ПРОГРАММИРОВАНИЕ, 2015, № 6, с. 18–32

БАЗЫ ДАННЫХ И БАЗЫ ЗНАНИЙ

УДК 004.65

ВНЕДРЕНИЕ ФРАГМЕНТНОГО ПАРАЛЛЕЛИЗМА В СУБД С ОТКРЫТЫМ КОДОМ *

© 2015 г. К.С. Пан, М.Л. Цымблер,
Южно-Уральский государственный университет
454080 Челябинск, пр. Ленина, 76
E-mail: pan@susu.ru, mzym@susu.ru
Поступила в редакцию 15.05.2014

В работе представлен оригинальный подход к параллельной обработке сверхбольших баз данных, основанный на внедрении фрагментного параллелизма в имеющиеся последовательные СУБД, свободно распространяемые на уровне исходных кодов. Описана архитектура и методы реализации параллельной СУБД, полученной посредством внедрения фрагментного параллелизма в свободную СУБД PostgreSQL. Приведены результаты вычислительных экспериментов, подтверждающих эффективность предложенного подхода.

1. ВВЕДЕНИЕ

В настоящее время феномен *Больших Данных (Big Data)* является одним из основных факторов, оказывающих существенное влияние на область технологий обработки данных. В условиях современного информационного общества имеется широкий спектр приложений (социальные сети, электронные библиотеки, геоинформационные системы и др.), в каждом из которых производятся неструктурированные данные, имеющие сверхбольшие объемы и высокую скорость прироста (от 1 Терабайта в день). Процессы очистки и структурирования Больших Данных приводят, в свою очередь, к появлению *сверхбольших реляционных баз данных (very large databases)*, требующих параллельной обработки.

Сегодня *параллельные системы баз данных* [1], обеспечивающие обработку запросов на многопроцессорных и многоядерных вычислительных системах, признаются научным сообществом как фактически единственное эффективное средство для организации хранения и обработки сверхбольших баз данных. Базисной концепцией параллельных систем баз данных является *фрагментный параллелизм* [2], предполагающий разбиение отношений базы данных на горизонтальные фрагменты, которые могут обрабатываться независимо на разных узлах кластерной вычислительной системы.

Однако существующие сегодня коммерческие СУБД на основе фрагментного параллелизма (Teradata [3], Greenplum [4], DB2 Parallel Edition [5] и др.), имеют высокую стоимость и, во многих случаях, ориентированы на специфические аппаратно-программные платформы.

Данное обстоятельство является мотивом появления *кластерных СУБД* [6], которые реализую параллельную обработку сверхбольших баз данных на вычислительных системах с кластерной архитектурой на основе использования программного обеспечения (ПО) промежуточного уровня. Кластерная СУБД, ориентированная на приложения класса OLTP (Online Transaction Processing), обрабатывает большое количество коротких транзакций и использует промежуточное ПО для обеспечения межтранзакционного параллелизма. Подключающиеся к системе клиенты распределяются для обработки нескольким экземплярам СУБД, что позволяет повысить доступность системы при большом количестве клиентов. В случае, когда кластерная СУБД ориен-

*Работа выполнена при финансовой поддержке Минобрнауки РФ в рамках ФЦП "Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2014–2020 годы" (Госконтракт № 14.574.21.0035).

тирована на приложения класса OLAP (Online Analytical Proicessing) и выполняет сложные запросы на выборку из баз данных большого объема, промежуточное ПО обеспечивает внутризапросный параллелизм, осуществляя прием запросов пользователя, их преобразование и распределение на узлы кластера, слияние частичных результатов и передачу их пользователю.

Кластерная СУБД MySQL Cluster [7] для OLTP-приложений построена на основе подключения к СУБД MySQL модуля NDB, который позволяет хранить данные в памяти множества распределенных вычислительных узлов с учетом фрагментации и репликации. В СУБД MySQL Cluster масштабируемость ограничена 48 узлами и не поддерживаются базы данных размером более 3 Терабайт. В кластерной СУБД Oracle Real Application Clusters [8] поддерживается хранение до трех реплик базы данных, вследствие чего обеспечивается высокая готовность данных и балансировка нагрузки между узлами кластера, однако данная СУБД не может быть масштабирована более чем на 100 вычислительных узлов.

Кластерная СУБД для OLAP-приложений реализована в исследовательском проекте ParGRES [9]. Эксперименты показывают хорошую масштабируемость данной системы, однако определенным недостатком можно считать использование в ней полной репликации всех таблиц базы данных на узлах вычислительного кластера. Система vParNDB [10] представляет собой ПО промежуточного слоя, которое переписывает запросы таким образом, чтобы они выполнялись параллельно с использованием вычислительных узлов, на которых установлена СУБД MySQL Cluster. Эксперименты показывают хорошее ускорение с использованием такого подхода, однако решения на основе MySQL Cluster наследуют вышеупомянутые ограничения данной СУБД.

