такого механизма для параллельных СУБД является крайне необходимым. Первичным фактором возникновения перекосов является использование фрагментного параллелизма в системах с распределенной памятью. Поскольку мы констатировали, что использование фрагментного параллелизма является обязательным для иерархических СУБД, и эти СУБД имеют распределенную память, то для них также необходимо наличие эффективного механизма балансировки загрузки.

1.5. Организация параллельной обработки запросов

Существуют две хорошо известные общие модели, используемые при реализации параллельных исполнителей запросов, называемые скобочной и операторной моделями [59]. В данном разделе описываются реализации этих моделей, которые были использованы при создании прототипа иерархической СУБД «Омега» (см. главу 3).

1.5.1. Скобочный шаблон

Скобочный шаблон используется для унифицированного представления узлов дерева запроса. Схематично структура скобочного шаблона изображена на рис. 4. В качестве основных методов здесь фигурируют функции **`reset`** и **`next`**, реализующие итератор [4], который позволяет потребителю результата работы физического оператора получать по одному кортежу в каждый момент времени. Функция **`reset`** устанавливает итератор в состояние «Перед первым кортежем»: инициализирует структуры данных, необходимые для выполнения операции и вызывает функции **`reset`** соответствующих аргументов оператора. Функция **`next`** выдает очередной кортеж результирующего отношения, модифицирует значения во внутренних структурах данных, обеспечивая возможность получения последующих кортежей. Один или несколько раз вызывает функции **`next`** аргументов оператора. Если множество кортежей исчерпано, функция **`next`** завершает итеративный процесс выполнения операции и возвращает специальный EOF-кортеж.

Основными атрибутами скобочного шаблона являются следующие:

– выходной буфер, в который помещается очередной кортеж результата;
– КОП – код реляционной операции, реализуемой данным узлом;
– указатель на скобочный шаблон левого сына;
– указатель на скобочный шаблон правого сына («пусто» для унарных операций).

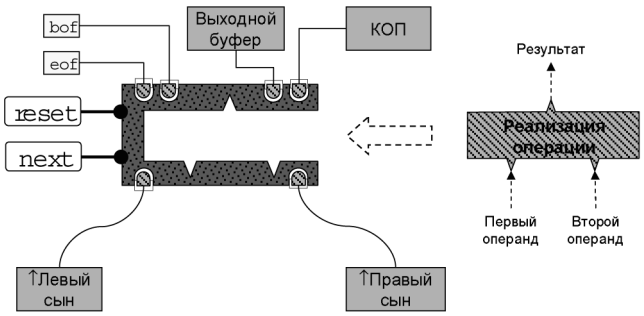

Рис. 4. Скобочный шаблон.

Сам по себе скобочный шаблон не содержит конкретной реализации реляционной операции. Однако, после оптимизации запроса СУБД «вставляет» в каждый скобочный шаблон ту или иную реализацию соответствующей реляционной операции. Например, для операции соединения можно выбрать «соединение вложенными циклами», «соединение слиянием», соединение хешированием» и др. Связь скобочного шаблона с конкретной реализацией операции осуществляется путем добавления еще одного специального атрибута в скобочный шаблон – «указатель на функцию реализации операции». В качестве параметра данной функции должен передаваться указатель на скобочный шаблон, к которому она привязывается.

1.5.2. Оператор обмена exchange

Оператор обмена **exchange** реализует концепцию операторной модели и может быть помещен в качестве узла в любое место дерева запроса.

Оператор **exchange** имеет два параметра, специально определяемых пользователем: номер *порта обмена* и указатель на *функцию распределения*. Функция распределения для каждого входного кортежа вычисляет логический номер процессорного модуля, на котором данный кортеж должен быть обработан. Параметр «порт обмена» позволяет включать в дерево запроса произвольное количество операторов **exchange**. Для каждого оператора указывается свой уникальный порт обмена.

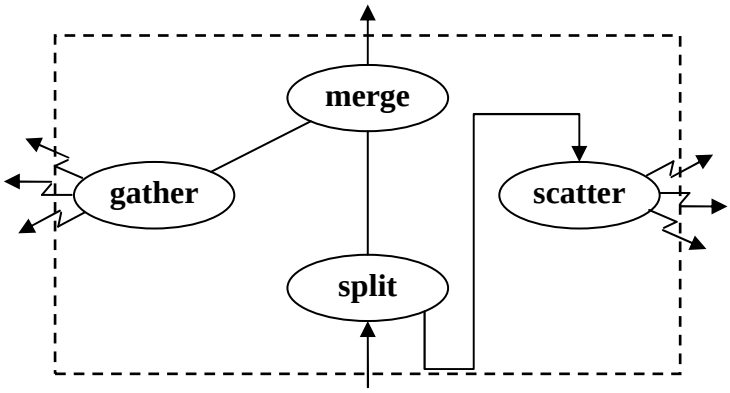

Рис. 5 . Структура оператора обмена **exchange**.

Структура оператора **exchange** показана на рис. 5. Оператор **exchange** является составным оператором и включает в себя четыре оператора: **gather**, **scatter**, **split** и **merge**. Оператор **split** – конъюнктивный бинарный оператор, который осуществляет разбиение поступающих кортежей на две группы: *свои* и *чужие*. Свои кортежи – это кортежи, которые должны быть обработаны данным процессорным модулем. Эти кортежи передаются выше по дереву запроса. Чужие кортежи передаются правому сыну, в качестве которого выступает оператор **scatter**. Нульарный оператор **scatter** получает кортежи от оператора **split** и передает их на соответствующие процессорные модули, используя заданный номер порта обмена. Нульарный оператор **gather** периодически выполняет чтение кортежей из указанного порта и передает их выше по дереву запроса. Оператор **merge** является бинарным оператором, который получает кортежи от своих сыновей и передает дальше по дереву запроса.

Отличительной особенностью оператора **exchange** является то, что он может быть помещен в любом месте дерева запроса, не оказывая при этом никакого влияния на соседние операторы. Оператор **exchange** не участвует в обменах данными на логическом уровне представления запроса и поэтому на данном уровне он выглядит как пустой оператор. Однако на физическом уровне оператор **exchange** выполняет функцию, которую не может выполнить никакой другой оператор, и которая заключается в

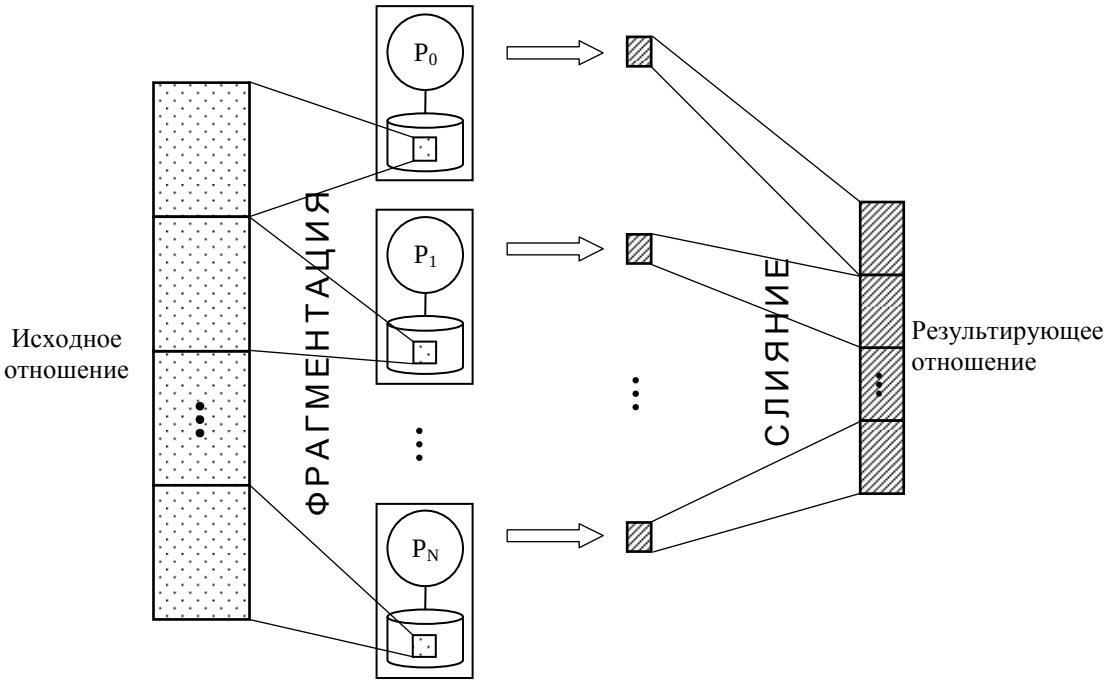

Рис. 6. Фрагментный параллелизм.

перераспределении данных между процессорными узлами при параллельном выполнении реляционных операций. Таким образом, оператор обмена **exchange** инкапсулирует в себе все механизмы, необходимые для реализации фрагментного параллелизма и обеспечивает упрощение реализации параллельных алгоритмов.

1.5.3. Параллельная обработка запроса

Основой параллельной обработки запросов является *фрагментный параллелизм* (см. рис. 6). Данная форма параллелизма предполагает *фрагментацию* отношения, являющегося аргументом реляционной операции, по дискам многопроцессорной системы. Способ фрагментации определяется *функцией фрагментации* ψ, которая для каждого кортежа отношения вычисляет номер процессорного узла, на котором должен быть размещен этот кортеж. В простейшем случае запрос параллельно выполняется на всех процессорных узлах в виде набора *параллельных агентов* [61], каждый из которых обрабатывает отдельный фрагмент отношения на выделенном ему процессорном узле. Полученные агентами результаты *сливаются* в

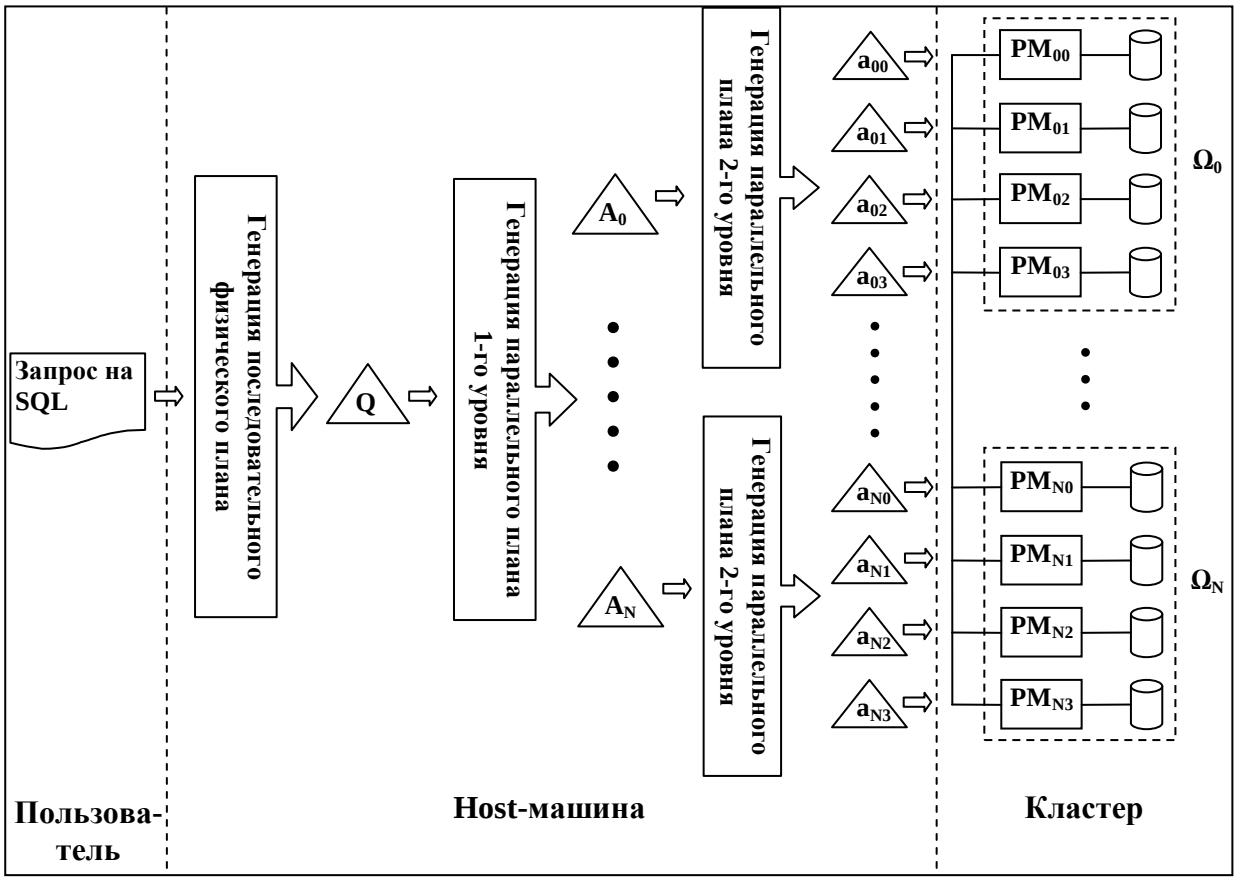

Рис. 7. Схема обработки запроса в параллельной системе баз данных с двухуровневой иерархической архитектурой. Q – последовательный физический план, A_i – агент 1-го уровня, a_{ij} –- агент 2-го уровня, PM_{ij} – процессорный модуль, Ω_i – вычислительный узел.

результирующее отношение. Несмотря на то, что каждый параллельный агент в процессе обработки запроса независимо обрабатывает свой фрагмент отношения, для получения корректного результата запроса необходимо выполнять пересылки кортежей. Для организации таких пересылок в соответствующие места дерева плана запроса вставляется оператор **exchange**.

Опишем общую схему организации параллельной обработки запросов в прототипе иерархической СУБД «Омега». В данном случае будем полагать, что вычислительная система представляет собой кластер, который имеет двухуровневую иерархическую архитектуру и состоит из N вычислительных узлов, каждый из которых содержит четыре процессорных модуля (см. рис. 7). Будем считать, что каждое отношение базы данных, задейство-

ванное в обработке запроса, фрагментировано по всем процессорным модулям вычислительной системы. В соответствии с данной схемой, обработка запроса состоит из трех этапов.

На первом этапе запрос на языке SQL передается пользователем на выделенную host-машину, где транслируется в некоторый *последовательный физический план* [8]. Данный последовательный физический план преобразуется в *параллельный план* 1-го уровня, представляющий собой совокупность *агентов 1-го уровня*. Каждый агент 1-го уровня, кроме реляционных операций, может содержать оператор **exchange**, семантика которого описывается в разделе 1.5.2. Параллельный план 1-го уровня распараллеливает запроса с точностью до вычислительного узла.

На следующем этапе параллельный план 1-го уровня преобразуется в *параллельный план 2-го уровня*. При этом каждый агент 1-го уровня преобразуется в четырех агентов 2-го уровня. Это достигается путем вставки оператора обмена **exchange** в соответствующие места дерева запроса. Параллельный план 2-го уровня задает распараллеливание запроса с точностью до процессорного модуля.

На завершающем этапе агенты 2-го уровня пересылаются с host-машины на соответствующие процессорные модули, где *интерпретируются* исполнителем запросов. Результаты выполнения агентов в пределах одного вычислительного узла объединяются корневым оператором **exchange** на нулевом процессорном модуле, откуда передаются на host-машину. Роль host-машины может играть любой узел вычислительной системы.

ГЛАВА 2. РАЗМЕЩЕНИЕ ДАННЫХ И БАЛАНСИРОВКА ЗАГРУЗКИ

2.1. Фрагментация и сегментация данных

Распределение базы данных в многопроцессорной иерархической системе задается следующим образом [76]. Каждое отношение разбивается на непересекающиеся горизонтальные фрагменты, которые размещаются на различных дисковых модулях. При этом предполагается, что кортежи фрагмента некоторым образом упорядочены, что этот порядок фиксирован для каждого запроса и определяет последовательность считывания кортежей в операции сканирования фрагмента. Будем называть этот порядок *естественным*. На практике естественный порядок может определяться физическим порядком следования кортежей или индексом.

Каждый фрагмент на логическом уровне разбивается на последовательность *сегментов* фиксированной длины. Длина сегмента измеряется в кортежах и является атрибутом фрагмента. Разбиение на сегменты выполняется в соответствии с естественным порядком и всегда начинается с первого кортежа. В соответствии с этим последний сегмент фрагмента может оказаться неполным.

Количество сегментов фрагмента F обозначается как $\mathrm{S}(F)$ и может быть вычислено по формуле

$$\mathrm{S}(F) = \left\lceil \frac{\mathrm{T}(F)}{\mathrm{L}(F)} \right\rceil. \qquad (1)$$

Здесь $\mathrm{T}(F)$ обозначает количество кортежей во фрагменте F, $\mathrm{L}(F)$ – длину сегмента для фрагмента F.

2.2. Репликация данных

2.2.1. Алгоритм построения реплики

Пусть фрагмент F_0 располагается на дисковом модуле $d_0 \in \mathfrak{D}(T)$ многопроцессорной иерархической системы *T*. Полагаем, что на каждом дисковом модуле $d_i \in \mathfrak{D}(T) \quad (i > 0)$ располагается *частичная реплика* F_i, включающая в себя некоторое подмножество (возможно пустое) кортежей фрагмента F_0.

Наименьшей единицей репликации данных является сегмент. Длина сегмента реплики всегда совпадает с длиной сегмента реплицируемого фрагмента:

$$\mathrm{L}(F_i) = \mathrm{L}(F_0), \qquad \forall d_i \in \mathfrak{D}(T).$$

Размер реплики F_i задается *коэффициентом репликации*

$$\rho_i \in \mathbb{R}, \quad 0 \le \rho_i \le 1,$$

являющимся атрибутом реплики F_i, и вычисляется по следующей формуле

$$\mathrm{T}(F_i) = \mathrm{T}(F_0) - \left\lceil (1 - \rho_i) \cdot \mathrm{S}(F_0) \right\rceil \cdot \mathrm{L}(F_0). \qquad (2)$$

Естественный порядок кортежей реплики F_i определяется естественным порядком кортежей фрагмента F_0. При этом *номер* N *первого кортежа* реплики F_i вычисляется по формуле

$$\mathrm{N}(F_i) = \mathrm{T}(F) - \mathrm{T}(F_i) + 1.$$

Для пустой реплики F_i будем иметь $\mathrm{N}(F_i) = \mathrm{T}(F_0) + 1$, что соответствует признаку «конец файла».

Описанный механизм репликации данных позволяет использовать в многопроцессорных иерархиях простой и эффективный метод балансировки загрузки, описываемый в разделе 2.3.

2.2.2. Метод частичного зеркалирования

Пусть имеется симметричное *DM*-дерево T высоты $H = h(T) > 1$. Пусть задана *функция репликации* $r(l)$, сопоставляющая каждому уровню $l < H$ дерева T некоторый коэффициент репликации. Полагаем, что всегда $r(H-1) = 1$. Это мотивируется тем, что уровень иерархии $H-1$ включает в себя поддеревья высоты 1, которым соответствуют SMP-системы. В SMP-системе все диски в равной мере доступны любому процессору, поэтому нет необходимости в физической репликации данных. На логическом уровне балансировка загрузки осуществляется путем сегментирования исходного фрагмента, то есть сам фрагмент играет роль своей реплики.