В настоящее время *СУБД с открытым кодом* [11] являются надежной альтернативой коммерческим СУБД [12]. В то же время имеется дефицит свободных СУБД, реализующих фрагментный параллелизм. В работе [13] описан прототип параллельной СУБД с открытым кодом для кластерных вычислительных систем. СУБД HadoopDB [14] представляет собой архитектурный гибрид вычислительной парадигмы MapReduce [15] и технологий реляционных СУБД. В СУБД HadoopDB фреймворк Hadoop [16] реализует MapReduce-вычисления и обеспечивает коммуникационную инфраструктуру, объединяющую узлы кластера, на которых выполняются экземпляры СУБД PostgreSQL. SQL-запросы пользователя транслируются в задания для среды MapReduce, которые далее передаются в экземпляры СУБД.

Немногочисленность свободных СУБД на основе фрагментного параллелизма объясняется тем, что параллельная СУБД относится к классу сложного системного программного обеспечения, разработка которого требует существенных финансовых и временны́х затрат.

В связи с этим перспективной можно считать идею модернизации существующего исходного кода свободной последовательной СУБД для построения на ее основе параллельной СУБД путем внедрения фрагментного параллелизма. При этом модернизация исходного кода подразумевает отсутствие масштабных изменений в реализации существующих подсистем, которые равнозначны разработке параллельной СУБД "с нуля". Коммерческие параллельные СУБД, ориентированные на специализированные аппаратные платформы, покажут ожидаемо бо́льшую эффективность в сравнении с параллельной СУБД для кластеров, полученной посредством модернизации исходного кода последовательной СУБД. Однако последняя потенциально способна показать сравнимую с коммерческими параллельными СУБД масштабируемость, достигаемую путем добавления в кластер новых вычислительных узлов, но оставаясь при этом финансово менее затратным решением.

В настоящей работе предлагается подход к организации параллельной обработки сверхбольших баз данных, основной идеей которого является модернизация существующего исходного кода свободной последовательной СУБД для построения на ее основе параллельной СУБД для кластерных вычислительных систем путем внедрения фрагментного параллелизма.

Статья организована следующим образом. В разделе 2 представлен подход к разработке параллельной СУБД, предполагающий модифика-

цию исходных текстов свободной последовательной СУБД для внедрения в нее концепции фрагментного параллелизма. Раздел 3 содержит описание архитектуры и методов реализации параллельной СУБД, полученной посредством применения данного подхода к свободной СУБД PostgreSQL. В разделе 4 приведены результаты вычислительных экспериментов, исследующих эффективность разработанных методов. В заключении суммируются основные результаты, полученные в данной статье, и намечаются направления дальнейших исследований.

2. МЕТОДЫ ВНЕДРЕНИЯ ФРАГМЕНТНОГО ПАРАЛЛЕЛИЗМА

В данном разделе описан комплекс методов, позволяющих осуществить внедрение фрагментного параллелизма в свободную СУБД с открытым исходным кодом.

2.1. *Тиражирование запроса*

Тиражирование запроса предполагает отправку данного запроса множеству экземпляров последовательной СУБД, каждый из которых обрабатывает свой собственный фрагмент базы данных (см. рис. 1).

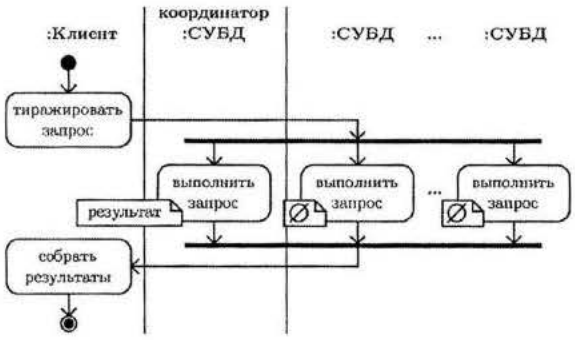

Рис. 1. Тиражирование запроса.

Один из экземпляров СУБД (например, запускаемый на узле кластера с нулевым номером) объявляется *координатором*. Выполнение запроса организуется таким образом, чтобы за исключением координатора, все экземпляры возвратили пустой результат, при этом до завершения выполнения запроса передав свой частичный результат координатору. Координатор осуществляет объединение частичных результатов и отправку их клиенту. В случае возникновения сбоя при выполнении запроса у одного из экземпляров координатор возвращает ошибку в качестве итогового результата.

2.2. *Параллельный план запроса и оператор обмена*

Несмотря на то, что каждый экземпляр СУБД в процессе выполнения запроса независимо обрабатывает свой фрагмент базы данных, для получения корректного результата необходимо выполнять пересылки кортежей. Например, при выполнении операции естественного соединения двух отношений по общему атрибуту кортежи, для которых условие соединения выполняется, могут храниться в разных фрагментах базы данных. Для обработки подобных ситуаций строится *параллельный план запроса*, который представляет собой последовательный план, в нужные места которого вставляется специальный оператор обмена.

Оператор обмена (*exchange*) [17] инкапсулирует в себе передачу кортежей между экземплярами СУБД на различных узлах кластерной вычислительной системы. Оператор обмена реализуется аналогично другим операторам физической алгебры в модифицируемой СУБД, которые имеют итераторный интерфейс. Оператор обмена имеет два дополнительных свойства: порт и функцию пересылки ψ. Свойство *порт* позволяет отличать операторы обмена в одном плане запроса друг от друга: кортежи из одной точки плана запроса должны попадать в ту же точку плана на другом вычислительном узле. *Функция пересылки* $\psi(t)$ вычисляет номер узла, на котором должен быть обработан данный кортеж t. Если кортеж t требуется на локальном узле, то он передается выше по плану, иначе — пересылается на узел с номером $\psi(t)$.

На рис. 2 приведена структура оператора обмена (стрелки показывают направление передачи кортежей). Составляющие оператор обмена операторы *split*, *scatter*, *gather* и *merge* также реализуются на основе итераторной модели.

Оператор *split* представляет собой бинарный оператор, который разделяет поступающие из входного потока кортежи на две категории: "свои" и "чужие". "Свои" кортежи должны быть обработаны на текущем вычислительном узле и направляются в выходной буфер оператора *split*.

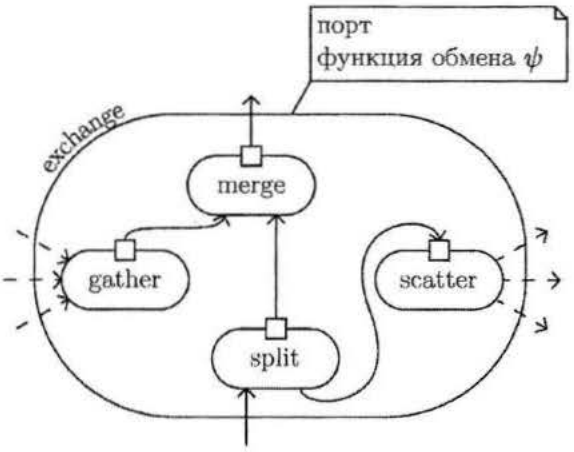

Рис. 2. Архитектура оператора обмена.

"Чужие" кортежи должны быть обработаны на вычислительных узлах, отличных от текущего, и помещаются оператором *split* в выходной буфер оператора *scatter*.

Оператор *scatter* определяется как нульарный оператор, который извлекает кортежи из своего выходного буфера, вычисляет для них значение функции пересылки и пересылает их на соответствующие вычислительные узлы, используя заданный номер порта обмена.

Оператор *gather* представляет собой нульарный оператор, который выполняет чтение кортежей из указанного порта обмена со всех вычислительных узлов, отличных от данного, в свой выходной буфер.

Оператор *merge* определяется как бинарный оператор, который поочередно забирает кортежи из выходных буферов своих сыновей и помещает их в собственный выходной буфер.

Оригинальный исполнитель запросов последовательной СУБД выполняет оператор обмена как и любой другой, не подразумевая никакого параллелизма. Параллелизм достигается за счет работы параллелизатора запросов, который должен разместить операторы *exchange* в нужные места плана запросов так, чтобы существующая логика исполнителя запросов привела к корректному результату.

2.3. *Добавление в словарь СУБД метаданных о фрагментации*

Способ фрагментации реляционного отношения определяется функцией фрагментации, ассоциированной с данным отношением. *Функция фрагментации* для каждого кортежа отношения вычисляет номер вычислительного узла, на котором должен быть размещен этот кортеж. Чтобы предоставить экземпляру СУБД информацию о фрагментации таблиц, язык баз данных должен быть расширен синтаксическими средствами, которые позволят указать функцию фрагментации при выполнении команды `CREATE TABLE`, а словарь СУБД необходимо дополнить метаданными о фрагментации отношений.

2.4. *Параллельный план запросов на изменение данных*

Рассмотренная выше схема построения параллельного плана работает корректно для запросов на выборку данных из отношений, фрагменты которых распределены по вычислительным узлам кластерной системы. Однако в эту схему необходимо внести изменения для того, чтобы обеспечить корректное выполнение запросов на вставку и обновление данных (см. рис. 3). При выполнении запроса `INSERT` необходимо обеспечить вставку кортежа только в один из фрагментов, несмотря на то, что данный запрос подвергается тиражированию. При обработке запроса `UPDATE` обновленные кортежи, для которых функция фрагментации выдает значение, отличное от номера текущего узла, должны быть перемещены и соответствующий вычислительный узел.

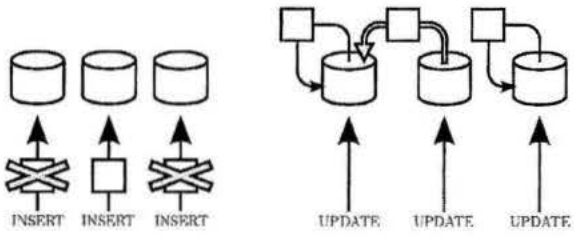

Рис. 3. Вставка и обновление кортежа.