Пусть фрагмент F_0 располагается на диске $d_0 \in \mathfrak{D}(T)$. Будем использовать следующий метод для построения реплики F_i на диске $d_i \in \mathfrak{D}(T) \quad (i > 0)$, называемый *методом частичного зеркалирования*. Построим последовательность поддеревьев дерева T

$$\{M_0, M_1, \ldots, M_{H-2}\}, \tag{3}$$

обладающую следующими свойствами:

$$\begin{cases} l(M_j) = j \\ d_0 \in \mathfrak{D}(M_j) \end{cases} \tag{4}$$

для всех $0 \le j \le H-2$. Здесь $l(M_j)$ обозначает уровень поддерева M_j. Для любого симметричного дерева T существует только одна такая последовательность. Действительно, предположим противное, то есть, что существуют две различные последовательности $\{M_0, \ldots, M_{H-2}\}$ и $\{M'_0, \ldots, M'_{H-2}\}$, такие, что:

$$\begin{cases} l(M_i) = i \\ d_0 \in \mathfrak{D}(M_i) \end{cases} \text{ и } \begin{cases} l(M'_i) = i \\ d_0 \in \mathfrak{D}(M'_i) \end{cases}.$$

Тогда в дереве T существуют два различных поддерева одного уровня M_i и M'_i, имеющие общий лист d_0. Следовательно, существует два различных

ориентированных пути от d_0 к корню дерева *T*, что противоречит определению ориентированного дерева.

Найдем наибольший индекс $j \geq 1$ такой, что

$$\{d_0, d_i\} \subset \mathfrak{D}(M_j).$$

Коэффициент репликации для реплики F_i вычисляется по формуле

$$\rho_i = r(j). \tag{5}$$

Для формирования реплики F_i на диске d_i используется алгоритм, описанный в пункте 2.2.1 с коэффициентом репликации, определяемым по формуле (5).

Следующая теорема дает оценку для размера реплики в методе частичного зеркалирования.

Теорема 1. Пусть *T* – симметричное *DM*-дерево высоты $H = h(T) > 0$. Пусть фрагмент F_0 располагается на диске $d_0 \in \mathfrak{D}(T)$. Пусть *M* поддерево дерева *T* такое, что $1 \leq l(M) \leq H - 1$ и $d_0 \in \mathfrak{D}(M)$. Пусть M' – произвольное смежное с *M* поддерево дерева *T*. Тогда для любого $d_i \in \mathfrak{D}(M')$ справедлива следующая оценка для размера реплики F_i фрагмента F_0, размещенной на диске d_i:

$$\mathrm{T}(F_i) = r\big(l(M) - 1\big)\mathrm{T}(F_0) + \mathrm{O}\big(\mathrm{L}(F_0)\big),$$

где $\mathrm{L}(F_0)$ – длина сегмента для фрагмента F_0.

Доказательство. В соответствии с формулой (2) имеем

$$\mathrm{T}(F_i) = \mathrm{T}(F_0) - \left\lceil \big(1 - \rho_i\big) \cdot \mathrm{S}(F_0) \right\rceil \cdot \mathrm{L}(F_0).$$

Отсюда получаем

$$\mathrm{T}(F_i) = \mathrm{T}(F_0) - \big(1 - \rho_i\big) \cdot \mathrm{S}(F_0) \cdot \mathrm{L}(F_0) + \mathrm{O}\big(\mathrm{L}(F_0)\big). \tag{6}$$

Так как $d_0 \in \mathfrak{D}(M)$, $d_i \in \mathfrak{D}(M')$ и поддеревья M и M' являются смежными, то минимальное поддерево $\bar{M}$, содержащее диски d_0 и d_i будет

иметь уровень $l(\bar{M}) = l(M) - 1$. Тогда в соответствии с (5) получаем $\rho_i = r(l(M) - 1)$. Подставив это значение в (6), будем иметь

$$\mathrm{T}(F_i) = \mathrm{T}(F_0) - \big(1 - r\big(l(M) - 1\big)\big) \cdot \mathrm{S}(F_0) \cdot \mathrm{L}(F_0) + \mathrm{O}\big(\mathrm{L}(F_0)\big).$$

Подставив вместо $\mathrm{S}(F_0)$ значение из (1), получим

$$\begin{aligned}
\mathrm{T}(F_i) &= \mathrm{T}(F_0) - \big(1 - r\big(l(M) - 1\big)\big) \cdot \left\lceil \frac{\mathrm{T}(F_0)}{\mathrm{L}(F_0)} \right\rceil \cdot \mathrm{L}(F_0) + \mathrm{O}\big(\mathrm{L}(F_0)\big) \\
&= \mathrm{T}(F_0) - \big(1 - r\big(l(M) - 1\big)\big) \cdot \frac{\mathrm{T}(F_0)}{\mathrm{L}(F_0)} \cdot \mathrm{L}(F_0) + \mathrm{O}\big(\mathrm{L}(F_0)\big) \\
&= \mathrm{T}(F_0) - \big(1 - r\big(l(M) - 1\big)\big)\mathrm{T}(F_0) + \mathrm{O}\big(\mathrm{L}(F_0)\big) \\
&= r\big(l(M) - 1\big)\mathrm{T}(F_0) + \mathrm{O}\big(\mathrm{L}(F_0)\big)
\end{aligned}$$

Теорема доказана.

Заметим, что размер сегмента $\mathrm{L}(F_0)$ является параметром репликации и не связан с фрагментацией базы данных. Таким образом, можно считать, что $\mathrm{L}(F_0)$ является константой, значение которой мало относительно общего размера базы данных и им можно пренебречь.

Оценка суммарного размера всех реплик фрагмента может быть получена с помощью следующей теоремы.

Теорема 2. Пусть T – симметричное *DM*-дерево высоты $H \geq 2$. Пусть фрагмент F_0 располагается на диске $d_0 \in \mathfrak{D}(T)$. Обозначим степень уровня l дерева T как δ_l. Обозначим $\mathrm{R}(F_0) = \sum_{i=1}^{|\mathfrak{D}(T)|} \mathrm{T}(F_i)$ – суммарное количество кортежей во всех репликах фрагмента F_0. Тогда

$$\mathrm{R}(F_0) = \mathrm{T}(F_0) \sum_{j=0}^{H-2} r(j)(\delta_j - 1) \prod_{k=j+1}^{H-2} \delta_k + \mathrm{O}\big(\mathrm{L}(F_0)\big). \qquad (7)$$

Доказательство. Доказательство проведем индукцией по высоте *H* дерева *T*.

Пусть $H = 2$. Тогда число дисков в дереве *T* равно δ_0. В соответствии с теоремой 1 каждая реплика будет иметь размер

$$\mathrm{T}_0(F_0) = r(0)\mathrm{T}(F_0) + \mathrm{O}\big(\mathrm{L}(F_0)\big).$$

Следовательно, суммарное количество кортежей во всех репликах фрагмента F_0 имеет следующую оценку

$$\begin{aligned}\mathrm{R}(F_0) &= \big(\mathrm{T}(F_0)r(0) + \mathrm{O}(\mathrm{L}(F_0))\big)\cdot(\delta_0 - 1)\\ &= \mathrm{T}(F_0)r(0)(\delta_0 - 1) + \mathrm{O}\big(\mathrm{L}(F_0)\big)\end{aligned}, \tag{8}$$

что согласуется с формулой (7) при $H = 2$.

Пусть $H > 2$. Тогда дерево *T* содержит δ_0 поддеревьев высоты $H - 1$:

$$M_0, M_1, \ldots, M_{\delta_0 - 1}.$$

Обозначим через $R_j(F_0)$ суммарное количество кортежей во всех репликах фрагмента F_0, расположенных на всех дисках поддерева M_j. Имеем

$$\mathrm{R}(F_0) = \sum_{j=0}^{\delta_0 - 1} \mathrm{R}_j(F_0). \tag{9}$$

Без ограничения общности можно считать, что $d_0 \in \mathfrak{D}(M_0)$. Тогда в силу симметричности дерева *T* из (9) получаем

$$\mathrm{R}(F_0) = \mathrm{R}_0(F_0) + (\delta_0 - 1)\mathrm{R}_1(F_0). \tag{10}$$

В соответствии с теоремой 1 любая реплика F_i, располагающаяся в поддереве M_1, имеет следующий размер

$$\mathrm{T}(F_i) = r(0)\mathrm{T}(F_0) + \mathrm{O}\big(\mathrm{L}(F_0)\big). \tag{11}$$

В силу симметричности дерева *T*, суммарное количество дисков в поддереве M_1 равно $\prod_{k=1}^{H-2}\delta_k$. Учитывая этот факт, из (11) получаем

$$\mathrm{R}_1(F_0) = r(0)\mathrm{T}(F_0)\prod_{k=1}^{H-2}\delta_k + \mathrm{O}\big(\mathrm{L}(F_0)\big). \tag{12}$$

С другой стороны, по предположению индукции имеем

$$\mathrm{R}_0(F_0) = \mathrm{T}(F_0)\sum_{j=1}^{H-2} r(j)(\delta_j - 1)\prod_{k=j+1}^{H-2}\delta_k + \mathrm{O}\big(\mathrm{L}(F_0)\big). \tag{13}$$

Подставляя в (10) значения правых частей из (12) и (13), имеем

$$\mathrm{R}(F_0)=\mathrm{T}(F_0)\sum_{j=1}^{H-2} r(j)\prod_{k=j+1}^{H-2}\delta_k+(\delta_0-1)r\left(0\right)\mathrm{T}\left(F_0\right)\prod_{k=1}^{H-2}\delta_k+\mathrm{O}\left(\mathrm{L}(F_0)\right)$$

$$=\mathrm{T}(F_0)\sum_{j=0}^{H-2} r(j)(\delta_j-1)\prod_{k=j+1}^{H-2}\delta_k+\mathrm{O}\left(\mathrm{L}(F_0)\right)$$

Теорема доказана.

2.2.3. Функция репликации

При выборе функции репликации $r(l)$ целесообразно учитывать коэффициенты трудоемкости узлов *DM*-дерева. Очевидно, что в симметричном *DM*-дереве все вершины уровня l имеют одинаковую трудоемкость $\eta(l)$, которую будем называть *трудоемкостью уровня l*.

Назовем симметричное *DM*-дерево T *регулярным*, если для любых двух уровней l и l' дерева T справедливо

$$l<l' \quad \Rightarrow \quad \eta(l)\geq\eta(l'), \tag{14}$$

то есть, чем выше уровень в иерархии, тем больше его трудоемкость.

Следующая теорема позволяет получить оценку трудоемкости покортежного формирования реплики в регулярном *DM*-дереве.

Теорема 3. Пусть T – регулярное *DM*-дерево высоты $H>0$. Пусть фрагмент F_0 располагается на диске $d_0\in\mathfrak{D}(T)$. Пусть M поддерево дерева T такое, что $1\leq l(M)\leq H-2$ и $d_0\in\mathfrak{D}(M)$. Пусть M' – произвольное смежное с M поддерево дерева T; F_i – реплика фрагмента F_0, размещенная на диске $d_i\in\mathfrak{D}(M')$. Обозначим $\tau(F_i)$ – трудоемкость покортежного формирования реплики F_i при отсутствии помех. Тогда

$$\tau(F_i)=\eta\left(l(M)-1\right)\cdot r\left(l(M)-1\right)\mathrm{T}\left(F_0\right)+\mathrm{O}\left(\eta_0\right),$$

где $\eta_0=\eta(0)$ – коэффициент трудоемкости корня дерева T.

Доказательство. Организуем конвейерную передачу кортежей с диска $d_0\in\mathfrak{D}(M)$ на диск $d_i\in\mathfrak{D}(M')$ в соответствии с моделью операционной среды из [13]. Скорость работы конвейера определяется самым медлен-

ным узлом. Так как M и M' являются смежными поддеревьями, их корневые узлы имеют общего родителя уровня $l(M)-1$, который в соответствии с (14) и будет самым медленным звеном конвейера. Следовательно, трудоемкость передачи одного кортежа при полностью запущенном конвейере равна $\eta\big(l(M)-1\big)$. Отсюда

$$\tau(F_i)=\eta\big(l(M)-1\big)\mathrm{T}(F_i)+\mathrm{O}\Big(\eta\big(l(M)-1\big)\Big). \qquad (15)$$

Здесь $\mathrm{O}\Big(\eta\big(l(M)-1\big)\Big)$ обозначает верхнюю границу для времени, необходимого для полного «разгона» конвейера в предположении, что высота дерева T является константой.

Так как T регулярно, то $\eta\big(l(M)-1\big)\le\eta_0$. На основании этого из (15) получаем

$$\tau(F_i)=\eta\big(l(M)-1\big)\mathrm{T}(F_i)+\mathrm{O}\big(\eta_0\big). \qquad (16)$$

По теореме 1 из формулы (16) получаем

$$\tau(F_i)\ =\eta\big(l(M)-1\big)\cdot r\big(l(M)-1\big)\mathrm{T}\big(F_0\big)+\mathrm{O}\big(\eta_0\big)\mathrm{O}\big(\mathrm{L}(F_0)\big)+\mathrm{O}\big(\eta_0\big). \qquad (17)$$

Мы вправе считать, что длина сегмента не меняется в процессе формирования реплики, то есть $\mathrm{L}(F_0)$ является константой. Тогда из (17) получаем

$$\begin{aligned}\tau(F_i)\ &=\eta\big(l(M)-1\big)\cdot r\big(l(M)-1\big)\mathrm{T}\big(F_0\big)+\mathrm{O}\big(\eta_0\big)+\mathrm{O}\big(\eta_0\big)\\ &=\eta\big(l(M)-1\big)\cdot r\big(l(M)-1\big)\mathrm{T}\big(F_0\big)+\mathrm{O}\big(\eta_0\big)\end{aligned}$$

Теорема доказана.

Оценка трудоемкости покортежного формирования всех реплик фрагмента без учета помех может быть получена с помощью следующей теоремы.

Теорема 4. Пусть T – регулярное *DM*-дерево высоты $H\ge 2$. Пусть фрагмент F_0 располагается на диске $d_0\in\mathfrak{D}(T)$. Обозначим степень уровня l дерева T как δ_l. Обозначим $\tau(F_0)=\sum_{i=1}^{|\mathfrak{D}(T)|}\tau(F_i)$ – суммарная трудоемкость

покортежного формирования всех реплик фрагмента F_0 без учета помех. Тогда

$$\tau(F_0) = \mathrm{T}(F_0)\sum_{j=0}^{H-2}\eta(j)r(j)(\delta_j - 1)\prod_{k=j+1}^{H-2}\delta_k + \mathrm{O}(\eta_0). \qquad (18)$$

Доказательство. Доказательство проведем индукцией по высоте H дерева T.

Пусть $H = 2$. Тогда число дисков в дереве T равно δ_0. В соответствии с теоремой 3 трудоемкость покортежного формирования любой реплики фрагмента F_0 в этом случае имеет следующую оценку:

$$\tau(F_i) = \eta(0)\cdot r(0)\mathrm{T}(F_0) + \mathrm{O}(\eta_0).$$

Следовательно, суммарная трудоемкость покортежного формирования всех реплик фрагмента F_0 без учета помех может быть оценена следующим образом:

$$\tau(F_i) = \eta(0)r(0)\mathrm{T}(F_0)(\delta_0 - 1) + \mathrm{O}(\eta_0),$$

что согласуется с формулой (18) при $H = 2$.

Пусть $H > 2$. Тогда дерево T содержит δ_0 поддеревьев высоты $H-1$:

$$M_0, M_1, \ldots, M_{\delta_0 - 1}.$$

Обозначим через $\tau_j(F_0)$ суммарную трудоемкость формирования без учета помех всех реплик фрагмента F_0, расположенных на всех дисках поддерева M_j. Имеем

$$\tau(F_0) = \sum_{j=0}^{\delta_0 - 1}\tau_j(F_0). \qquad (19)$$

Без ограничения общности можно считать, что $d_0 \in \mathfrak{D}(M_0)$. Тогда в силу симметричности дерева T из (19) получаем

$$\tau(F_0) = \tau_0(F_0) + (\delta_0 - 1)\tau_1(F_0). \qquad (20)$$

В соответствии с теоремой 3 для любой реплики F_i, располагающейся в поддереве M_1, имеем

$$\tau(F_i) = \eta(0) r(0) \mathrm{T}(F_0) + \mathrm{O}(\eta_0). \quad (21)$$

В силу симметричности дерева T, суммарное количество дисков в поддереве M_1 равно $\prod_{k=1}^{H-2} \delta_k$. Учитывая этот факт, из (21) получаем

$$\tau_1(F_0) = \eta(0) r(0) \mathrm{T}(F_0) \prod_{k=1}^{H-2} \delta_k + \mathrm{O}(\eta_0). \quad (22)$$

С другой стороны, по предположению индукции имеем

$$\tau_0(F_0) = \mathrm{T}(F_0) \sum_{j=1}^{H-2} \eta(j) r(j) (\delta_j - 1) \prod_{k=j+1}^{H-2} \delta_k + \mathrm{O}(\eta_0). \quad (23)$$

Подставляя в (20) значения правых частей из (22) и (23), имеем

$$\begin{aligned} \tau(F_0) &= \mathrm{T}(F_0) \sum_{j=1}^{H-2} \eta(j) r(j) (\delta_j - 1) \prod_{k=j+1}^{H-2} \delta_k + (\delta_0 - 1) \eta(0) r(0) \mathrm{T}(F_0) \prod_{k=1}^{H-2} \delta_k + \mathrm{O}(\eta_0) \\ &= \mathrm{T}(F_0) \sum_{j=0}^{H-2} \eta(j) r(j) (\delta_j - 1) \prod_{k=j+1}^{H-2} \delta_k + \mathrm{O}(\eta_0) \end{aligned}$$

Теорема доказана.

Определим рекурсивно *нормальную* функцию репликации $r(l)$ следующим образом:

1. для $l = H-2$: $r(H-2) = \dfrac{1}{\eta(H-2)(\delta_{H-2} - 1)}$;

2. для $0 \le l < H-2$: $r(l) = \dfrac{\eta(l+1)(\delta_{l+1} - 1) r(l+1)}{\eta(l)(\delta_l - 1)\delta_{l+1}}$.

Докажем следующую лемму.

Лемма 1. Рекурсивно нормальную функцию репликации можно представить в виде:

$$r(l) = \frac{1}{\eta(l)(\delta_l - 1) \prod_{j=l+1}^{H-2} \delta_j}, \quad (24)$$

где $\prod_{j=l+1}^{H-2}\delta_j = 1$ при $l = H-2$.

Доказательство.

Пусть $l = H-2$. Тогда из (24) получаем:

$$r(H-2) = \frac{1}{\eta(H-2)(\delta_{H-2}-1)\cdot 1},$$

что согласуется с определением нормальной функции репликации.

Пусть $l < H-2$. Тогда, по определению, нормальная функция репликации будет иметь вид:

$$r(l) = \frac{\eta(l+1)(\delta_{l+1}-1)r(l+1)}{\eta(l)(\delta_l-1)\delta_{l+1}}. \qquad (25)$$

Вместе с этим, по предположению индукции имеем:

$$r(l+1) = \frac{1}{\eta(l+1)(\delta_{l+1}-1)\prod_{j=l+2}^{H-2}\delta_j}. \qquad (26)$$

Подставляя значение функции $r(l+1)$ из (26) в (25) получаем:

$$r(l) = \frac{\eta(l+1)(\delta_{l+1}-1)}{\eta(l+1)(\delta_{l+1}-1)\eta(l)(\delta_l-1)\delta_{l+1}\prod_{j=l+2}^{H-2}\delta_j} = \frac{1}{\eta(l)(\delta_l-1)\prod_{j=l+1}^{H-2}\delta_j},$$

что полностью согласуется с формулой (24).