2.5. *Прозрачное портирование приложений оригинальной СУБД*

Пользовательские приложения, написанные для оригинальной СУБД с окрытым кодом, должны требовать минимальных изменений в

исходном тексте, чтобы обеспечить возможность их запуска в параллельной СУБД, разработанной на ее основе. Прозрачное портирование приложений оригинальной СУБД в параллельную СУБД реализуется посредством разработки библиотеки прикладного программиста, интерфейс которой *идентичен* интерфейсу оригинальной библиотеки. Новая библиотека реализует тиражирование запроса путем многократного вызова функций из оригинальной библиотеки и, имея идентичный оригинальной библиотеке интерфейс, обеспечивает прозрачную работу приложения с параллельной СУБД. Таким образом, при переходе от последовательной СУБД к параллельной в коде приложения требуется изменить лишь название подключаемой библиотеки прикладного программиста.

```
// origfile.c
#include "newfile.c"
typedef struct origstruct {
        ...
        newstruct ns;
} origstruct;

int origfunc() {
        ...
        newfunc();
        ...
}

// newfile.c
typedef struct newstruct {
        ...
} newstruct;

int newfunc() {
    ...
}
```

Рис. 4. Добавление полей в структуру и вызова в функцию.

2.6. *Мягкая модификация исходных текстов оригинальной СУБД*

СУБД представляет собой сложное системное программное обеспечение, исходные тексты которого исчисляются десятками тысяч строк. Отсутствие технологической дисциплины при модификации исходных текстов подобных систем может иметь фатальные последствия для проекта.

Предлагаемая техника модификации исходных текстов позволяет минимизировать вносимые в код изменения, инкапсулировав новый код в отдельных подсистемах. Дополнения в структуры данных и алгоритмы инкапсулируются в *новых* файлах исходных текстов, подключаемых к исходным текстам оригинальной СУБД.

На рис. 4 показан пример применения данной техники. При добавлении новых полей в оригинальную структуру данных в новом файле описывается тип `newstruct`, содержащий новые поля, а в оригинальную структуру добавляется новое поле, имеющее тип данных `newstruct`. При изменении оригинальных алгоритмов в тело оригинальной функции добавляется вызов новой функции `newfunc()`, а сама функция `newfunc()` определяется в файле исходных текстов новой подсистемы.

3. ВНЕДРЕНИЕ ПАРАЛЛЕЛИЗМА В СУБД POSTGRESQL

В данном разделе описано применение предложенных методов к СУБД PostgreSQL [18], которая в настоящее время является одной из наиболее популярных СУБД с открытым кодом [9]. Выбор PostgreSQL обусловлен наличием свободно доступных детальных внутренних спецификаций и качественной документации для программистов данной СУБД. Объем разработанных авторами исходных текстов составил около 5 тысяч строк кода и занял около трех человеко-месяцев. Полученная в результате параллельная СУБД получила название PargreSQL [19, 20].

3.1. *Архитектура параллельной СУБД PargreSQL*

Архитектура параллельной СУБД PargreSQL представлена на рис. 5.

Оригинальная СУБД PostgreSQL рассматривается в качестве одной из подсистем параллельной СУБД. Структура СУБД PostgreSQL кратко может быть описана следующим образом.

Подсистема `Parser` осуществляет разбор запроса. Подсистема `Rewriter` выполняет преобразование текста запроса в соответствии с правилами, определенными администратором (например, замена имен представлений на их определения). Подсистема `Planner` обеспечивает

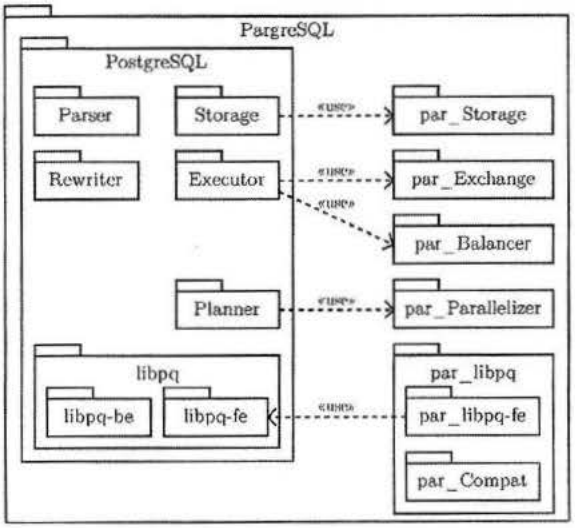

Рис. 5. Архитектура СУБД PargreSQL.

создание и оптимизацию плана запроса. Подсистема `Executor` исполняет план. Подсистема `Storage` обеспечивает низкоуровневое хранение данных и метаданных. Библиотека `libpq` представляет собой интерфейс прикладного программиста СУБД PostgreSQL, реализуя протокол взаимодействия клиента (`libpq-fe`) и сервера (`libpq-be`).

В сеансе работы с СУБД PostgreSQL участвуют три вида взаимодействующих процессов (см. рис. 6): *Frontend* (приложение-клиент), *Daemon* (демон) и *Backend* (серверный процесс). Демон осуществляет прием соединений, устанавливаемых клиентами, и создает отдельный серверный процесс для обработки запросов каждого отдельного клиента.