Лемма доказана.

Справедлива следующая теорема.

Теорема 5. Пусть *T* – регулярное *DM*-дерево высоты $H \geq 2$. Пусть $\mathbb{F}$ – множество фрагментов, составляющих базу данных. Пусть $\mathbb{R}$ – множество всех реплик всех фрагментов из множества $\mathbb{F}$, построенных с использованием нормальной функции репликации. Пусть $\mathrm{T}(\mathbb{F})$ – размер базы данных в кортежах (здесь предполагается, что все кортежи имеют одинаковую длину в байтах), $\tau(\mathbb{R})$ – суммарная трудоемкость покортежного формирования всех реплик без учета помех. Тогда

$$\tau(\mathbb{R}) \approx k\,\mathrm{T}(\mathbb{F}),$$

где k – некоторая константа, не зависящая от $\mathbb{F}$.

Доказательство. Пусть база данных состоит из m фрагментов. Пусть $\tau(\mathbb{R}) = \sum_{i=0}^{m-1} \tau(F_i)$ – суммарная трудоемкость покортежного формирования всех реплик без учета помех. В соответствии с теоремой 4 имеем:

$$\tau(F_i) = \mathrm{T}(F_i) \sum_{j=0}^{H-2} \eta(j) r(j) (\delta_j - 1) \prod_{k=j+1}^{H-2} \delta_k + \mathrm{O}(\eta_0). \qquad (27)$$

Воспользуемся леммой 1. Подставляя в (27) значение нормальной функции репликации из (24) получаем:

$$\begin{aligned}\tau(F_i) &= \mathrm{T}(F_i) \sum_{j=0}^{H-2} \eta(j) r(j) (\delta_j - 1) \prod_{k=j+1}^{H-2} \delta_k + \mathrm{O}(\eta_0) = \\ &= \mathrm{T}(F_i) \sum_{j=0}^{H-2} \frac{\eta(j)(\delta_j - 1)}{\eta(j)(\delta_j - 1) \prod_{i=j+1}^{H-2} \delta_i} \prod_{k=j+1}^{H-2} \delta_k + \mathrm{O}(\eta_0) = \mathrm{T}(F_i) \sum_{j=0}^{H-2} 1 + \mathrm{O}(\eta_0) = . \\ &= (H-2)\mathrm{T}(F_i) + \mathrm{O}(\eta_0)\end{aligned}$$

Таким образом:

$$\begin{aligned}\tau(\mathbb{R}) &= \sum_{i=0}^{m-1} \left[(H-2)\mathrm{T}(F_i) + \mathrm{O}(\eta_0) \right] = (H-2) \cdot \sum_{i=0}^{m-1} \mathrm{T}(F_i) + m \cdot \mathrm{O}(\eta_0) = \\ &= (H-2) \cdot \mathrm{T}(\mathbb{F}) + m \cdot \mathrm{O}(\eta_0) \approx k\,\mathrm{T}(\mathbb{F})\end{aligned},$$

где k не зависит от $\mathbb{F}$.

Теорема доказана.

Данная теорема показывает, что при использовании нормальной функции репликации трудоемкость обновления реплик в регулярной многопроцессорной иерархической системе пропорциональна размеру обновляемой части базы данных при условии, что соединительная сеть обладает достаточной пропускной способностью.

2.3. Метод балансировки загрузки

2.3.1. Схема работы параллельного агента

Пусть задан некоторый запрос $\mathbb{Q}$, имеющий n входных отношений. Пусть $\mathfrak{Q}$ – параллельный план [7] запроса $\mathbb{Q}$. Каждый агент $Q \in \mathfrak{Q}$ имеет n

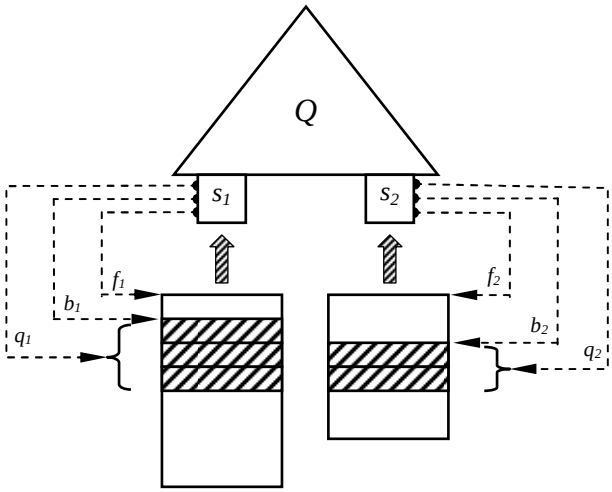

Рис. 8. Параллельный агент с двумя входными потоками.

входных потоков $s_1,\ldots,s_n$. Каждый поток s_i $(i=1,\ldots,n)$ задается четырьмя параметрами:

1) f_i – указатель на фрагмент отношения;
2) q_i – количество сегментов в отрезке, подлежащем обработке;
3) b_i – номер первого сегмента в обрабатываемом отрезке;
4) a_i – индикатор балансировки: 1 – балансировка допускается, 0 – балансировка не допускается.

На рис. 8 изображен пример параллельного агента с двумя входными потоками.

Параллельный агент Q может находиться в одном из двух состояний: *активном* и *пассивном*. В *активном* состоянии Q последовательно считывает и обрабатывает кортежи из всех входных потоков. При этом в ходе обработки динамически изменяются значения параметров q_i и b_i для всех $i=1,\ldots,n$. В *пассивном* состоянии Q не выполняет никаких действий. На начальном этапе выполнения запроса выполняется инициализация агента, в результате которой происходит определение параметров всех входных потоков. Затем агент переводится в активное состояние и начинает обработку фрагментов, ассоциированных с его входными потоками. В каждом фраг-

менте обрабатываются только те сегменты, которые входят в отрезок, определяемый параметрами потока, ассоциированного с данным фрагментом. После того, как все назначенные сегменты во всех входных потоках обработаны, агент переходит в пассивное состояние.

2.3.2. Алгоритм балансировки загрузки

При выполнении параллельного плана запроса одни агенты могут завершить свою работу и находиться в пассивном состоянии в то время, как другие агенты будут продолжать обработку назначенных им отрезков. Тем самым возникает *ситуация перекоса* [82]. Нами предлагается следующий *алгоритм балансировки загрузки*, использующий репликацию данных [12].

Пусть возникла ситуация, когда параллельный агент $\bar{Q} \in \mathfrak{Q}$ закончил обработку назначенных ему сегментов во всех входных потоках и перешел в пассивное состояние, в то время как агент $\tilde{Q} \in \mathfrak{Q}$ все еще продолжает обработку своей порции данных и необходимо произвести балансировку загрузки. Будем называть простаивающий агент $\bar{Q}$ *лидером*, а перегруженный агент $\tilde{Q}$ – *аутсайдером*. В этой ситуации будет выполняться процедура балансировки загрузки между лидером $\bar{Q}$ и аутсайдером $\tilde{Q}$, которая заключается в передаче части необработанных сегментов от агента $\tilde{Q}$ агенту $\bar{Q}$. Схема алгоритма балансировки изображена на рис. 9 (использован Си-подобный псевдокод). При балансировке загрузки используется внешняя по отношению к этой процедуре *функция балансировки* Delta, которая вычисляет количество сегментов соответствующего входного потока, *передаваемых* от аутсайдера $\tilde{Q}$ лидеру $\bar{Q}$.

Для эффективного использования описанного алгоритма балансировки загрузки необходимо решить следующие *задачи*.

1. При наличии простаивающих агентов-лидеров, необходимо выбрать некоторый агент-аутсайдер, который будет являться объектом балансировки. Способ выбора агента-аутсайдера будем называть *стратегией выбора аутсайдера*.
2. Необходимо решить, какое количество необработанных сегментов данных необходимо передать от аутсайдера лидеру. Функцию, вычисляющую это число, будем называть *функцией балансировки*.

```
/* Процедура балансировки загрузки между параллельными
агентами Q̄ (лидер) и Q̃ (аутсайдер). */
ū =Node(Q̄); // указатель на узел агента Q̄.
pause Q̃; // переводим Q̃ в пассивное состояние.
for(i=1;i<=n;i++){
  if(Q̃.s[i].a==1){
     f̃_i = Q̃.s[i].f ; // фрагмент, обрабатываемый агентом Q̃.
     r̄_i = Re(f̃_i, ū); // реплика фрагмента f̃_i на узле ū.
     Δ_i = Delta(Q̃.s[i]); //   количество передаваемых сегментов.
     Q̃.s[i].q−= Δ_i ;
     Q̄.s[i].f = r̄_i ;
     Q̄.s[i].b = Q̃.s[i].b + Q̃.s[i].q ;
     Q̄.s[i].q = Δ_i ;
  }else
    print("Балансировка не разрешена");
};
activate Q̃; // переводим Q̃ в активное состояние
activate Q̄; // переводим Q̄ в активное состояние
```

Рис. 9. Алгоритм балансировки загрузки двух параллельных агентов.

2.3.3. Стратегия выбора аутсайдера

В этом разделе определяется стратегия выбора аутсайдера, называемая *оптимистической*. При этом будем опираться на метод частичного зеркалирования, описанный в разделе 2.2.2.

Рассмотрим иерархическую многопроцессорную систему со структурой в виде симметричного *DM*-дерева T. Пусть $\mathfrak{Q}$ – параллельный план запроса $\mathbb{Q}$, Ψ – множество узлов *DM*-дерева, на которых осуществляется выполнение параллельного плана $\mathfrak{Q}$. Пусть в процессе обработки запроса в некоторый момент времени агент-лидер $\bar{Q} \in \mathfrak{Q}$, расположенный на узле $\bar{\psi} \in \Psi$, закончил свою работу и перешел в пассивное состояние. Необходимо из множества агентов параллельного плана $\mathfrak{Q}$ выбрать некоторый агент-аутсайдер $\tilde{Q} \in \mathfrak{Q} (\tilde{Q} \neq \bar{Q})$, которому будет *помогать* агент-лидер $\bar{Q}$. Будем предполагать, что агент $\tilde{Q}$ располагается на узле $\tilde{\psi} \in \Psi$, и что $\tilde{\psi} \neq \bar{\psi}$. Обозначим через $\tilde{\bar{M}}$ минимальное поддерево дерева T, содержащее узлы $\bar{\psi}$ и $\tilde{\psi}$.

Для выбора агента-аутсайдера используется механизм рейтингов. Каждому агенту параллельного плана в процессе балансировки загрузки присваивается рейтинг, задаваемый вещественным числом. В качестве аутсайдера всегда выбирается агент, имеющий максимальный положительный рейтинг. Если таковые агенты отсутствуют, то освободившийся агент $\bar{Q}$ просто завершает свою работу. Если сразу несколько агентов имеют максимальный положительный рейтинг, то в качестве аутсайдера из их числа выбирается тот, к которому дольше всего не применялась процедура балансировки загрузки.

Для вычисления рейтинга оптимистическая стратегия использует *рейтинговую функцию* $\gamma : \mathfrak{Q} \to \mathbb{R}$ следующего вида:

$$\gamma\left(\tilde{Q}\right) = \tilde{a}_i \operatorname{sgn}\left(\max_{1 \le i \le n}\left(\tilde{q}_i\right) - B\right) \lambda r\left(l(\tilde{\bar{M}})\right) \sum_{i=1}^{n} \tilde{q}_i .$$

Здесь λ – некоторый положительный весовой коэффициент, регулирующий влияние коэффициента репликации на величину рейтинга; B – целое неотрицательное число, задающее нижнюю границу количества сегмен-

тов, которое можно передавать при балансировке нагрузки. Напомним, что $l(M)$ обозначает уровень поддерева M в дереве T (см. 48).

2.3.4. Функция балансировки

Функция балансировки Δ для каждого потока $\tilde{s}_i$ агента-аутсайдера $\tilde{Q}$ определяет количество сегментов, передаваемых агенту-лидеру $\bar{Q}$ на обработку. В простейшем случае можно положить

$$\Delta(\tilde{s}_i) = \min\left(\lceil \tilde{q}_i/2 \rceil, r\left(l(\tilde{\bar{M}})\right)\mathrm{S}\left(\tilde{f}_i\right)\right).$$

Функция $S(\tilde{f}_i)$, введенная в разделе 2.1, вычисляет количество сегментов фрагмента $\tilde{f}_i$. Таким образом, функция Δ передает от $\tilde{Q}$ к $\bar{Q}$ половину необработанных сегментов, если только реплика фрагмента $\tilde{f}_i$ на узле агента $\tilde{Q}$ не меньше этой величины. В противном случае передается столько сегментов, сколько содержится в соответствующей реплике фрагмента $\tilde{f}_i$.

ГЛАВА 3. ИЕРАРХИЧЕСКАЯ СУБД «ОМЕГА»

На основе описанного выше метода частичного зеркалирования была разработана иерархическая СУБД «Омега» для использования в многопроцессорных системах с иерархической архитектурой. Были исследованы основные требования к параллельным СУБД, разработана модель вариантов использования СУБД для многопроцессорных систем с иерархической архитектурой [11]. На основе данной модели была выполнена реализация иерархической СУБД «Омега» использующая алгоритм балансировки загрузки, описанный в разделе 2.3.2.

3.1. Модель вариантов использования СУБД «Омега»

В данном разделе описаны спецификации иерархической СУБД, ориентированной на использование в многопроцессорных иерархиях. Спецификация данной иерархической СУБД разработана с учетом распределенного характера системы, неоднородности вычислительных узлов и коммуникационной сети. СУБД проектируется таким образом, чтобы изолировать пользователя от сложной структуры и технологий, лежащих в основе вычислительной системы. Требования к СУБД задаются при помощи модели вариантов использования [98], построенной на основе языка моделирования UML версии 2.0 [101].

3.1.1. Общие требования к иерархической СУБД

Иерархическая система управления базами данных представляет собой программную систему, которая предназначена для параллельной обработки запросов в многопроцессорных иерархиях.

Иерархическая СУБД поддерживает фрагментный параллелизм и имеет механизм балансировки загрузки, основанный на стратегии репликации, называемый методом частичного зеркалирования (см. раздел 2.2). Заложенный в СУБД параллелизм обработки запросов прозрачен для пользователя.

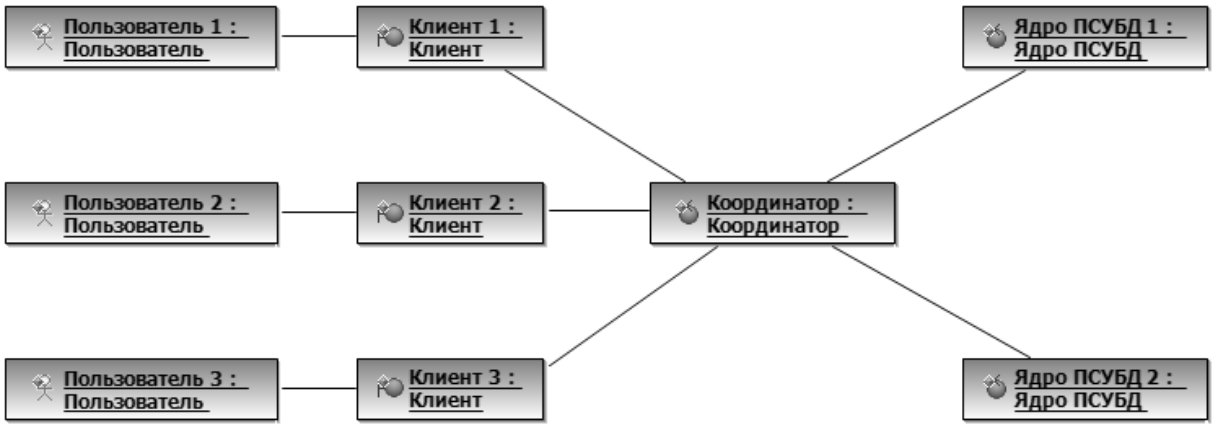

Рис. 10. Диаграмма объектов для случая с тремя пользователями и двумя ядрами СУБД.

Иерархическая СУБД поддерживает многопользовательский режим работы, т.е. одновременно система может обрабатывать запросы нескольких пользователей. Для организации приема запросов от пользователей в данной СУБД использован реляционный язык запросов RQL (см. раздел 3.2.1). Запрос, записанный на языке RQL, передается СУБД в виде текстового файла. После окончания обработки запроса СУБД передает пользователю *результирующую таблицу* и *лог-файл*.

Результирующая таблица представляется в виде текстового файла в формате CSV (Comma Separated Value) [94]. Если результатом запроса является пустая таблица, то текстовый файл с результирующей таблицей будет пустым.

Лог-файл передается пользователю в виде текстового файла в формате CSV. Лог-файл содержит диагностические сообщения о ходе выполнения запроса. Структура и семантика сообщений лог-файла определены в разделе 3.2.2.

3.1.2. Структура иерархической СУБД

Иерархическая СУБД состоит из объектов четырех классов*: Пользователь, Клиент, Координатор* и *Ядро СУБД*. Пример, демонстрирующий статическую структуру иерархической СУБД, показан на рис. 10.

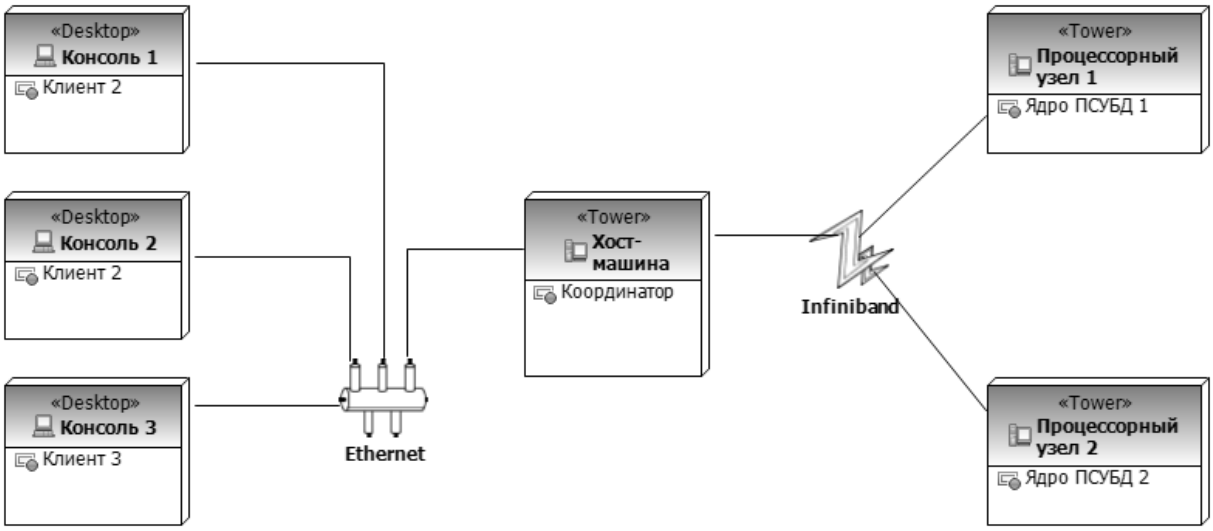

Рис. 11. Диаграмма размещения подсистем иерархической СУБД для случая с тремя пользователями и двумя ядрами СУБД.