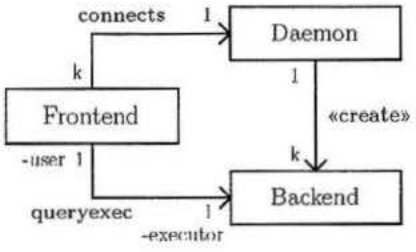

Рис. 6. Клиент-серверная схема в PostgreSQL.

Остальные подсистемы параллельной СУБД PargreSQL реализуют методы, описанные в разделе 2. Подсистема `par_libpq` реализует тиражирование запроса. Подсистема `par_Compat` представляет собой набор макроподстановок и обеспечивает прозрачное портирование приложений в новую СУБД. Подсистемы `par_Parallelizer` и `par_Exchange` реализуют соответственно построение параллельного плана запроса и оператор обмена. Подсистема `par_Storage` отвечает за хранение метаданных о фрагментации таблиц. Подсистема `par_Balancer` предназначена для обеспечения балансировки загрузки во время исполнения запроса.

Схема клиент-серверного взаимодействия параллельной СУБД PargreSQL представлена на рис. 7.

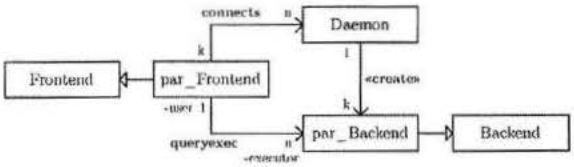

Рис. 7. Процессы СУБД PargreSQL.

В отличие от последовательной СУБД PostgreSQL клиент может взаимодействовать с двумя и более серверами одновременно.

Реализация компонентов `par_Backend` и `par_Frontend` осуществляется на основе оригинальных компонентов `Backend` и `Frontend` СУБД PostgreSQL соответственно. Компонент `Backend` расширяется для обеспечения обменов кортежами между экземплярами СУБД, а компонент `Frontend` дополняется функцией тиражирования запросов.

3.2. *Реализация тиражирования запросов*

Порядок взаимодействия клиентского приложения и СУБД PargreSQL представлен на рис. 8.

Взаимодействие осуществляется следующим образом. Клиентское приложение подключается последовательно ко всем демонам СУБД. Результатом подключения является запуск `par_Backend` на каждом узле. Затем клиент последовательно отправляет запрос каждому из этих компонентов. Получив запрос, каждый экземпляр `par_Backend` выполняет его над своим фрагментом базы данных, при этом,

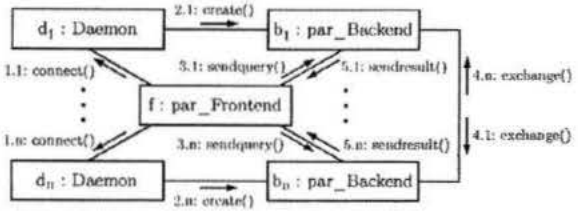

Рис. 8. Взаимодействие клиента и серверов PargreSQL.

возможно, обмениваясь данными с другими экземплярами с помощью оператора *exchange*. По завершении обработки запроса клиентское приложение получает от экземпляров результаты и агрегирует их.

3.3. *Реализация построения параллельного плана запроса*

Для построения параллельного плана запроса используется следующая техника [17]. Осуществляется концевой обход дерева последовательного плана и вставка оператора *exchange* ниже узла соединения в том случае, если атрибут фрагментации сына не совпадает с атрибутом, по которому производится соединение. При этом атрибут фрагментации распространяется по дереву от дочерних узлов к родительским. Таким образом, в каждой точке плана известно, по какому атрибуту фрагментирован результат операции. Соответствующие случаи, которые требуют вставки оператора обмена, представлены на рис. 9.

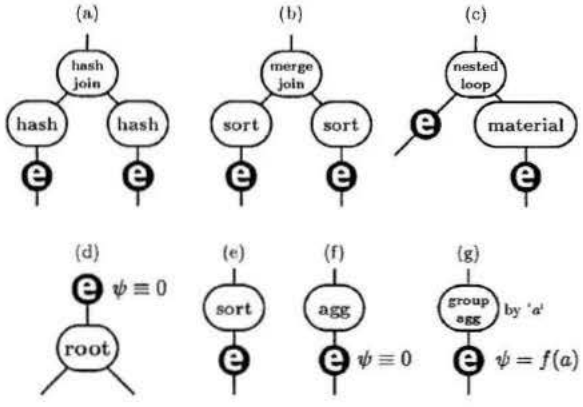

Рис. 9. Вставка оператора *exchange*.

При формировании плана запроса в СУБД PostgreSQL используются следующие типы узла, реализующего операцию соединение: соединение хешированием (HashJoin) [21], соединение слиянием (MergeJoin) [22] и соединение вложенными циклами (NestedLoop) [23]. Вставка оператора обмена в каждом из этих случаев имеет следующие особенности.

Операция HashJoin предполагает формирование хеш-таблицы для каждого из отношений, участвующих в соединении. Узел *HashJoin* имеет два узла-потомка типа *Hash*, каждый из которых выполняет создание хеш-таблицы для своего сына. Вставка оператора обмена осуществляется между узлом *Hash* и его поддеревом (см. рис. 9a), чтобы создание хеш-таблицы осуществлялось после того, как будут получены кортежи, отправленные другими вычислительными узами с помощью оператора *exchange* на текущий вычислительный узел.