Пользователь представляет собой человека, взаимодействующего с иерархической СУБД посредством Клиента, однако Клиент допускает вызов в формате командной строки и может быть использован программными системами, являющимися внешними по отношению к данной СУБД.

Клиент представляет собой подсистему иерархической СУБД, реализующую интерфейс Пользователя. Для каждого Пользователя иерархической СУБД порождается отдельный экземпляр Клиента. Поступающие от пользователя запросы Клиент передает Координатору, который обеспечивает совместную обработку запроса Ядрами СУБД.

Основной задачей *Координатора* является доставка запроса от Клиента на Ядра СУБД. Координатор принимает запрос от каждого обратившегося к нему Клиента и передает его Ядрам СУБД, которые будут выполнять обработку данного запроса. В рамках рассматриваемого примера совместная обработка запросов осуществляется двумя Ядрами СУБД и управляется одним Координатором. В целях повышения производительности в распределенной вычислительной системе может быть запущено несколько Координаторов. В таком случае Клиент может обращаться с запросами к любому доступному Координатору. Необходимость введения

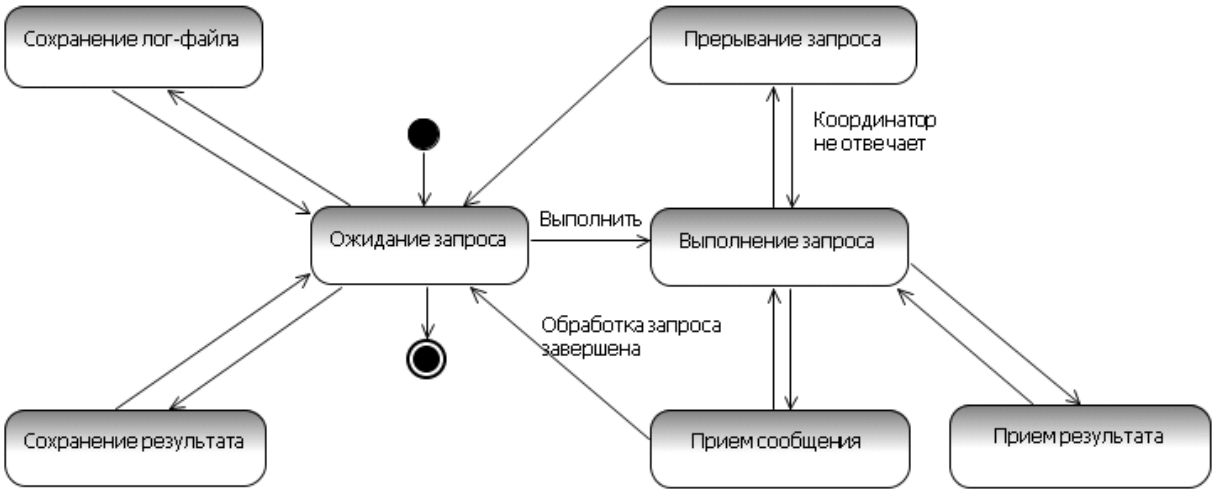

Рис. 12. Диаграмма состояний подсистемы «Клиент».

дополнительного Координатора может возникать, например, при увеличении количества Клиентов или объемов поступающих от Ядер СУБД результатов.

Ядро СУБД непосредственно выполняет обработку запроса и передает результат тому Координатору, от которого поступил запрос.

На рис. 11 показана типовая конфигурация размещения подсистем иерархической СУБД. Клиент запускается на рабочей станции пользователя (в данной конфигурации на консоли). Если с СУБД на одной рабочей станции взаимодействует несколько пользователей, то для каждого пользователя на данной рабочей станции запускается отдельный экземпляр Клиента.

Координатор размещается в распределенной вычислительной системе на выделенном узле (хост-машине). Координатор взаимодействует с Клиентами через локальную коммуникационную сеть (в данном случае Ethernet). Координатор взаимодействует с ядрами через коммуникационную сеть (в данном случае Infiniband). Ядро СУБД размещается на выделенном ему процессорном узле. Процессорные узлы системы соединяются между собой коммуникационной сетью (Infiniband).

Клиент предоставляет пользователю интерфейс доступа к СУБД: обеспечивает передачу запросов СУБД, мониторинг процесса обработки запроса и доступ к результату его выполнения. В каждый момент време-

ни Клиент может обрабатывать только один запрос пользователя. Клиент может использоваться в диалоговом и пакетном режимах. При использовании Клиента в диалоговом режиме пользователю предоставляется диалоговое окно, основными элементами которого являются строка ввода запроса, область просмотра результата и область просмотра диагностических сообщений.

В пакетном режиме работы Клиенту передается имя файла с запросом и имя файла, в который будет записана результирующая таблица.

На рис. 12 показаны возможные состояния Клиента. Рабочий цикл Клиента начинается с состояния «Ожидание запроса», в котором Клиент ожидает команд от Пользователя. Из данного состояния Клиент может перейти в состояние «Выполнение запроса», «Сохранение результата» или «Сохранение лог-файла». Пользователь может также корректно завершить Клиента, вызвав функцию «Завершить работу».

В состоянии «Выполнение запроса» Клиент ожидает результатов обработки запроса от Координатора. При поступлении от Координатора результирующей таблицы или диагностического сообщения, Клиент переходит в состояние «Прием результата» или «Прием сообщения» соответственно. Пользователь может спровоцировать прерывание обработки запроса, вызвав функцию Клиента «Прервать». При этом, Клиент переходит в состояние «Прерывание запроса».

В состоянии «Сохранение результата» Клиент помещает результирующую таблицу в заданный пользователем файл. При нормальном завершении операции Клиент переходит в состояние «Ожидание запроса».

В состоянии «Сохранение лог-файла» Клиент помещает полученные в ходе обработки запроса диагностические сообщения в указанный пользователем лог-файл. При нормальном завершении операции Клиент переходит в состояние «Ожидание запроса».

В состоянии «Прием сообщения» Клиент принимает диагностическое сообщение от Координатора Если полученное сообщение уведомляет об окончании обработки запроса, то Клиент переходит в состояние «Ожидание запроса». В противном случае он переходит в состояние «Выполнение запроса».

В состоянии «Прием результата» Клиент принимает результирующую таблицу от Координатора. При нормальном завершении операции Клиент переходит в состояние «Ожидание запроса».

В состоянии «Прерывание запроса» Клиент передает Координатору сообщение о прерывании обработки запроса. Если сообщение удается передать Координатору, прерывание запроса заканчивается успешно и Клиент переходит в состояние «Ожидание запроса». Если Координатор оказывается недоступен, Клиент переходит в состояние «Выполнение запроса».

Подсистема «Координатор» организует параллельное выполнение запроса на множестве Ядер иерархической СУБД. Координатор может взаимодействовать с несколькими Клиентами. Во время работы Координатор ведет лог-файл, в который заносятся диагностические сообщения о работе Координатора.

Подсистема «Ядро СУБД» работает на отдельном SMP-узле и выполняет обработку поступающих запросов в асинхронном режиме. На одном SMP-узле может работать только одно Ядро СУБД. Ядро СУБД поддерживает межтранзакционный параллелизм, т.е. одновременно может обрабатывать несколько запросов.

3.1.3. Варианты использования системы «Омега»

В данном разделе приведены варианты использования подсистем «Клиент» и «Координатор». Вариант использования описывается в соответствии со следующей дисциплиной. Во-первых, приводится краткое

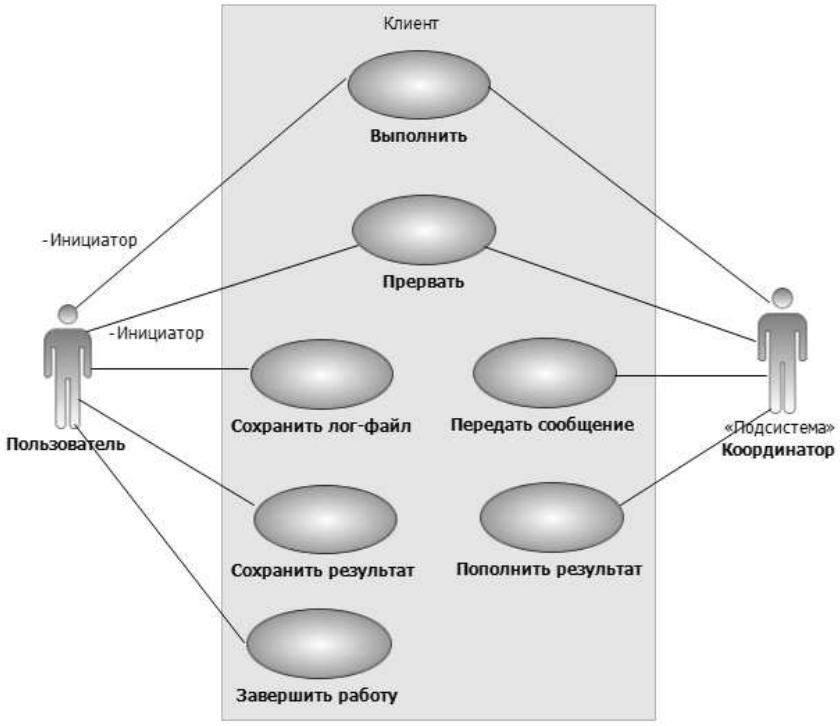

Рис 13. Диаграмма вариантов использования подсистемы «Клиент».

описание варианта использования и предварительные условия, необходимые для начала его выполнения. Во-вторых, приводится описание основных шагов, составляющих поток событий варианта использования. В-третьих, обсуждаются альтернативные потоки событий и приводятся дополнительные требования.

На рис. 13 приведены варианты использования подсистемы Клиент. Подсистема «Клиент» предусматривает диалоговый и пакетный режимы работы. Далее приводится детальное описание каждого варианта использования.

Вариант использования «Выполнить» передает запрос от Пользователя к Координатору. Вариант использования начинается, когда пользователь указывает путь к текстовому файлу с запросом и активирует функцию «Выполнить». При этом Клиент должен находиться в состоянии ожидания запроса и текстовый файл с запросом должен существовать на узле Клиента.

Поток событий состоит из следующих шагов. На первом шаге Клиент переходит в состояние «Выполнение запроса» и подготавливает систему к

обработке запроса: очищает область просмотра результирующей таблицы, очищает область просмотра сообщений, формирует пустой файл результата и пустой лог-файл. На втором шаге Клиент передает текстовый файл с запросом Координатору. Вариант использования завершается.

Вариант использования предусматривает один альтернативный поток событий. Если попытка Клиента передать запрос Координатору завершается неудачей, то Клиент выводит в области просмотра соответствующее сообщение об ошибке, записывает его в лог-файл и переходит в состояние «Ожидание запроса». Вариант использования завершается.

Вариант использования «Прервать» осуществляет прерывание выполнения запроса (досрочное завершение) по требованию Пользователя. Вариант использования начинается, когда Пользователь вызывает функцию «Прервать». При этом Клиент должен находиться в состоянии «Выполнение запроса».

Поток событий состоит из следующих шагов. На первом шаге Клиент переходит в состояние вывода сообщения и посылает Координатору сообщение «Прервать». Если передача сообщения завершается успешно, то на втором шаге Клиент формирует сообщение о досрочном прерывании запроса, выводит его в области просмотра сообщений и записывает в лог-файл. На последнем шаге Клиент переходит в состояние «Ожидание запроса». Вариант использования завершается.

Если попытка Клиента передать сообщение Координатору заканчивается неудачей, то он формирует сообщение об ошибке, выводит его в области просмотра и записывает в лог-файл. После этого Клиент переходит в состояние «Выполнение запроса». Вариант использования завершается.

Вариант использования «Передать сообщение» уведомляет Клиента о возникновении некоторого существенного события в ходе выполнения запроса. Вариант использования начинается, когда Координатор вызывает

функцию «Передать сообщение». При этом Клиент должен находиться в состоянии «Выполнение запроса».

Поток событий состоит из следующих шагов. На первом шаге Клиент переходит в состояние «Прием сообщения» и получает от Координатора диагностическое сообщение. На втором шаге Клиент выводит полученное сообщение в области просмотра сообщений и записывает его в лог-файл. На третьем шаге Клиент переходит в состояние «Выполнение запроса». Вариант использования завершается.

Вариант использования предусматривает альтернативный поток событий. Если Клиент на первом шаге получает от Координатора сообщение с кодом фатальной ошибки или кодом, соответствующим нормальному завершению, то выполняется следующая последовательность шагов. На первом шаге Клиент выводит полученное сообщение в области просмотра сообщений и записывает его в лог-файл. На втором шаге Клиент переходит в состояние «Ожидание запроса», после чего вариант использования завершается.

Вариант использования «Пополнить результат» передает очередную часть результирующей таблицы от Координатора Клиенту. Вариант использования начинается, когда Координатор вызывает функцию «Пополнить результат».

Поток событий состоит из следующих шагов. На первом шаге Клиент переходит в состояние «Прием результата» и принимает от Координатора очередную часть результирующей таблицы. На втором шаге Клиент выводит полученную часть результирующей таблицы в области просмотра результата и записывает ее в результирующий файл. На третьем шаге Клиент переходит в состояние выполнения запроса. Вариант использования завершается.

Вариант использования «Сохранить результат» помещает результат запроса в текстовый файл в формате CSV. Вариант использования начина-

ется, когда Пользователь вызывает функцию «Сохранить результат». При этом Клиент должен находиться в состоянии ожидания нового запроса.

Поток событий состоит из следующих шагов. На первом шаге Клиент переходит в состояние «Сохранение результата», получает от Пользователя имя файла, в который будет помещена результирующая таблица. На втором шаге Клиент сохраняет результирующую таблицу в текстовом файле, указанном пользователем, в формате CSV. На третьем шаге Клиент переходит в состояние «Ожидание запроса». Вариант использования завершается.

Полученная Клиентом результирующая таблица хранится во временном файле до тех пор, пока Пользователь не запустит выполнение нового запроса.

Вариант использования «Сохранить лог-файл» помещает лог-файл запроса в текстовый файл в формате CSV. Вариант использования начинается, когда Пользователь вызывает функцию «Сохранить лог-файл». При этом Клиент должен находиться в состоянии «Ожидание запроса».

Поток событий состоит из следующих шагов. На первом шаге Клиент переходит в состояние «Сохранение лог-файла», запрашивает у Пользователя путь и имя файла для сохранения лог-файла. На втором шаге Клиент сохраняет лог-файл в текстовом файле, указанном пользователем. На третьем шаге Клиент переходит в состояние «Ожидание запроса». Вариант использования завершается.

Полученные Клиентом от Координатора сообщения хранятся во временном файле до тех пор, пока Пользователь не запустит выполнение нового запроса.

Вариант использования «Завершить работу» завершает сеанс работы Пользователя с СУБД. Вариант использования начинается, когда Пользователь вызывает функцию «Завершить работу». При этом Клиент должен находиться в состоянии «Ожидание запроса».

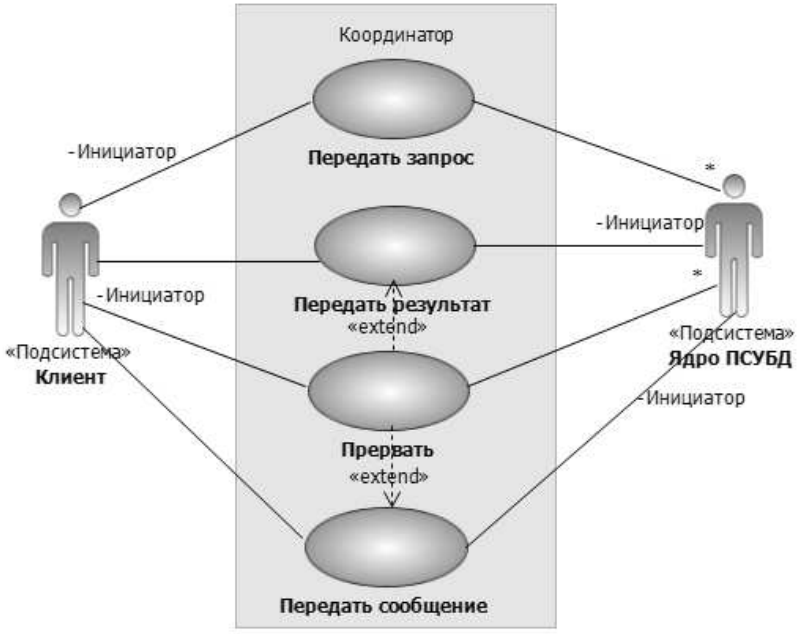

Рис 14. Диаграмма вариантов использования подсистемы «Координатор».

Поток событий состоит из следующих шагов. На первом шаге Клиент освобождает используемые структуры данных. На втором шаге Клиент завершает свою работу (возвращает поток управления операционной системе). Вариант использования завершается.

На рис. 14 приведены варианты использования подсистемы «Координатор». Координатор взаимодействует с множеством клиентов и должен представлять собой систему высокой готовности. Поэтому варианты использования спроектированы таким образом, чтобы максимально уменьшить время отклика Координатора. Далее приводится детальное описание каждого варианта использования.

Вариант использования «Передать запрос» передает запрос Ядрам СУБД для дальнейшей обработки. Вариант использования начинается, когда Клиент вызывает функцию «Передать запрос».

Поток событий состоит из следующих шагов. На первом шаге Координатор получает запрос от Клиента как параметр функции «Передать запрос». На втором шаге Координатор анализирует файл конфигурации СУБД и формирует список ядер СУБД, которые будут выполнять запрос.

На третьем шаге Координатор передает запрос ядрам СУБД в асинхронном режиме. Вариант использования завершается.

Вариант использования «Передать результат» передает результирующую таблицу, выдаваемую Ядром СУБД, Клиенту. Вариант использования начинается, когда Ядро СУБД вызывает функцию «Передать результат».

Поток событий состоит из следующей последовательности шагов. На первом шаге Координатор получает результирующую таблицу в виде текстового файла от Ядра СУБД в синхронном режиме. На втором шаге Координатор определяет Клиента, который запустил данный запрос и вызывает функцию «Пополнить результат» (передает Клиенту файл с результатом, полученный от Ядра СУБД). Вариант использования завершается.

Данный вариант использования включает в себя следующие два альтернативных потока. Если на шаге 2 основного потока Координатору не удается передать результирующую таблицу Клиенту, то выполняются следующие действия. На первом шаге Координатор записывает в лог-файл сообщение «Клиент недоступен». На втором шаге Координатор вызывает функцию «Прервать». Вариант использования завершается.

Если на шаге 2 основного потока Координатор обнаруживает, что Клиент прервал выполнение запроса, то Координатор удаляет результирующую таблицу и вариант использования завершается.

Вариант использования «Передать сообщение» уведомляет Координатор о возникновении некоторого существенного события, произошедшего с Ядром СУБД. Вариант использования начинается, когда Ядро СУБД вызывает функцию «Передать сообщение».

Основной поток событий состоит из следующей последовательности шагов. На первом шаге Координатор получает сообщение от Ядра СУБД. На втором шаге Координатор интерпретирует полученное сообщение и формирует диагностическое сообщение для клиента. На последнем шаге

Координатор передает данное диагностическое сообщение Клиенту. Вариант использования завершается.