Операция MergeJoin предполагает предварительную сортировку отношений, участвующих в соединении. Узел *MergeJoin* имеет два узла-потомка типа *Sort*, каждый из которых выполняет сортировку данных своего сына. Вставка оператора обмена осуществляется между узлом *Sort* и его поддеревом (см. рис. 9b), чтобы сортировка осуществлялась после того, как будут получены кортежи, отправленные другими вычислительными узами с помощью оператора *exchange* на текущий вычислительный узел.

Операция NestedLoop предполагает, что правое отношение полностью загружено в память для осуществления его многократного сканирования во внутреннем цикле соединения. Правый сын узла *NestedLoop* — это узел *Material*, выполняющий загрузку данных своего сына в память. Вставка оператора обмена осуществляется между узлом *Material* и его поддеревом (см. рис. 9c), чтобы загрузка данных в память осуществлялась после того, как будут получены кортежи, отправленные другими вычислительными узами с помощью оператора *exchange* на текущий вычислительный узел. Вставка оператора обмена над узлом *Material* приведет к тому, что пересылка кортежей правого отношения будет осуществляться столько раз, сколько кортежей в левом отношении. Это приводит к взаимной блокировке в случае, когда фрагменты ле-

вого отношения на разных узлах кластера содержат различное количество кортежей.

На рис. 9d показана вставка оператора обмена в корень плана запроса. В данном случае оператор обмена обеспечивает слияние частичных результатов запроса, полученных на различных вычислительных узлах кластера, на вычислительном узле-координаторе. Оператор обмена, вставляемый в корень плана запроса, имеет функцию пересылки, тождественно равную номеру узла-координатора.

Для получения корректного параллельного плана, помимо вставки оператора обмена в случае операции соединения отношений, требуется также вставка оператора обмена при обработке узлов плана, выполняющих сортировку и агрегацию кортежей.

Операция Sort используется для сортировки поступающих из поддерева кортежей, и, если поместить сразу над ней операцию *exchange*, порядок кортежей нарушится, и сортировка будет выполнена впустую. Поэтому в таких случаях (см. рис. 9e) операция *exchange* сдвигается на уровень ниже — под узел *Sort*. Таким образом, сортировка выполняется после обменов, и в итоге получается корректный результат.

Операция Agg используется для вычисления агрегирующих функций без группировки в запросах вида `select sum(a) from t`. Поскольку операция агрегации должна обработать кортежи, находящиеся во всех фрагментах отношения, то для получения корректных результатов под узел *Agg* вставляется операция *exchange* (см. рис. 9f) с функцией обмена, тождественно равной номеру вычислительного узла-координатора. Это обеспечивает пересылку всех кортежей на один узел и корректный подсчет значения агрегирующей функции.

Операция GroupAgg используется для вычисления агрегирующих функций с группировкой в запросах вида `select a, sum(b) from t group by a`. В отличие от предыдущего случая, для правильного выполнения данной операции достаточно обработать каждую отдельную группу кортежей целиком. Поэтому для получения корректных результатов под узел *Agg* вставляется операция *exchange* с функцией обмена, зависящей от атрибута группировки (см. рис. 9g). Это обеспечивает пересылку всех кортежей из одной группы на один узел и, соответственно, корректное вычисление агрегирующей функции для каждой группы.

3.4. *Реализация оператора обмена* exchange

Реализация оператора *exchange* предполагает внедрение в оригинальную СУБД PostgreSQL новых функций и типов данных. На рис. 10 представлен пакет *par_Exchange*, содержащий новые классы, добавляемые в СУБД PostgreSQL.

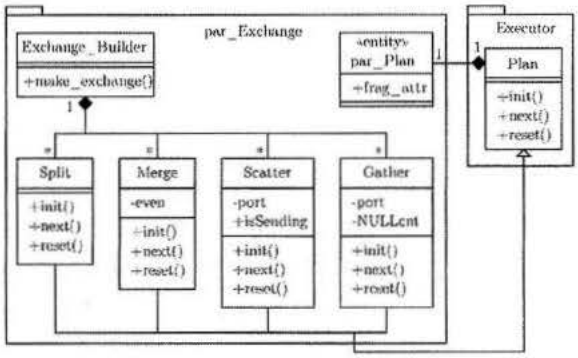

Рис. 10. Диаграмма классов оператора *exchange*.

Классы данного пакета *Merge*, *Split*, *Scatter* и *Gather* реализуют одноименные узлы оператора *exchange*. Класс *Exchange_Builder* выполняет функции строителя, предоставляя метод для создания вышеперечисленных узлов плана и формирования из них цельного оператора *exchange*.

Для хранения атрибута фрагментации отношения необходимо выполнить модификацию класса *Plan* в СУБД PostgreSQL, который представляет собой узел плана запроса: в данный класс необходимо добавить целочисленный атрибут `frag_attr`.