Данный вариант использования включает в себя три альтернативных потока. Если Координатору на третьем шаге основного потока событий не удается передать диагностическое сообщение Клиенту, то начинается выполнение альтернативного потока, состоящего из следующих шагов. На первом шаге Координатор помещает в лог-файл, сообщение «Клиент недоступен». На втором шаге Координатор вызывает функцию «Прервать». Выполнение варианта использования завершается.

Если на шаге 2 основного потока Координатор обнаруживает, что получено сообщение «Обработка запроса завершена», то выполняется следующая последовательность шагов. На первом шаге Координатор удаляет ядро из списка Ядер СУБД, выполняющих данный запрос. На втором шаге Координатор передает Клиенту диагностическое сообщение о том, что Ядро СУБД завершило обработку запроса. Вариант использования завершается.

Если на шаге 1 альтернативного потока 2 Координатор обнаруживает, что список Ядер СУБД, обрабатывающих запрос, пуст, то выполняется следующая последовательность шагов. На первом шаге Координатор передает Клиенту диагностическое сообщение о том, что Ядро СУБД завершило обработку запроса. На втором шаге Координатор передает Клиенту диагностическое сообщение «Обработка запроса завершена». Вариант использования завершается.

Вариант использования «Прервать» останавливает выполнение запроса на ядрах СУБД по требованию Клиента. Вариант использования начинается, когда Клиент вызывает функцию «Прервать».

Основной поток событий состоит из следующей последовательности шагов. На первом шаге Координатор получает от Клиента сообщение «Прервать». На втором шаге Координатор посылает всем ядрам СУБД, задействованным в обработке запроса, сообщение «Прервать». Вариант использования завершается.

3.2. Форматы входных и выходных данных

3.2.1. Спецификация языка RQL

Язык запросов RQL (Relational Query Language) базируется на реляционной алгебре. Описанная здесь нотация включает в себя операции простой выборки и эквисоединения. Простая выборка (`Restrict`) допускает только условия следующего вида: `<Атрибут><Операция сравнения> <Константа>`. Операция эквисоединения (`eQuijoin`) выполняет соединение по равенству одной пары атрибутов. Предложенная нотация может быть легко распространена на другие реляционные операции.

В качестве разделителя лексем могут использоваться пробелы, символы табуляции, перевода строки и возврата каретки в любом количестве и сочетании.

Грамматика языка RQL, описанная с помощью нотации Бэкуса-Наура представлена ниже:

```
<Запрос> ::= <Оператор>
<Запрос> ::= <Запрос>;<Оператор>
<Оператор> ::= <Метка> <Реляционная операция>
<Реляционная операция> ::= <Выборка> | <Эквисоединение>
<Выборка> ::= R <Условие> <Таблица>
<Условие>::= <Номер атрибута> <Операция сравнения> <Константа>
<Эквисоединение> ::= Q <Номер атрибута> <Номер атрибута>
<Таблица> <Таблица>
<Таблица> ::= <Метка> | <Идентификатор хранимой таблицы>
<Метка> ::= <Целое без знака>
<Номер атрибута> ::= <Целое без знака>
<Операция сравнения> ::= > | < | = <Идентификатор хранимой таб-
лицы> ::= #<Целое без знака>
```

3.2.2. Спецификация лог-файла

Лог-файл содержит диагностические сообщения, выдаваемые ядрами иерархической СУБД. В качестве разделителя лексем могут использоваться пробелы, символы табуляции, перевода строки и возврата каретки в любом количестве и сочетании. Описание языка сообщений лог-файла в нотации Бэкуса-Наура имеет следующий вид:

```
<Лог-файл> ::= <Сообщение>{<Сообщение>}
<Сообщение> ::= <Ядро> <Результат> <Комментарий>
<Ядро> ::= <Целое без знака>
<Результат> ::= <Целое без знака>
<Комментарий> ::= <Строка>
```

3.3. Реализация СУБД «Омега»

В данном разделе описаны основные аспекты реализации параллельной СУБД «Омега». Операторы дерева запроса реализованы в соответствии с моделью скобочного шаблона (см. раздел 1.5.1).

В описании реализации реляционных операторов используются следующие обозначения:

– `left` – указатель на скобочный шаблон левого сына;
– `right` – указатель на скобочный шаблон правого сына;
– `node` – указатель на объемлющий скобочный шаблон;
– `left->result` – указатель на выходной буфер левого сына;
– `right->result` – указатель на выходной буфер правого сына;
– `result` – указатель на кортеж.

3.3.1. Реализация оператора обмена `exchange`

Оператор **`exchange`** представляет собой оператор, обеспечивающий автоматическое распараллеливание запроса. Структура и семантика оператора **`exchange`** были описаны в разделе 1.5.2. В настоящем разделе детально рассматривается реализация данного оператора.

Оператор **`exchange`** вводит в набор физических операций четыре новых оператора: **`merge`**, **`split`**, **`scatter`** и **`gather`**, каждый из которых реализован в соответствии с итераторной моделью управления. Для реализации данных операторов вводится специальный кортеж `LEOFTUPLE`, который информирует интерпретатор о временном отсутствии кортежей в потоке параллельного агента. Необходимость введения `LEOFTUPLE` обусловлена тем, что в процессе обработки запроса может возникать одна из двух следующих ситуаций:

```
/* Реализация метода Next для оператора split */
if (Next(right)) == LEOFTUPLE)
  return LEOFTUPLE;
// Получить очередной кортеж от левого сына
tuple = Next(left);
if (tuple == LEOFTUPLE)
  return LEOFTUPLE;
// Проверить состояние входного потока.
if (tuple == EOFTUPLE) {
  right->result = tuple;
  Next(right);
  return LEOFTUPLE;
}
dest=distribute_fn(tuple); // Вычислить номер узла-приемника
// Передать кортеж оператору merge
if (dest == WHOAMI)
  return result;
// Поместить кортеж в выходной буфер оператора scatter
right->result = tuple;
Next(right);
return LEOFTUPLE;
```

Рис. 15. Схема реализации оператора **split**.

– параллельный агент завершил обработку выделенного ему отрезка данных и простаивает (см. раздел 2.3.2);
– оператор **scatter** не принял кортежей, однако ожидает поступления кортежей по сети от других агентов.

При возникновении одной из таких ситуаций интерпретатор получает от оператора кортеж LEOFTUPLE, который сигнализирует о том, что попытку необходимо будет повторить через некоторый промежуток времени.

*Схема реализации на псевдокоде функции **Next** оператора **split*** представлена на рис. 15. Оператор **split** определяет стратегию расщепления кортежей, поступающих из входного потока на «свои» и «чужие».

Как показано на данной схеме, за один вызов функции **Next** оператор **split** выполняет обработку одного кортежа из выходного буфера левого сына. При этом для начала работы оператора **split** необходима готовность оператора **scatter** к приему кортежей. В том случае, если оператор **scatter** не готов к приему очередного кортежа, оператор **split** возвращает значение LEOFTUPLE.

```
/* Реализация метода Next для оператора scatter */
while (TRUE) {
  switch (state) {
  case READY:
    // Оператор SCATTER готов обработать кортеж.
    if (IsEOF(node->result)) {
      state = SENDBUFS;
      break;
    }
    // Если от оператора SPLIT был получен не EOF
    dest = distribute_fn(tuple);
    // Помещаем кортеж в буфер
    BUF[dest][BUF[dest].current++] = tuple;
    // Если один из буферов заполнен - отправить все буферы
    if (BUF[dest].current == BUF_SIZE) {
      state = SENDBUFS;
      break;
    }
    // Если буфер заполнен не до конца
    return node->result;
  case SENDBUFS:
    for (i=0; i<AGENTS_NUM; i++) {
      Isend(BUF[i], i);
    state = WAIT;
    break;
  case WAIT:
    // Проверить завершение отправки
    for each BUF {
      Test(flag);
      // Выйти, если хотя бы один буфер не отправлен
      if (flag == 0)
        return LEOFTUPLE;
    }
    if (node->result == EOFTUPLE)
      return EOFTUPLE;
    else
      state = READY;
    return node->result;
  }
}
```

Рис. 16. Схема реализации оператора **scatter**.

*Схема реализации на псевдокоде функции **Next** оператора **scatter*** представлена на рис. 16.

Операторы **scatter** и **gather** определяют стратегию обменов кортежами между параллельными агентами в процессе обработки запроса. В данной реализации предлагается стратегия пакетированной передачи кор-

тежей, предназначенная для эффективного использования коммуникационной сети. Как показано на схеме, для этой цели оператор **scatter** использует структуру BUF, представляющую собой набор буферов, каждому из которых ставится в соответствие узел-приемник. При вызове метода **Next** оператор **scatter** помещает кортеж, назначенный к отправке, в соответствующий буфер. Таким образом, для каждого узла-приемника формируется пакет кортежей.

Логика работы оператора **scatter** реализована в виде трех состояний, имеющих следующую семантику:

– «READY» – оператор **scatter** готов к приему очередного кортежа. В данном состоянии оператор **scatter** принимает кортеж и помещает его в соответствующий буфер. Если оператор **scatter** обнаруживает, что необходимо завершить процедуру формирования пакетов, то он переходит в состояние «SENDBUFS»;

– «SENDBUFS» – оператор **scatter** выполняет формирование пакетов кортежей и инициализирует отправку данных пакетов соответствующим узлам. После выполнения инициализации отправки пакетов оператор **scatter** выполняет безусловный переход в состояние «WAIT».

– «WAIT» – оператор **scatter** ждет подтверждения об отправке всех пакетов кортежей. Если отправка пакетов не завершена, оператор **scatter** возвращает значение LEOFTUPLE, сигнализируя, таким образом, оператору **split** о невозможности обработки очередного кортежа. Если оператор **scatter** обнаруживает, что все пакеты отправлены по назначению, то выполняет переход в состояние «READY».

Как можно заметить из предложенной на рис. 15 схемы, оператор **scatter** использует метод асинхронной передачи сообщений. Данный метод позволяет уменьшить время простоя при отправке пакетов кортежей узлам-приемникам и решает проблему возникновения тупиковых ситуаций.

Схема реализации на псевдокоде функции **`Next`** *оператора* **`gather`** представлена на рис. 17.

В данной реализации используются следующие функции и структуры данных:

– Функция `Irecv()` выполняет инициализацию асинхронного приема пакета кортежей. В качестве параметров данная функция принимает указатель на буфер, в который будет помещен принимаемый пакет, узел-отправитель, от которого будет принят пакет и максимально допустимый размер принимаемого пакета.

– Функция `Test()` выполняет проверку поступления пакета, удовлетворяющего условиям, указанным при инициализации приема. В случае успешного завершения приема возвращает в параметре `flag` значение `TRUE`.

– Константа `NODES_NUM` задает количество контрагентов, от которых оператор **`gather`** может ожидать пакеты кортежей.

Логика работы оператора **`gather`** реализована в виде трех состояний, имеющих следующую семантику:

– «`READY`» – Оператор **`gather`** готов выдать очередной кортеж из пакета, принятого им по сети.

«`WAIT`» – оператор **`gather`** ждет поступления по сети очередного пакета кортежей. В данном состоянии оператор **`gather`** возвращает значение `LEOFTUPLE`, которое побуждает оператор **`merge`** на некоторое время отложить попытку получения кортежей по сети.

– «`EOF`» – оператор gather получил EOF-кортеж от всех контрагентов и завершен прием кортежей по сети.

В основе данной реализации оператора **`gather`** находится процедура асинхронного приема пакета, которая состоит из двух фаз: *инициализации* и *ожидания*. На фазе инициализации оператор **`gather`** определяет порядок

```
/* Реализация метода Next для оператора gather */
switch (state) {
  case READY:
    result = BUF[currentTuple];
    currentTuple++;
    if (BUF[currentTuple] == NULL_VALUE) {
      // Инициировать прием очередного пакета кортежей
      Irecv(BUF, ANY_SOURCE, BUF_SIZE_MAX);
      state = WAIT;
      break;
    }
    if (result == EOFTUPLE) {
      EOFCounter++;
      if (EOFCounter == NODES_NUM) {
        state = EOF;
        break;
      }
      return LEOFTUPLE;
    }
    return result;
  case WAIT:
    Test(flag);
    if (flag == 0)
      return LEOFTUPLE;
    state = READY;
    break;
  case EOF:
    return EOFTUPLE;
}
```

Рис. 17. Схема реализации оператора **gather**.

приема пакетов и задает параметры принимаемых пакетов. На фазе ожидания оператор **gather** ждет поступления очередного пакета и возвращает оператору **merge** кортеж LEOFTUPLE.

Преимущество такой асинхронной процедуры состоит в уменьшении накладных расходов на прием поступающих из сети пакетов кортежей. В процессе приема очередного пакета параллельный агент может выполнять обработку кортежей, поступающих из его входных потоков с диска.

Схема реализации на псевдокоде функции ***Next*** *оператора* ***merge*** представлена на рис. 18. Оператор **merge** определяет стратегию чтения кортежей с диска, и из сети. В данной реализации предлагается попеременная схема опроса левого и правого сына. На четной итерации оператор

```
/* Реализация метода Next для оператора merge */
even = !even
if (even) {
  Получить кортеж от оператора gather
  ltuple = Next(left);
  if ((ltuple != LEOFTUPLE) && (ltuple != EOFTUPLE))
    return ltuple;
  Получить кортеж от оператора split
  rtuple = Next(right);
  if ((rtuple != LEOFTUPLE) && (rtuple != EOFTUPLE))
    return rtuple;
}
else if (!even) {
  Получить кортеж от оператора split
  rtuple = Next(right);
  if ((rtuple != LEOFTUPLE) && (rtuple != EOFTUPLE))
    return rtuple;
  Получить кортеж от оператора gather
  ltuple = Next(left);
  if ((ltuple != LEOFTUPLE) && (ltuple != EOFTUPLE))
    return ltuple;
}
if ((ltuple == EOFTUPLE) && (rtuple == EOFTUPLE))
  return EOFTUPLE;
else
  return LEOFTUPLE;
```

Рис. 18. Схема реализации оператора **merge.**

merge сначала выполняет попытку получить кортеж с диска, от оператора **split**. В случае неудачи выполняется попытка чтения кортежа, из выходного буфера оператора **gather**. На нечетной итерации порядок обращения к левому и правому сыну меняется. Данная схема была предложена для устранения дисбаланса между операциями обмена с сетью и диском. В настоящей реализации операция чтения кортежей из сети признается равноправной операции чтения кортежей с диска.

3.3.2. Механизм балансировки загрузки

Реализация механизма балансировки загрузки в иерархической СУБД «Омега» обеспечивается подсистемами *Менеджер параллельных агентов* и *Менеджер заданий*. Менеджер параллельных агентов выполняет функции управления параллельными агентами. Менеджер заданий выполняет функ-

ции сервера статистики, отражающий ход обработки запроса. Менеджер заданий ведет *таблицу агентов*, в которой содержатся сведения о состоянии входных потоков каждого агента.

В предлагаемой реализации процедура балансировки загрузки носит асинхронный характер. При возникновении ситуации балансировки интерпретатор передает менеджеру агентов сообщение о необходимости выполнения балансировки и не прерывает обработку запроса. Если в процессе обработки запроса поток полностью вырабатывает выделенный ему отрезок данных, то при каждом последующем вызове он немедленно возвращает кортеж EOFTUPLE.

Ситуация балансировки загрузки возникает в следующих двух случаях:

– получен сигнал от таймера, о завершении очередного кванта времени, выделенного агенту на обработку запроса.
– один из потоков параллельного агента выдал кортеж LEOFTUPLE;

В основе данной реализации механизма балансировки лежит понятие кванта времени, который определяет периодичность, с которой параллельный агент должен отправлять отчет о состоянии входных потоков менеджеру заданий. При получении сигнала от таймера об истечении выделенного ему времени работы, агент отправляет менеджеру агентов сообщение «Выполнить балансировку», инициализирует системный таймер новым квантом времени и продолжает обработку запроса. Если агент получает из входного потока кортеж LEOFTUPLE, то он передает менеджеру агентов сообщение «Поток пуст» и продолжает обработку запроса.

При поступлении сообщения от интерпретатора, менеджер параллельных агентов формирует *отчет*, в котором описывается состояние каждого из входных потоков агента. Данный отчет передается менеджеру заданий. На рис. 19 показан сценарий балансировки загрузки для случая, когда

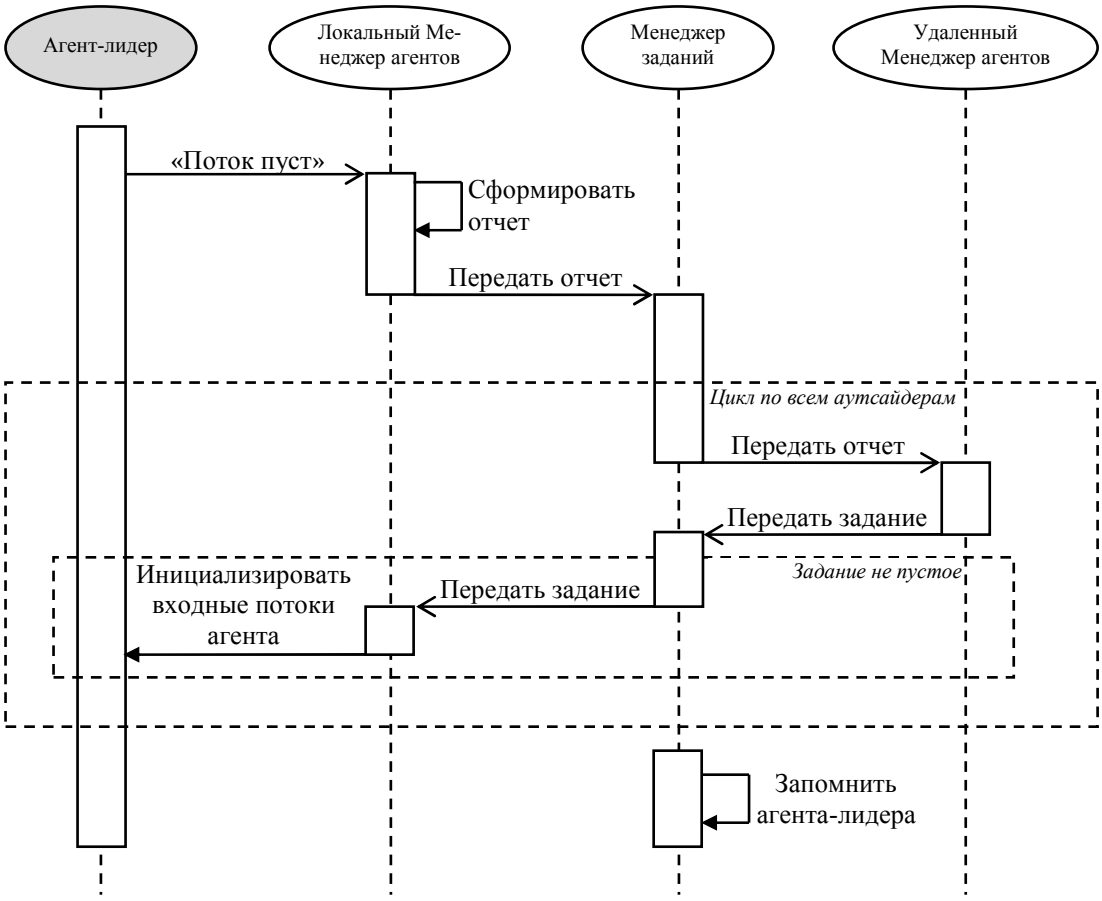

Рис. 19. Диаграмма последовательности для случая, когда процесс балансировки загрузки инициирует агент-лидер.