Алгоритм реализации метода `next` узла *Split* (см. рис. 11) состоит в следующем. Узел *Split* вызывает метод `next` левого сына и применяет к полученному от него результирующему кортежу функцию пересылки. Если функция пересылки показывает, что данный кортеж "свой" (значение функции совпадает с номером текущего вычислительного узла), то узел *Split* завершает работу, возвращая значение этого кортежа в качестве результата. В противном случае данный кортеж помещается в буфер правого сына (узел *Scatter*), осуществляется вызов метода `next` узла

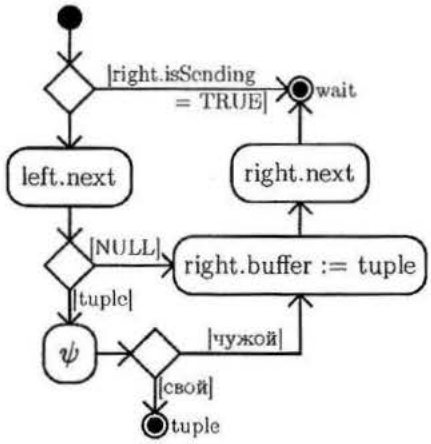

Рис. 11. Метод next узла *Split*.

Scatter, и оператор *exchange* переводится в состояние ожидания.

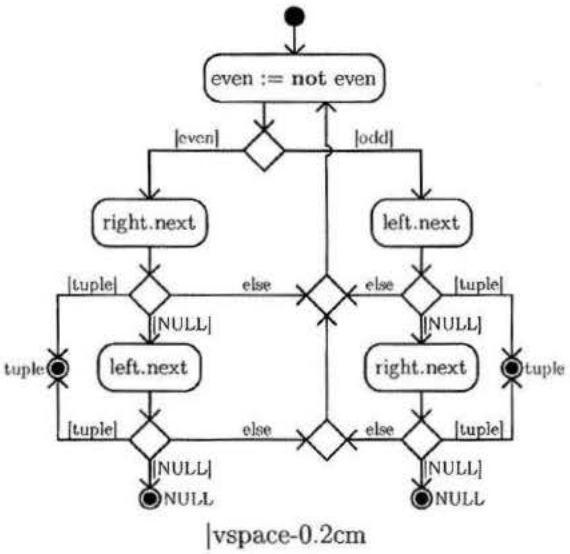

Рис. 12. Метод next узла *Merge*.

На рис. 12 представлен алгоритм метода **next** узла *Merge*. Узел *Merge* попеременно вызывает методы **next** своего левого и правого сыновей — узлов *Gather* и *Split*. Вызовы осуществляются, пока оператор *exchange* находится в состоянии ожидания. Если оба сына вернули пустое значение NULL, это означает, что входной поток кортежей исчерпан, и узел *Merge* завершает работу, возвращая значение NULL. Если хотя бы один из сыновей возвратил в качестве результата кортеж, то происходит завершение работы узла *Merge*, и этот кортеж возвращается как результат работы.

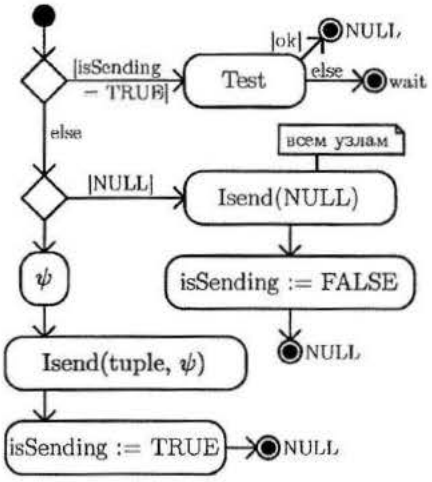

Рис. 13. Метод next узла *Scatter*.

Алгоритм реализации метода **next** узла *Scatter* показан на рис. 13. Оператор *Scatter* не имеет дочерних операторов, и вызов его метода **next** инициирует отправку кортежа, переданного ему от родительского оператора (*Split*), на вычислительный узел, номер которого совпадает с результатом применения к кортежу функции обмена. Если во время вызова метода **next** отправка кортежа не завершена, то возвращается значение **WAIT**.

Оператор *Gather* (см. рис. 14) инициирует получение кортежей от всех вычислительных узлов. При вызове метода **next** данного оператора проверяются статус операций получения, и если получен кортеж от некоторого узла, то инициируется новая операция получения от данного узла, а полученный кортеж возвращается в качестве результата. Если от всех узлов получено значение NULL вместо кортежа, значит, отношение исчерпано, и метод возвращает NULL в качестве признака конца отношения.

Для реализации описанных выше операторов *scatter* и *gather* в СУБД PargreSQL на основе технологии MPI (Message Passing Interface, интерфейс обмена сообщениями) [24] разработан *менеджер собщений*. Реализация обменов сообщениями на базе MPI является стандартным приемом для систем с распределенной памятью, однако в

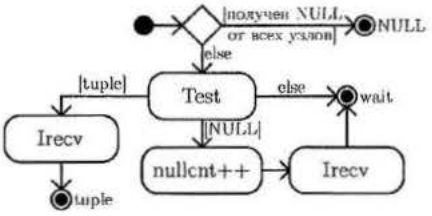

Рис. 14. Метод next узла *Gather*.

```
create table Person (
    id int,
    name varchar(30),
    gender char(1),
    birth date
) with (fragattr = id);
```

Рис. 15. Создание таблиц в PargreSQL.