процесс балансировки инициирован агентом-лидером. Как показано на данной диаграмме, менеджер заданий принимает отчет от менеджера агентов и выполняет процедуру поиска агента-аутсайдера, включающую в себя следующие два основных этапа:

– Поиск агента-аутсайдера в таблице агентов;

– Выполнение запроса на получение задания.

Если поиск агента-аутсайдера завершается неудачей, менеджер заданий помещает отчет данного агента в таблицу агентов и завершает процедуру балансировки. Если от агента-аутсайдера успешно получено задание, менеджер заданий передает его менеджеру агентов, который инициализирует входные потоки агента-лидера отрезками данных, содержащимися в задании.

На рис. 20 показан сценарий балансировки загрузки для случая, когда процесс балансировки инициируется агентом-аутсайдером. Как можно заметить, в данном случае будут выполняться действия аналогичные предыдущей ситуации с той лишь разницей, что менеджер заданий будет выполнять поиск подходящего для балансировки агента-лидера.

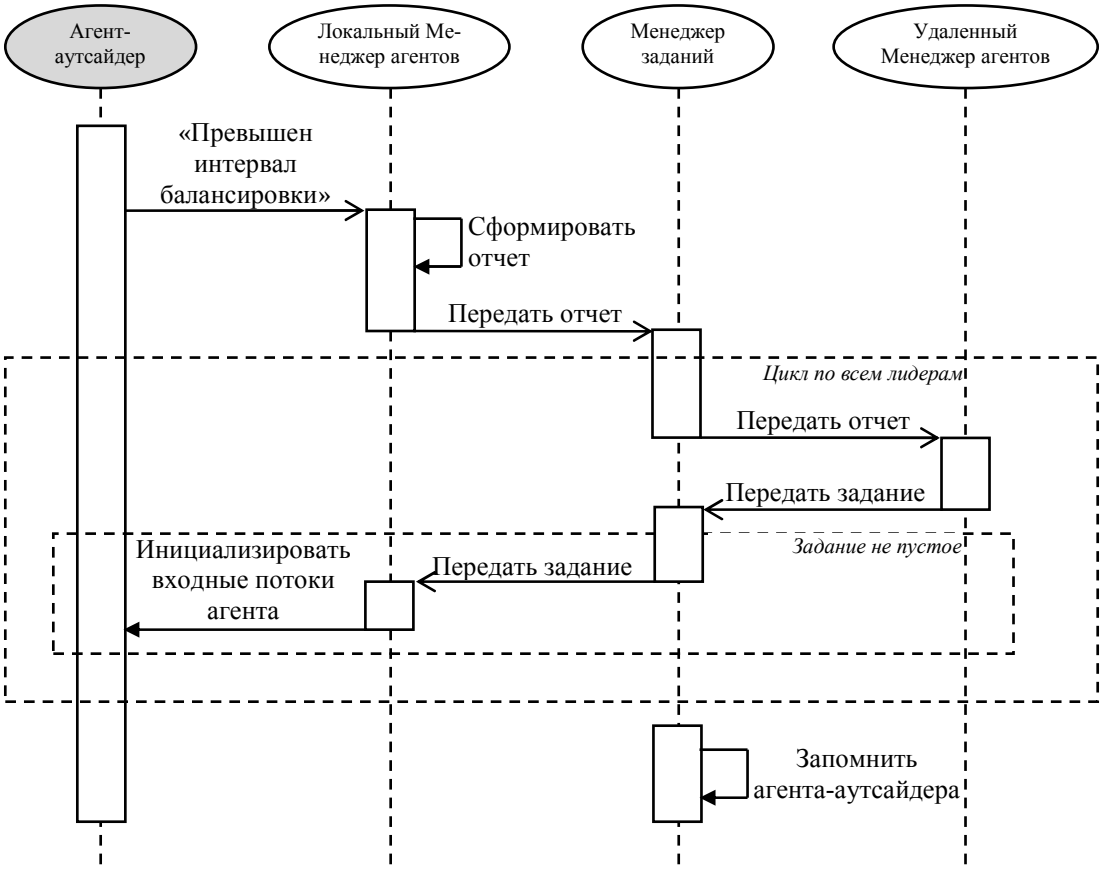

Рис. 20. Диаграмма последовательности для случая, когда процесс балансировки загрузки инициирует агент-аутсайдер.

ГЛАВА 4. ВЫЧИСЛИТЕЛЬНЫЕ ЭКСПЕРИМЕНТЫ

Для исследования предложенного метода балансировки загрузки было проведено несколько серий вычислительных экспериментов на базе разработанного прототипа иерархической СУБД «Омега». Изучена эффективность алгоритма балансировки загрузки в ситуациях перекоса по значению атрибута фрагментации и перекоса по значению атрибута соединения. Исследовано влияние данного алгоритма на показатель масштабируемости СУБД, получены оценки оптимальных параметров балансировки.

4.1. Операция соединения методом хеширования в оперативной памяти

Для исследования алгоритма балансировки загрузки в прототипе иерархической СУБД «Омега» был использован алгоритм соединения хешированием в оперативной памяти [10]. Данный алгоритм интенсивно используется в современных промышленных СУБД при обработке запросов в тех случаях, когда одно из соединяемых отношений целиком помещается в оперативной памяти.

Современные кластерные системы имеют значительную суммарную оперативную память. Например, кластер СКИФ Урал имеет суммарную оперативную память 1.33 ТБ. В контексте систем баз данных это означает, что при использовании фрагментного параллелизма отношение размером меньше 1.33 ТБ может быть фрагментировано по процессорным узлам таким образом, что каждый фрагмент этого отношения целиком поместится в оперативную память. В бинарных реляционных операциях обычно одно из отношений по размеру значительно превосходит другое. Применительно к операции соединения меньшее отношение называется *опорным*, а большее - *тестируемым*. В реальных приложениях баз данных случаи, когда опорное отношение имеет размер более терабайта, достаточно редки. Поэтому алго-

ритм соединения методом хеширования в контексте кластерных систем имеет важное практическое значение.

Алгоритм выполнения реляционной операции соединения методом хеширования в оперативной памяти можно разделить на две фазы [14].

На *фазе распределения* выполняется инициализация операции соединения и строится хеш-таблица опорного отношения *R* в оперативной памяти. При этом каждый процессорный узел выполняет следующие операции:

1. Выполняет покортежное сканирование своего фрагмента отношения *R*. Для каждого кортежа вычисляется значение функции распределения *p(t)*, вырабатывающей номер узла-приемника данного кортежа. Кортеж передается на узел-приемник.
2. Строит хеш-таблицу в оперативной памяти при помощи функции хеширования *h(t)*, используя кортежи, переданные ему на шаге 1.

На *фазе сравнения* выполняется обработка тестируемого отношения *S* и соединение с кортежами отношения *R*. При этом каждый процессорный узел выполняет следующие операции:

1. Выполняет покортежное сканирование своего фрагмента отношения *S*. Для каждого кортежа вычисляется значение функции распределения *p(t)*. Кортеж передается на узел-приемник.
2. Принимает переданный ему на шаге 1 кортеж и выполняет соединение с кортежами из хеш-таблицы, построенной на шаге 2 фазы распределения. Формирует результирующий кортеж.

Перераспределение кортежей между процессорными узлами на каждой фазе алгоритма осуществляет оператор **exchange**. Пример использования оператора **exchange** показан на рис. 21.

На данном рисунке представлен параллельный план запроса, реализующего операцию **MHJ** соединения хешированием в основной памяти. Оператор **scan** выполняет сканирование отношения. Оператор **exchange**

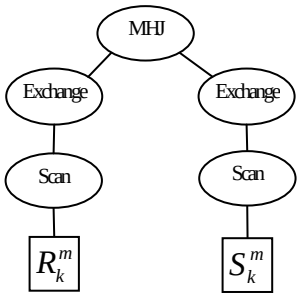

Рис. 21. Параллельный план запроса R⊳⊲S.

вставляется в качестве левого и правого сына оператора **MHJ** и в процессе обработки запроса выполняет перераспределение кортежей, поступающих от оператора **scan**, между процессорными узлами.

4.2. Параметры вычислительных экспериментов

Эффективность предлагаемых методов и алгоритмов исследована в ряде вычислительных экспериментов при различных значениях параметров базы данных. Основные системные параметры приведены в табл. 2. В ходе экспериментов были задействованы вычислительные кластеры, входящие в грид-систему «СКИФ-Полигон». Для обменов сообщениями между процессорными узлами при обработке запроса использовались сети Infiniband и Gigabit Ethernet. При выполнении запроса, в качестве опорного отношения по умолчанию принимается отношение *R*, тестируемого – отношение *S*. При этом предполагается, что оперативной памяти достаточно для того, чтобы на каждом из процессорных узлов хеш-таблица опорного отношения *R* целиком поместилась в оперативной памяти. Балансировка по опорному отношению отключена, поскольку операция формирования хеш-таблицы не требует балансировки ввиду небольшого размера опорного отношения. Поэтому в экспериментах, при выполнении операции соединения методом хеширования в основной памяти балансировка используется на этапе обработки тестируемого отношения.

При проведении экспериментов использовалась тестовая база данных, состоящая из отношений с целочисленными атрибутами. Для

Табл . 2. Параметры среды выполнения экспериментов.

Параметр	Значение
Параметры вычислительной установки	
Количество процессорных узлов, задействованных в тестировании:	64
Тип процессора:	Intel Xeon E5472 (4 ядра по 3.0 GHz)
Оперативная память:	8 ГБ/узел
Дисковая память:	120 ГБ/узел
Тип сети:	InfiniBand (20 ГБит/с) Gigabit Ethernet
Операционная система:	SUSE Linux Enterprise Server 10
Параметры базы данных	
Размер отношения R	1 500 000 записей
Размер отношения S	60 000 000 записей
Параметры запроса	
Индикатор балансировки отношения *R*	0
Индикатор балансировки отношения *S*	1

формирования значений атрибута фрагментации использовалась вероятностная модель IRM (Independent Reference Model) [36]. В соответствии с данной моделью *коэффициент перекоса* θ, $(0 \le \theta \le 1)$ задает распределение, при котором каждому фрагменту назначается некоторый весовой коэффициент $p_i, i = 1 \ldots N$, $p_i = \frac{1}{i^{\theta} H_N^{(\theta)}}$, $\sum_{i=1}^{N} p_i = 1$, где N – количество фрагментов отношения, $H_N^{(s)} = 1^{-s} + 2^{-s} + \ldots + N^{-s}$ – *N-е* гармоническое число порядка *s*. Например, при θ=0.5 распределение весовых коэффициентов соответствует правилу «45-20», в соответствии с которым, 45% кортежей отношения будет храниться в 20% фрагментов. В настоящем исследовании использовались значения коэффициента перекоса 0, *0.2, 0.5* и *0.68*. Значение 0.68 соответствует правилу «70-30», значение 0.2 – «25-15». Значение 0 соответствует равномерному распределению.

Для формирования значений атрибута соединения использовался коэффициент перекоса μ, в соответствии с которым кортежи подразделяются на два класса: «*свои*» и «*чужие*» Коэффициент μ указывает процентное содержание «чужих» кортежей во фрагменте. При этом распределение значений атрибута соединения в рамках каждого из классов подчиняется равномерному закону. Например, при $\mu=50\%$ отношение формируется таким образом, что каждый узел при выполнении соединения будет передавать своим контрагентам по сети примерно 50% кортежей из своего фрагмента. При этом все контрагенты получат примерно одинаковое количество кортежей.

4.3. Исследование параметров балансировки загрузки

В первой серии экспериментов были исследованы параметры балансировки загрузки, такие как интервал балансировки и размер сегмента. Результаты экспериментов показаны на рис. 22 и рис. 23. Исследование интервала балансировки загрузки (см. рис. 22) показывает, что время выполнения запроса практически не изменяется при уменьшении интервала балансировки от 1 секунды до 0.001 секунды. Это показывает, что накладные расходы при увеличении частоты опроса узлов не оказывают существенного воздействия на итоговое время выполнения запроса. Этот факт достигается благодаря асинхронному методу реализации балансировки загрузки в прототипе иерархической СУБД «Омега» (см. раздел 3.3.2). Вместе с этим, увеличение интервала балансировки от 1 секунды до 10 секунд значительно уменьшает эффективность метода. Это объясняется тем, что потенциальные возможности балансировки используются не в полной мере: агент-лидер длительное время простаивает, ожидая появления агента-аутсайдера. Проведенные эксперименты показывают, что хорошим выбором будет использование интервала балансировки равного 1 секунде.

Исследование влияния размера сегмента на эффективность балансировки (см. рис. 23) показывает, что сегменты небольшого размера ухудшают показатели балансировки, поскольку существенно возрастает количество «мелких» балансировок, что ведет к росту накладных расходов на

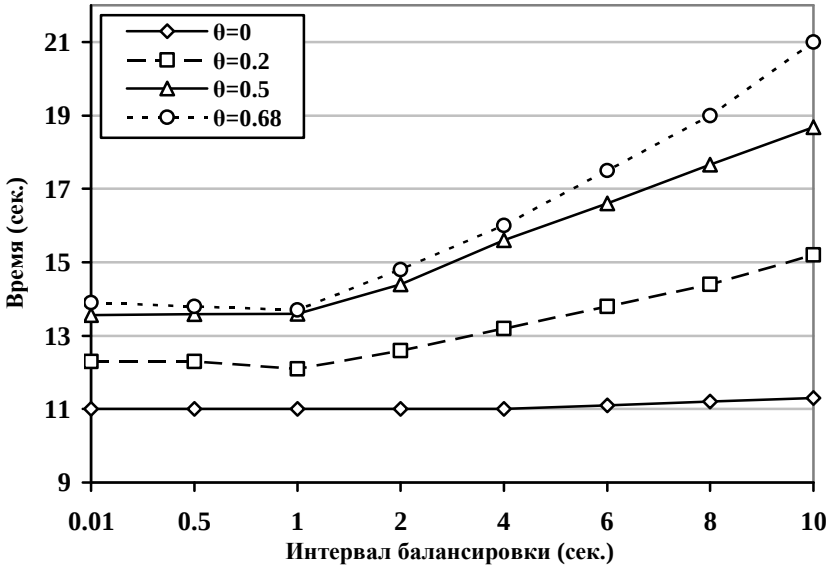

Рис 22. Зависимость времени выполнения запроса от интервала балансировки (n=64, μ =50%).

перераспределение заданий между параллельными агентами. Большие значения размера сегмента также являются неэффективными, так как в этом случае размер сегмента становится сравнимым с размером фрагмента, что делает балансировку невозможной. Таким образом, результаты экспериментов показывают, что оптимальным будет промежуточное значение, которое в данном случае равно 20 000 кортежей. Эта величина составляет примерно 0.01% от средней величины фрагмента.

4.4. Исследование влияния балансировки загрузки на время выполнения запросов

Во второй серии экспериментов исследовалось влияние метода балансировки загрузки на время обработки запросов. Эксперименты проводились с четырьмя значениями перекоса по атрибуту фрагментации на 64 процессорных узлах. Во всех испытаниях данной серии коэффициент μ равен 50%.

Результаты данных испытаний показаны на рис. 24. Данные эксперименты показывают, что оптимальным выбором является значение $\rho=0.8$., которое позволяет практически полностью устранить негативное влияние перекосов по атрибуту фрагментации. Полная репликация не дает

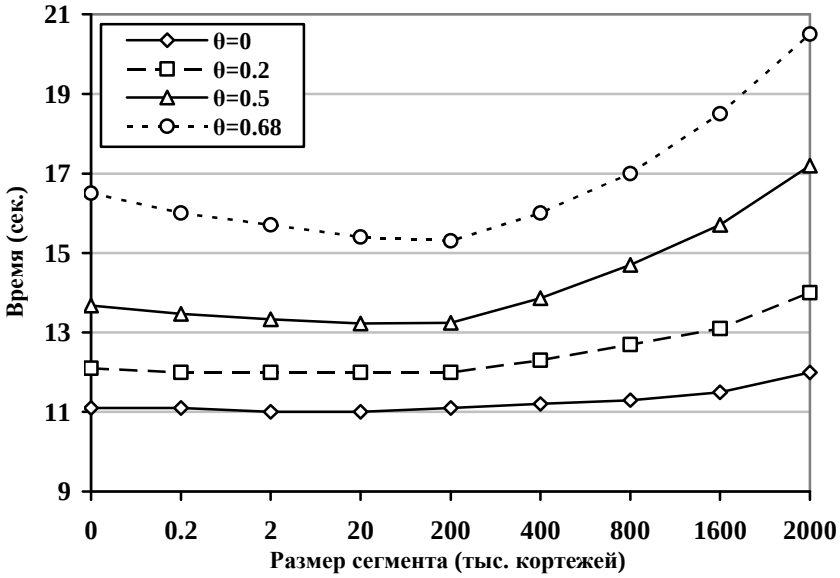

Рис 23. Зависимость времени выполнения запроса от размера сегмента (n=64, μ =50%).

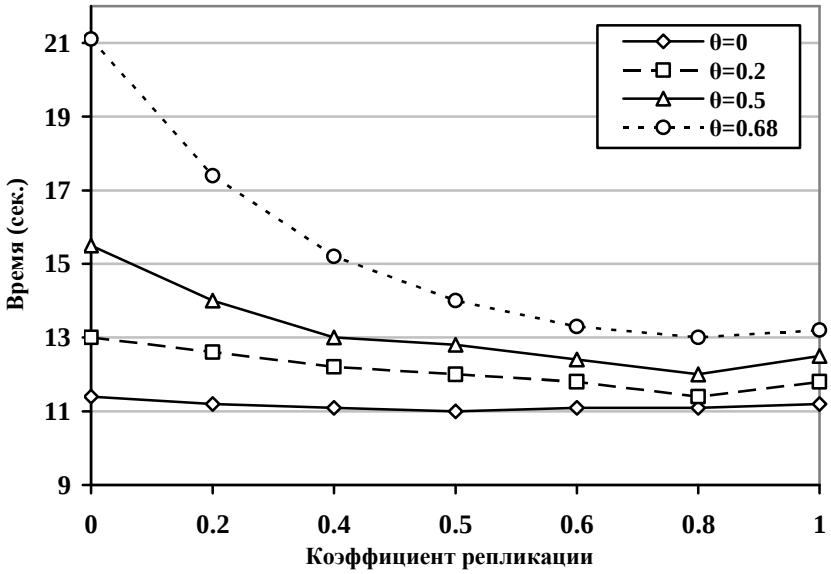

Рис 24. Влияние коэффициента репликации на время выполнения запроса (μ =50%, n=64).

дополнительных преимуществ, поскольку выгода от балансировки загрузки при полной репликации перекрывается увеличением накладных расходов на выполнение балансировки и поиск отрезка данных в реплике. Вместе с этим эффективность метода балансировки загрузки значительно возрастает при увеличении коэффициента перекоса θ. При перекосе «25-15» применение

балансировки загрузки позволяет сократить время выполнения запроса на 13%. В то же время, применение балансировки загрузки при перекосе «70-30» сокращает выполнение запроса на 40%. Проведенные эксперименты показывают, что метод балансировки загрузки может успешно применяться в иерархических многопроцессорных системах баз данных для устранения дисбаланса загрузки при обработке «плохих» запросов.

4.5. Исследование масштабируемости алгоритма балансировки загрузки

Последняя серия экспериментов была посвящена исследованию масштабируемости предложенного в диссертационной работе метода балансировки загрузки. Результаты этих экспериментов представлены на рис. 25, 26 и 27.

Масштабируемость можно определить как меру эффективности распараллеливания алгоритма на многопроцессорных конфигурациях с различным количеством процессорных узлов [16]. Одной из важнейших качественных характеристик эффективности распараллеливания является ускорение. В данной работе под ускорением понимается следующее. Пусть даны две различные конфигурации *A* и *B* параллельной системы баз данных с заданной архитектурой, различающиеся количеством процессоров и ассоциированных с ними устройств. Пусть задан некоторый тест *Q*. Ускорение a_{AB}, получаемое при переходе от конфигурации *A* к конфигурации *B* определяется следующей формулой

$$a_{AB} = \frac{t_{QB}}{t_{QA}},$$

где t_{QA} – время, затраченное конфигурацией *A* на выполнение теста *Q*; t_{QB} – время, затраченное конфигурацией *B* на выполнение теста *Q*.

В экспериментах, показанных на графике 25, исследовалось влияние коэффициента репликации на ускорение. Результаты данных экспериментов демонстрируют ускорение обработки запроса при выполнении балансировки загрузки. При этом наблюдается уменьшение величины ускорения при полной репликации, что связано со значительным увеличением накладных рас-

ходов на использование реплик. При этом следует отметить, что в данном случае хорошим выбором будет $\rho=0.8$.

В экспериментах, показанных на графике 26, исследовалось влияние перекоса по атрибуту фрагментации на ускорение при выполнении балансировки загрузки. Несмотря на то, что перекосы существенно ухудшают значение ускорения, балансировка загрузки обеспечивает ощутимое

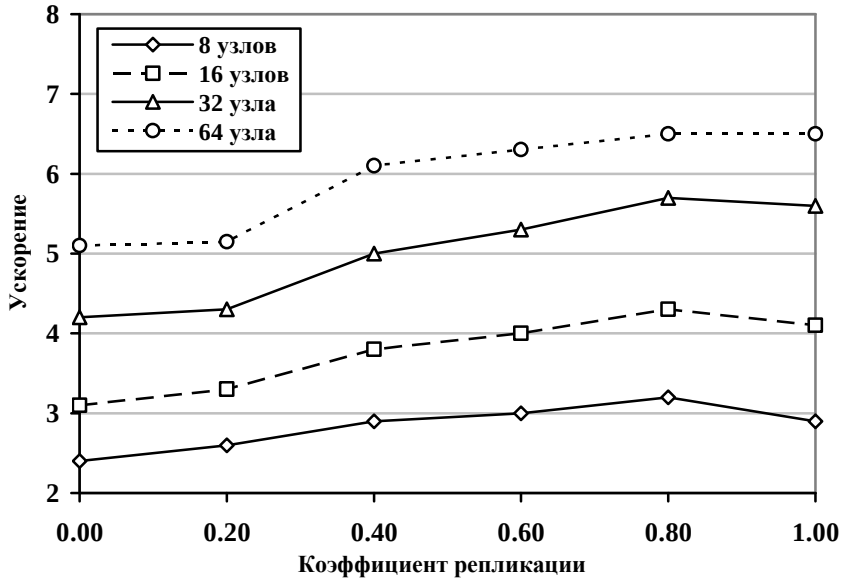

Рис 25. Зависимость ускорения от коэффициента репликации ρ. (*n=64*, *μ=50%*).

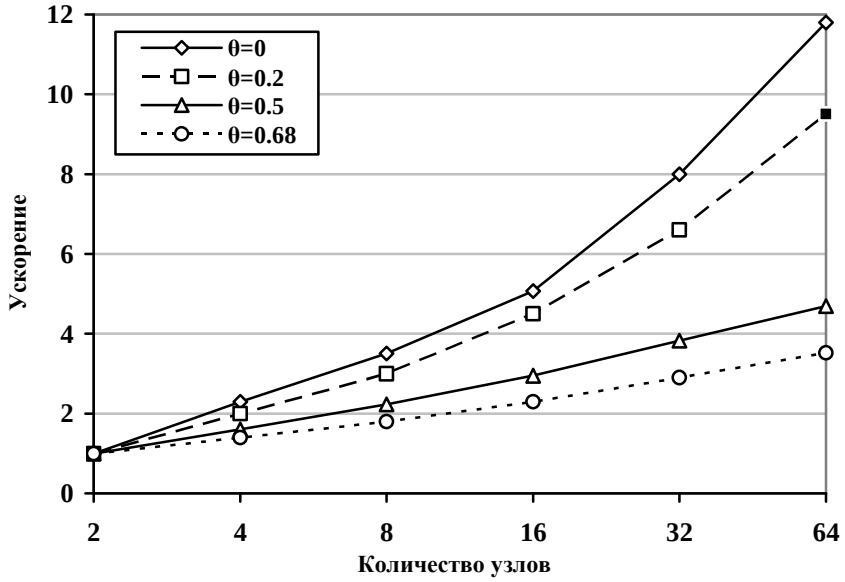

Рис 26. Влияние перекоса по данным на коэффициент ускорения (μ=50%, ρ=0.50).

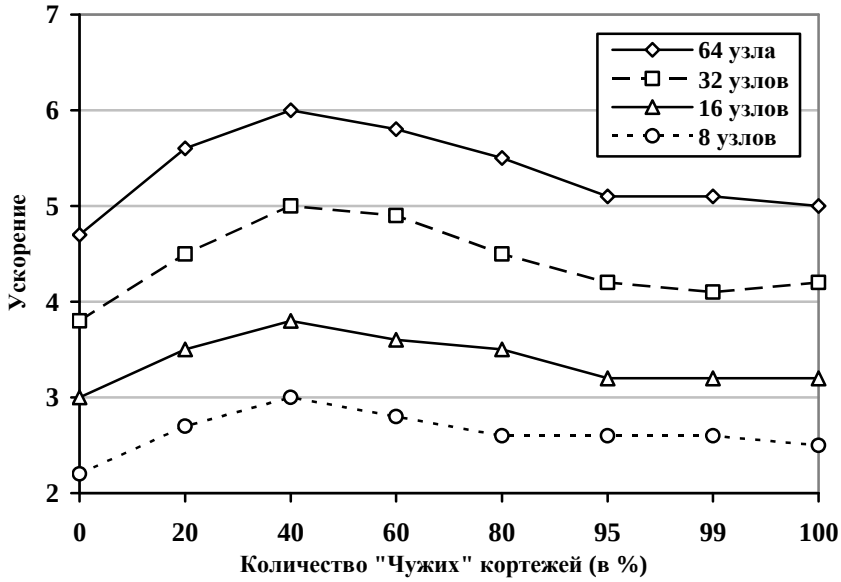

Рис 27. Зависимость коэффициента ускорения от перекоса по значению атрибута соединения μ (*n=64, ρ=0.5*).

ускорение даже при больших значениях коэффициента перекоса.

В экспериментах, показанных на графике 27, было исследовано влияние величины коэффициента перекоса по атрибуту соединения на ускорение. Результаты данных экспериментов показывают, что балансировка загрузки дает наибольший эффект при значении $\mu=50\%$. Опять же следует отметить, что даже в худшем случае, когда 100% кортежей оказываются «чужими», балансировка загрузки обеспечивает заметное ускорение.

ЗАКЛЮЧЕНИЕ

В диссертационной работе были рассмотрены вопросы параллельной обработки запросов в многопроцессорных системах с иерархической архитектурой. Была детально исследована проблема балансировки загрузки и связанная с ней проблема размещения данных. Введена модель симметричной многопроцессорной иерархической системы, которая описывает представительный класс реальных систем и является математическим фундаментом для определения стратегии распределения данных в многопроцессорных иерархиях. Для симметричной иерархии предложен алгоритм формирования реплик, базирующийся на логическом разбиении фрагмента отношения на сегменты равной длины. На основе данного алгоритма разработан метод частичного зеркалирования, использующий функцию репликации. Функция репликации отображает уровень иерархии в коэффициент репликации, который определяет размер реплики по отношению к реплицируемому фрагменту. Доказаны теоремы, позволяющие получить оценки для размеров реплик и трудоемкости их формирования без учета помех. Предложен вариант функции репликации, при котором трудоемкость обновления реплик в многопроцессорной иерархической системе пропорциональна размеру обновляемой части базы данных при условии, что соединительная сеть обладает достаточной пропускной способностью. Описан алгоритм балансировки загрузки, в основе которого лежит метод частичного зеркалирования. Предложена стратегия выбора аутсайдера и конкретная формула для вычисления функции балансировки. Построена модель вариантов использования иерархической многопроцессорной системы, описаны основные требования к иерархической СУБД. Приведена общая структура и варианты использования ключевых подсистем. Предложена оригинальная реализация оператора обмена **exchange**, в основе которой находится механизм пакетирования передаваемых данных и асинхронный режим передачи сообще-

ний. Выполнено проектирование и реализация предложенных методов и алгоритмов в прототипе иерархической СУБД «Омега». Отлаженный код системы составил 10 000 строк на языке Си. С прототипом иерархической СУБД «Омега» проведены масштабные вычислительные эксперименты на кластере «СКИФ Урал». Результаты проведенных вычислительных экспериментов подтверждают эффективность предложенных методов и алгоритмов.

Работа выполнялась при поддержке *Российского фонда фундаментальных исследований* (проект 06-07-89148).

В заключение перечислим основные полученные результаты диссертационной работы, приведем данные о публикациях и апробациях, и рассмотрим направления дальнейших исследований в данной области.

ОСНОВНЫЕ РЕЗУЛЬТАТЫ ДИССЕРТАЦИОННОЙ РАБОТЫ

На защиту выносятся следующие новые научные результаты:

1. Построена математическая модель многопроцессорной иерархии. На основе этой модели разработан метод частичного зеркалирования, который может быть использован для динамической балансировки загрузки. Доказаны теоремы, позволяющие получить аналитическую оценку трудоемкости формирования и обновления реплик при использовании метода частичного зеркалирования.
2. Предложен метод параллельной обработки запросов в иерархических многопроцессорных системах, позволяющий осуществлять эффективную динамическую балансировку загрузки на базе техники частичного зеркалирования.
3. Разработан прототип иерархической СУБД «Омега», реализующий предложенные методы и алгоритмы. Проведены тестовые испытания СУБД «Омега» на вычислительных кластерах, входящих в грид-

систему «СКИФ-Полигон», подтвердившие эффективность предложенных алгоритмов, методов и подходов.

ПУБЛИКАЦИИ ПО ТЕМЕ ДИССЕРТАЦИИ

1. *Лепихов А.В.*Технологии параллельных систем баз данных для иерархических многопроцессорных сред / *Лепихов А.В., Соколинский Л.Б., Костенецкий П.С.* // Автоматика и телемеханика. –2007. –No. 5. -С. 112-125.
2. *Лепихов А.В.* Модель вариантов использования параллельной системы управления базами данных для грид // Вестник ЮУрГУ. Серия «Математическое моделирование и программирование». –Челябинск : ЮУрГУ, 2008 г. –№ 15 (115). –Вып. 1. –С. 42–53.
3. *Лепихов А.В.* Балансировка загрузки при выполнении операций соединения в параллельных СУБД для кластерных систем // Научный сервис в сети Интернет: решение больших задач. Труды Всероссийской научной конференции (22–27 сентября 2008 г., г. Новороссийск). –М.: Изд-во МГУ, 2008. –С. 292–295.
4. *Лепихов А.В.* Стратегия размещения данных в многопроцессорных системах с симметричной иерархической архитектурой / *А.В. Лепихов, Л.Б. Соколинский* // Научный сервис в сети Интернет: технологии параллельного программирования. Труды Всероссийской научной конференции (18–23 сентября 2006 г., г. Новороссийск). –М.: Изд-во МГУ, 2006. –С. 39-42.
5. *Lepikhov A.V.* Data Placement Strategy in Hierarchical Symmetrical Multiprocessor Systems / *A.V. Lepikhov, L.B. Sokolinsky* // Proceedings of Spring Young Researchers' Colloquium in Databases and Information Systems (SYRCoDIS'2006), June 1-2, 2006. -Moscow, Russia: Moscow State University. -2006. -С. 31-36.

6. *А.В. Лепихов* Свидетельство Роспатента об официальной регистрации программы для ЭВМ «Параллельная СУБД «Омега» для кластерных систем» / *А.В. Лепихов, Л.Б. Соколинский, М.Л. Цымблер*; -№2008614996 от 03.10.2008.

Статья [1] опубликована в научном журнале «Автоматика и телемеханика», включенном ВАК в перечень журналов, в которых должны быть опубликованы основные результаты диссертаций на соискание ученой степени доктора наук. В статье [1] А.В. Лепихову принадлежит раздел 3 (стр. 118-124). В работах [4, 5] Л.Б. Соколинскому принадлежит постановка задачи; А.В. Лепихову принадлежат все полученные результаты.

АПРОБАЦИЯ РАБОТЫ

Основные положения диссертационной работы, разработанные модели, методы, алгоритмы и результаты вычислительных экспериментов докладывались автором на следующих международных и всероссийских научных конференциях:

- на Четвертом весеннем коллоквиуме молодых исследователей в области баз данных и информационных систем (SYRCoDIS) (1–2 июня 2006 г., Москва);
- на Всероссийской научной конференции «Научный сервис в сети Интернет: технологии параллельного программирования» (18-23 сентября 2006 г., Новороссийск);
- на Всероссийской научной конференции «Научный сервис в сети Интернет: решение больших задач» (22-27 сентября 2008 г., Новороссийск);
- на Международной научной конференции «Параллельные вычислительные технологии» (29 января – 2 февраля 2007 г., Челябинск).

НАПРАВЛЕНИЯ ДАЛЬНЕЙШИХ ИССЛЕДОВАНИЙ

Теоретические исследования и практические разработки, выполненные в рамках данной диссертационной работы, предполагается продолжить по следующим направлениям.

1. Аналитическое исследование метода частичного зеркалирования: получение аналитических оценок трудоемкости обновления реплик с учетом помех.
2. Исследование эффективности использования различных функций рейтинга при балансировке загрузки. При расчете функции рейтинга планируется использовать дополнительную статистическую информацию об истории операций балансировки загрузки агента, выступающего в роли помощника.
3. Дальнейшее развитие метода частичного зеркалирования. Предполагается применение данного метода в алгоритме GRACE-соединения и гибридного соединения.
4. Реализация метода частичного зеркалирования и алгоритма балансировки загрузки в параллельной СУБД с открытым исходным кодом.

ЛИТЕРАТУРА

1. *Абламейко С.В., Абрамов С.М., Анищенко В.В., Парамонов Н.Н.* Принципы построения суперкомпьютеров семейства «СКИФ» и их реализация// Ежеквартальный научный журнал «Информатика». ОИПИ НАН Беларуси, –2004. №1. С. 89–106.
2. *Воеводин В.В., Воеводин Вл.В.* Параллельные вычисления. –СПб.: БХВ-Петербург, 2002. –608 с.
3. *Воеводин Вл.В.* Решение больших задач в распределенных вычислительных средах. //Автоматика и Телемеханика. –2007, No. 5, С. 32–45.
4. *Гарсиа-Молина Г., Ульман Д., Уидом Д.* Системы баз данных. –М.: Издательский дом «Вильямс», 2004. –1088 с.
5. *Игнатович Н.* Семейство реляционных баз данных IBM DB2 // СУБД. -1997. –№2. –С. 5–17.
6. *Кнут Д.Э.* Искусство программирования, т. 1. Основные алгоритмы, 3-е изд. –М.: Издательский дом «Вильямс», 2000. –720 с.
7. *Костенецкий Л.Б., Лепихов А.В., Соколинский Л.Б.* Некоторые аспекты организации параллельных систем баз данных для мультипроцессоров с иерархической архитектурой // Алгоритмы и программные средства параллельных вычислений: [Сб. науч. Тр.]. -Екатеринбург: УрО РАН. -2006.
8. *Кузнецов С.Д.* SQL. Язык реляционных баз данных. –М.: Майор, 2001. -192 с.
9. *Кузьминский М.* Процессоры для высокопроизводительных вычислений // Открытые системы. –2007. –№ 9. –С. 14–19.
10. *Лепихов А.В.* Балансировка загрузки при выполнении операций соединения в параллельных СУБД для кластерных систем // Научный сервис

в сети Интернет: решение больших задач: тр. Всерос. науч. конф. (22-27 сентября 2008 г., г. Новороссийск). -М., -2008. -С. 292-295.

11. Лепихов А.В. Модель вариантов использования параллельной системы управления базами данных для грид // Вестник ЮУрГУ. Серия «Математическое моделирование и программирование» –2008. –№ 15 (115). -Вып. 1. –С. 42–53.
12. *Лепихов А.В., Соколинский Л.Б.* Стратегия размещения данных в многопроцессорных системах с симметричной иерархической архитектурой // Научный сервис в сети Интернет: технологии параллельного программирования: тр. Всерос. науч. конф. (18–23 сентября 2006 г., г. Новороссийск). –М., –2006. –С. 39–42.
13. *Лепихов А.В., Соколинский Л.Б., Костенецкий П.С.* Технологии параллельных систем баз данных для иерархических многопроцессорных сред // Автоматика и телемеханика. –2007. –No. 5. –С. 112-125.
14. *Новиков Б.А., Домбровская Г.Р.* Настройка приложений баз данных. – СПб.: БХВ-Петербург, 2006. –240 с.
15. *Соколинский Л.Б.* Обзор архитектур параллельных систем баз данных // Программирование. -2004. -№ 6. С. 49-63.
16. *Четверушкин Б.Н.* Высокопроизводительные многопроцессорные вычислительные системы // Вестник российской академии наук. –2002. -Том 72, №9. –С. 786-794.
17. *Alexander W., Copeland G.* Process and dataflow control in distributed data-intensive systems // Proceedings of the 1988 ACM SIGMOD international conference on Management of data, Chicago, Illinois, United States, 1988 -ACM. -1988. –P. 90–98.
18. *Alfawair M., Aldabbas O., Bartels P., Zedan H.* Grid Evolution // IEEE International Conference on Computer Engineering & Systems, Cairo, Egypt,

27-29 November, 2007, Proceedings. –IEEE Computer Society, 2007 -P. 158-163.

19. *Baker M., Apon A., Ferner C., Brown J.* Emerging Grid Standards // Computer. –2005. –Vol. 38, No. 4 –P. 43–50.
20. *Bell G., Gray J.* What's next in high-performance computing // Communications of. ACM. –2002. –Vol. 45, No. 2 –P. 91–95.
21. *Bernstein P., et al.* Query processing in a system for distributed databases (SDD-1) // ACM Transactions on Database Systems –1981. –Vol. 6, No. 4. -P. 602-625.
22. *Boral H., Alexander W., Clay L., Copeland G., Sanforth S., Franklin M., Hart B., Smith M., Valduriez P.* Prototyping Bubba: a Highly Parallel Database System // IEEE Trans. on Knowledge and Data Engineering. –1990. –Vol. 2, No. 1. –P. 4–24.
23. *Boral H.* Parallelism in bubba // Proceedings of the first international symposium on Databases in parallel and distributed systems, Austin, Texas, United States, 1988. –IEEE Computer Society Press. –1988. –P. 68–71.
24. *Borkar S.Y. et al.* Platform 2015: Intel processor and platform evolution for the next decade: [tbp.berkeley.edu/~jdonald/research/cmp/borkar_2015.pdf]. –2005.
25. *Bouganim L., Florescu D., Valduriez P.* Dynamic Load Balancing in Hierarchical Parallel Database Systems // Proceedings of the 22th international Conference on Very Large Data Bases, September 03 – 06, 1996. –Morgan Kaufmann. –1996. P. 436–447.
26. *Cavin R, Hutchby J.A., Zhirnov V., Brewer J.E., Bourianoff G.* Emerging Research Architectures // Computer. –2008. Vol. 41, No. 5. –P. 33–37.
27. *Chamberlin D.D., et al.* A History and Evaluation of System R // Communications of the ACM. –1981. –Vol. 24, No. 10. –P. 632–646.

28. *Chambers L., Cracknell D.* Parallel Features of NonStop SQL // Proceedings of the 2nd International Conference on Parallel and Distributed Information Systems (PDIS 1993), Issues, Architectures, and Algorithms, San Diego, CA, USA, January 20–23, 1993. –IEEE Computer Society, 1993. –P. 69–70.

29. *Chandy K.M., Misra J.* A distributed algorithm for detecting resource deadlocks in distributed systems // First ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing Ottawa, Canada, August 18 – 20, 1982, Proceedings. –ACM, 1982. P. 157–164.

30. *Chen H., Decker J., Bierbaum N.* Future Networking for Scalable I/O // 24th IASTED international Conference on Parallel and Distributed Computing and Networks, Innsbruck, Austria, February 14 – 16, 2006, Proceedings – ACTA Press, 2006. P. 128–135.

31. *Chen T., Raghavan R., Dale J.N.* Cell Broadband Engine Architecture and its first implementation –A performance View // IBM J. Res. Dev., -Vol. 51, №. 5. –2007.

32. *Chen M.-S., Yu P.S., Wu K.-L.* Scheduling and Processor Allocation for Parallel Execution of Multi-Join Queries // Proceedings of the Eighth International Conference on Data Engineering, February 3–7, 1992, Tempe, Arizona. -IEEE Computer Society, 1992. –P. 58–67.

33. *Chien A., Calder B., Elbert S., Bhatia K.* Entropia: architecture and performance of an enterprise desktop grid system // J. Parallel Distrib. Comput. -2003. –Vol. 63, No. 5. –P. 597–610.

34. *Chu W.W., Hellerstein J.* The Exclusive-Writer Approach to Updating Replicated Files in Distributed processing Systems // IEEE Trans. on Computers –1985. –Vol. 34, No. 6. –P. 489–500.

35. *Chu W. W., Merzbacher M., Berkovich L.* The design and implementation of CoBase. SIGMOD Record. –1993. –Vol. 22, No. 2. –P. 517–522.

36. Coffman E.G., Denning P.G., Operating Systems Theory. –Prentice Hall, 1973.

37. *Copeland G., Alexander W., Boughter E., Keller T.* Data placement in Bubba // Proceedings of the 1988 ACM SIGMOD international Conference on Management of Data, United States, Chicago, Illinois, June 01 – 03, 1988. P. 99-108.

38. *Conway S., Walsh R.* HPC Management Software: Reducing the Complexity of HPC Cluster and Grid Resources. Whitepaper: [http://www.findwhitepapers.com/networking/grid-computing/], 2008.

39. *Crawford C.H., Henning P., Kistler M., Wright C.* Accelerating computing with the cell broadband engine processor // 5th Conference on Computing Frontiers, Ischia, Italy, May 5–7, 2008, proceedings. –ACM, 2008. P. 3–12.

40. *Deshpande A., Ives Z., Raman V.* Adaptive query processing. Found. Trends databases. –2007. –Vol. 1, No. 1 P. 1–140.

41. *DeWitt D.J., et al.* The Gamma database machine project // IEEE Transactions on Knowledge and Data Engineering. –1990. –Vol. 2, No. 1. P. 44–62.

42. *DeWitt D.J., Gerber R.H.* Multiprocessor Hash-Based Join Algorithms // Proceedings of 11th International Conference on Very Large Data Bases, Stockholm, Sweden, August 21-23, 1985. Morgan Kaufmann. –1985. –P. 151–164.

43. *DeWitt D.J., Gray J.* Parallel Database Systems: The Future of High-Performance Database Systems // Communications of the ACM. –1992. –Vol. 35, No. 6. –P. 85–98.

44. *Dwyer P.A., Jelatis G.D., Thuraisingham B.M.* Multi-level security in database management systems // Computers and Security. – 1987. –Vol. 6, No. 3. –P. 252–260.

45. *Ferhatosmanogly F., Hakan S.* Optimal Parallel I/O Using Replication // In Proceedings of the 2002 international Conference on Parallel Processing Workshops, August 18 – 21, 2002. –IEEE Computer Society. –2002. -P. 506.

46. *Ferhatosmanoglu H., Tosun A. S., Ramachandran A.* Replicated declustering of spatial data // Proceedings of the Twenty-Third ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, Paris, France, June 14 – 16, 2004. –ACM. –2004. –P. 125–135.

47. *Ferhatosmanoglu H., Tosun A. Ş., Canahuate G., Ramachandran, A.* Efficient parallel processing of range queries through replicated declustering // Distrib. Parallel Databases.–2006. –Vol. 20, No. 2. –P. 117–147.

48. *Foster I.T., Grossman R.L.* Blueprint for the future of high-performance networking: Data integration in a bandwidth-rich world // Communications of the ACM. –2003. –Vol. 46, No. 11. –P. 50–57.

49. *Foster I.T., Kesselman C., Nick J., Tuecke S.* The Physiology of the Grid: An Open Grid Service Architecture for Distributed Systems Integration. Global Grid Forum: [http://www.globus.org/ogsa/], 2002

50. *Foster I.* What is the Grid? A Three Point Checklist: [http://www.gridtoday.com/02/0722/100136.html], 2002.

51. *Fushimi S., Kitsuregawa M., Tanaka H.* An Overview of The System Software of A Parallel Relational Database Machine GRACE // Twelfth International Conference on Very Large Data Bases, Kyoto, Japan, August 25-28, 1986. –Morgan Kaufmann, 1986. –P. 209–219.

52. *Ganguly S., Hasan W., Krishnamurthy R.* Query optimization for parallel execution // ACM SIGMOD international Conference on Management of Data, San Diego, California, United States, June 02 – 05, 1992, Proceedings. –ACM, 1992. –P. 9–18.

53. *Garcia-Molina H., Lindsay B.* Research directions for distributed databases // SIGMOD Record. –1990. –Vol. 19, No. 4 –P. 98–103.

54. *Geer D.* Grid Computing Using the Sun Grid Engine. Whitepaper: [http://whitepapers.silicon.com/0,39024759,60108958p,00.htm], 2003.

55. *Gligor V., Popescu-Zeletin R.* Transaction management in distributed heterogeneous database management systems // Information Systems. –1986. -Vol. 11, No. 4. –P. 287–297.

56. *Gounaris A., Paton N. W., Fernandes A., Sakellariou R.* Adaptive Query Processing: A Survey. In 19th British National Conference on Databases, Sheffield, UK, 2002. –P. 11–25.

57. *Gounaris A., Sakellariou R., Paton N. W., Fernandes A. A.* A novel approach to resource scheduling for parallel query processing on computational grids // Distrib. Parallel Databases. –Vol. 19, No. 2-3 –2006. P. 87–106.

58. *Graefe G.* Encapsulation of Parallelism in the Volcano Query Processing Systems // Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23–25, 1990. –ACM Press, 1990. –P. 102–111.

59. *Graefe G.* Query Evaluation Techniques for Large Databases // ACM Computing Surveys. –1993. –Vol. 25, No 2. –P. 73–169.

60. *Graefe G.* Volcano – An Extensible and Parallel Query Evaluation System // IEEE Trans. Knowl. Data Eng. –1994. –Vol. 6, No. 1. –P. 120–135.

61. *Jennings N.R., Wooldridge M.* Agent Technology: Foundations, Applications and Markets. –Springer Verlag, 1998. –325 p.

62. *Halici U., Dogac A.* Concurrency Control in Distributed Databases Through Time Intervals and Short-Term Locks // IEEE Transactions on Software Engineering. –1989. Vol. 15, No. 8. P. 994–1003.

63. *Harada L. Kitsuregawa M.* Dynamic Join Product Skew Handling for Hash-Joins in Shared-Nothing Database Systems //Proceedings of the 4th international Conference on Database Systems For Advanced Applications April, 11–13, 1995. World Scientific Press, 1995. P. 246–255.

64. *Helal A., Yuan D., El-Rewini H.* Dynamic Data Reallocation for Skew Management in Shared-Nothing Parallel Databases // Distrib. Parallel Databases. –1997. –Vol. 5, No. 3. P. 271–288.

65. *Hong W., Stonebraker M.* Optimization of parallel query execution plans in XPRS // First international Conference on Parallel and Distributed information Systems, Miami, Florida, United States, 1991. –IEEE Computer Society Press, 1991. –P. 218–225.

66. *Hua K. A., Lee C., Hua C. M.* Dynamic Load Balancing in Multicomputer Database Systems Using Partition Tuning. *IEEE Trans. on Knowl. and Data Eng.* –1995. –Vol. 7, No. 6. P. 968–983.

67. *Hua K.A., Lee C.* Handling Data Skew in Multiprocessor Database Computers Using Partition Tuning // Proceedings of the 17th International Conference on Very Large Data Bases, September 03–06, 1991, San Francisco, CA. - Morgan Kaufmann, 1991, –P. 525-535.

68. *Hua K.A., Lee C.* An adaptive data placement scheme for parallel database computer systems // Proceedings of the sixteenth international conference on Very large databases, Brisbane, Australia. –Morgan Kaufmann. –1990. P. 493-506.

69. *Katayama Y.* Trends in Semiconductor Memories. *IEEE Micro* 17, 6 (Nov. 1997), 10-17.

70. *Kim C.* Future Memory Technology Trends and Challenges // 7th international Symposium on Quality Electronic Design, March 27 – 29, 2006, proceedings. –IEEE Computer Society. –P. 513.

71. *Kitsuregawa M, Tanaka H, Moto-Oka T.* Application of Hash to Data Base Machine and Its Architecture // New Generation Comput. –1983. –Vol. 1, No. 1 P. 63–74.

72. *Kossmann D.* The state of the art in distributed query processing. ACM Comput. Surv. –2000. –Vol. 32, No. 4 –P. 422–469.

73. *Kotowski N., Lima A., Pacitti E., Valduriez P. Mattoso M.* Parallel query processing for OLAP in Grids // Concurrency and Computation: Practice and Experience. –Wiley InterScience. –2008.

74. *Koupras E.* Grid Computing: Past, Present and Future. IBM Whitepaper: [http://www-03.ibm.com/grid/pdf/innovperspective.pdf], june 2006.

75. *Lakshmi M.S., Yu P.S.* Effect of Skew on Join Performance in Parallel Architectures // Proceedings of the first international symposium on Databases in parallel and distributed systems, Austin, Texas, United States. IEEE Computer Society Press. –1988. P. 107–120.

76. *Lepikhov A.V., Sokolinsky L.B.* Data Placement Strategy in Hierarchical Symmetrical Multiprocessor Systems // Proceedings of Spring Young Researchers' Colloquium in Databases and Information Systems (SYRCoDIS'2006), June 1-2, 2006. -Moscow, Russia: Moscow State University. -2006. -C. 31-36.

77. *Livny M., Khoshafian S., Boral H.* Multi-disk management algorithms // SIGMETRICS Perform. Eval. Rev. –1987. Vol. 15, No. 1. –P. 69-77.

78. *Lo Y., Hua K.A., Young H.C.* GeMDA: A Multidimensional Data Partitioning Technique for Multiprocessor Database Systems // Distributed and Parallel Databases. –2001. Vol. 9, No. 3. P. 211–236.

79. *Lowenthal D.K., Andrews G.R.* An Adaptive Approach to Data Placement. // 10th international Parallel Processing Symposium, April 15 – 19, 1996, Proceedings. -IEEE Computer Society, 1996. P. 349–353.

80. *Lu H., Tan K.*-L. Dynamic and Load-balanced Task-Oriented Database Query Processing in Parallel Systems // Advances in Database Technology - EDBT'92, 3rd Int. Conf. on Extending Database Technology, Vienna, Austria, March 23-27, 1992, Proceedings. Lect. Not. in Comp. Sc., Vol. 580. Springer. -1992. -P. 357-372.

81. *Mach W., Schikuta E.* Parallel Database Join Operations in Heterogeneous Grids // In Proceedings of the Eighth international Conference on Parallel and Distributed Computing, Applications and Technologies (December 03 – 06, 2007). –IEEE Computer Society. –2007. P. 236–243.

82. *Maertens H.* A Classification of Skew Effects in Parallel Database Systems // 7th International Euro-Par Conference, August 28-31, 2001, Manchester, UK, Proceedings. Springer. Vol. 2150. -2001. –P.291-300.

83. *Mehta M., DeWitt D.J.* Data Placement in Shared-Nothing Parallel Database Systems // The VLDB Journal. -January 1997. –Vol. 6, No. 1. –P. 53–72.

84. *Meuer H.W.* The TOP500 Project: Looking Back Over 15 Years of Supercomputing Experience // Informatik Spektrum. –2008. –Vol. 31, No. 3. -P. 203–222.

85. *Moore N., Conti A, Leeser M, Smith L.A.*Vforce: An Extensible Framework for Reconfigurable Supercomputing // IEEE Computer. –2007. Vol. 40, No. 3. –P.39–49.

86. *Omiecinski E.* Performance Analysis of a Load Balancing Hash-Join Algorithm for a Shared Memory Multiprocessor // 17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings. Morgan Kaufmann. -1991. -P. 375-385.

87. *Page J.* A Study of a Parallel Database Machine and its Performance: the NCR/Teradata DBC/1012 // Advanced Database Systems, 10th British National Conference on Databases, BNCOD 10, Aberdeen, Scotland, July 6–8,

1992, Proceedings. –Springer, 1992 (Lecture Notes in Computer Science, Vol. 618). –P. 115–137.

88. *Parkhurst J., Darringer J., Grundmann B.* From single core to multi-core: preparing for a new exponential // IEEE/ACM international Conference on Computer-Aided Design, San Jose, California, November 05 – 09, 2006. -ACM, 2006. –P. 67–72.

89. *Pfister G.* Sizing Up Parallel Architectures // Database Programming Design OnLine [http://www.dbpd.com], –1998. Vol. 11, No. 5.

90. *Rubinovitz H., Thuraisingham B.* Security constraint processing in a distributed database environment // 22nd Annual ACM Computer Science Conference on Scaling Up, Phoenix, Arizona, United States, March 08 – 10, 1994, Proceedings. –ACM, 1994. –P. 356–363.

91. *Rahm E. Marek R.* Dynamic Multi-Resource Load Balancing in Parallel Database Systems. Proceedings of the 21th international Conference on Very Large Data Bases, San Francisco, September 11–15, 1995. Morgan Kaufmann. –1995. P. 395–406.

92. *Reddy P.K., Bhalla S.* Deadlock prevention in a distributed database system // ACM SIGMOD Record. –1993. –Vol. 22, No. 3. –P. 40–46.

93. *Reed D.A.* Grids, the TeraGrid, and Beyond // Computer. –2003. Vol. 36, No. 1. –P. 62–68.

94. RFC4180: Common Format and MIME Type for Comma-Separated Values (CSV) Files: [http://tools.ietf.org/html/rfc4180], October 2005.

95. *Roberts L. G.* Beyond Moore's Law: Internet Growth Trends // Computer -2000. Vol. 33, No. 1 –P. 117–119.

96. *Roure D., Baker M., Jennings N. R. Shadbolt N.* The evolution of the Grid, in: Grid Computing: Making The Global Infrastructure a Reality, Berman F., Hey A. Fox G. –Wiley Publishing Company, 2003, –P. 65–100.

97. *Ruggiero W., Sato L.* M. Data warehouse using parallel processing on a distributed environment // In Proceedings of the 2007 Annual Conference on international Conference on Computer Engineering and Applications, Gold Coast, Queensland, Australia, January 17 – 19, 2007. –Stevens Point. –2007. P. 212–217.

98. *Rumbaugh J.* Getting Started – Using Use Cases to Capture Requirements // Journal of Object Oriented Programming. –1994. –vol. 7, No 5. –P. 8–12.

99. *Scheuermann P., Weikum G., Zabback* P. Data partitioning and load balancing in parallel disk systems // The VLDB Journal. –1998. –Vol. 7, No. 1. P. 48–66.

100. *Schoene R., Nagel W.E., Oiger S* Analyzing Cache Bandwidth on the Intel 2 Core Architecture // Parallel Computing: Architectures, Algorithms and Applications. –2007. –Vol. 38. –P. 365–372.

101. *Selic B.* UML 2: a model-driven development tool // IBM Syst. J. –2006. -Vol. 45, No 3. –P. 607–620.

102. *Shekhar S., Ravada S., Kumar V., Chubb D., Turner G.* Declustering and Load-Balancing Methods for Parallelizing Geographic Information Systems. IEEE Trans. on Knowl. and Data Eng. –1998. Vol. 10, No. 4. P. 632–655.

103. *Son S. H.* Replicated data management in distributed database systems // ACM SIGMOD Record. –1988. –Vol. 17, No. 4. –P. 62–69.

104. *Stonebraker M.* The case for shared nothing // Database Engineering Bulletin. –1986. –Vol. 9, No. 1. –P. 4–9.

105. *Stonebraker M., Aoki P., Litwin W., Pfeffer A., Sah A., Sidell J., Staelin C., Yu A.* Mariposa: a wide-area distributed database system // The VLDB Journal. –1996. –Vol. 5, No. 1. –P. 48–63.

106. *Stonebreaker M.* The design and implementation of distributed INGRES // The INGRES papers: anatomy of a relational database system. Addison-

Wesley Series In Computer Science. Addison-Wesley Longman Publishing Co., Boston, MA, 187-196.

107. *Thomas G., et al.* Heterogeneous distributed database systems for production use. ACM Comput. Surv. –1990. –Vol. 22, No. 3. P. 237–266.

108. *Thomas R.* Majority Consensus Approach to Concurrency Control for Multiple Copy Distributed database Systems // ACM Trans. Database Syst. –1979. –Vol. 4, No. 2. –P. 180–209.

109. *Traiger I.L., Gray J.N., Galtieri C.A., Lindsay B.G.* Transactions and Consistency in Distributed Database Management Systems // ACM Trans. Database Syst. –1982. –Vol. 7, No. 3. P. 323–342.

110. *Tremblay M., Chaudhry S.* A Third-Generation 65nm 16-Core 32-Thread Plus 32-Scout-Thread CMT SPARC Processor // IEEE International multithreaded Solid State Circuits Conference, San Francisco, CA, 3–7 february, 2008, proceedings. –IEEE Computer Society, 2008. P. 82-83.

111. *Vangal S.* An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS // Solid-State circuits conference, February 11–15,2007. –P. 98–105.

112. *Williams M.H., Zhou S.* Data Placement in Parallel Database Systems // Parallel database techniques / IEEE Computer society. –1998. –P. 203–218.

113. *Xu, Y., Kostamaa, P., Zhou, X., and Chen, L.* Handling data skew in parallel joins in shared-nothing systems // ACM SIGMOD international Conference on Management of Data Vancouver, Canada, June 09 – 12, 2008, proceedings. –ACM, –2008. P. 1043–1052.