случае СУБД PargreSQL прямое использование MPI затруднено, поскольку архитектура данной СУБД предполагает динамическое порождение серверных процессов.

Менеджер сообщений состоит из двух модулей: коммуникатор и библиотека. *Коммуникатор* представляет собой MPI-программу и запускается в виде независимого демона в одном экземпляре на каждом вычислительном узле. *Библиотека* предоставляет серверным процессам интерфейс для подключения к коммуникатору через общую память и организует обмен сообщениями. Библиотека менеджера сообщений включает в себя следующие основные функции: "начать операцию отправки данных", "начать операцию приема данных", "проверить завершение операции", — которые имеют интерфейс и семантику, схожие с асинхронными функциями `MPI_Isend`, `MPI_Irecv` и `MPI_Test` соответственно.

3.5. *Реализация хранения метаданных о фрагментации*

Чтобы реализовать поддержку фрагментации данных, в СУБД PostgreSQL в метаданные таблиц вводится новый атрибут *fragattr*. Данный атрибут имеет строковый тип и задает имя столбца, от которого зависит функция фрагментации соответствующей таблицы. При создании таблицы значение данного атрибута должно быть задано явно. Атрибут *fragattr* указывается в запросе `CREATE TABLE` с помощью существующей в СУБД PostgreSQL конструкции `WITH`. Пример запроса приведен на рис. 15.

Атрибут под именем, которое указано в параметре *fragattr* таблицы, будет использован при обработке запросов `UPDATE` и `INSERT` для обеспечения фрагментации с функцией фрагментации $\varphi(t) = t.\text{fragattr} \bmod N$, где N — количество вычислительных узлов в кластерной системе, а mod — операция взятия остатка от деления.

3.6. *Реализация запросов на изменение данных*

При обработке запросов на вставку данных необходимо добавлять в корень плана запроса операцию выборки с условием $\psi(t) = i$, где i — номер текущего узла (см. рис. 16).

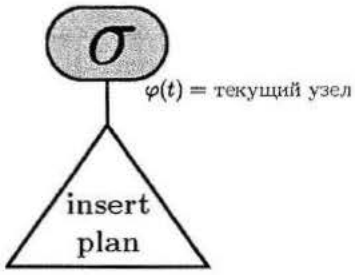

Рис. 16. Параллельный план запроса INSERT.

Такое условие отсечет все кортежи, которые должны быть вставлены на другие вычислительные узлы. Таким образом, каждый вставляемый в базу данных кортеж попадет только в один фрагмент базы данных.

Для перемещения измененных кортежей в алгоритм работы оператора обмена нужно внести изменения. Модифицированный оператор обмена (см. рис. 17) должен обнаруживать кортежи, у которых изменилось значение атрибута фрагментации, и создавать копию таких кортежей.

Один экземпляр передается далее по плану с пометкой "удалить", второй экземпляр передается на соответствующий вычислительный узел с пометкой "вставить". Таким образом, модифицированный алгоритм обмена позволяет перемещать кортежи, которые в результате обновления стали "чужими": если $\varphi(t') \neq \varphi(t)$, то на узле с

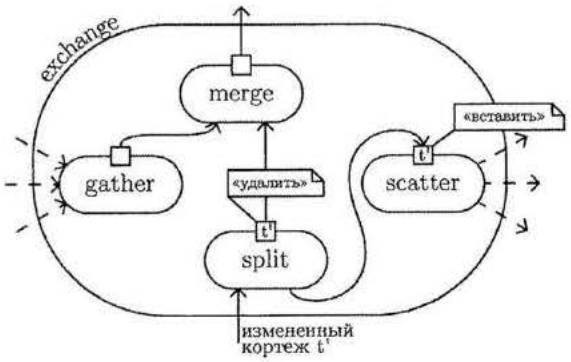

Рис. 17. Поток кортежей в операторе обмена при запросе UPDATE.

номером $\varphi(t)$ кортеж t удаляется, а на узле с номером $\varphi(t')$ вставляется кортеж t'.

4. ВЫЧИСЛИТЕЛЬНЫЕ ЭКСПЕРИМЕНТЫ

С целью оценки эффективности предложенных методов и алгоритмов, реализованных в параллельной СУБД PargreSQL, нами были проведены две серии вычислительных экспериментов. Первая серия экспериментов была направлена на исследование масштабируемости СУБД PargreSQL. Во второй серии экспериментов осуществлялось сравнение производительности СУБД PargreSQL с производительностью имеющихся в настоящее время аналогичных систем. Аппаратной платформой экспериментов послужил суперкомпьютер "СКИФ-Аврора ЮУрГУ" [25], характеристики которого представлены в табл. 1.

4.1. *Масштабируемость*

Масштабируемость представляет собой меру эффективности распараллеливания алгоритма для аппаратных платформ с различным количеством вычислительных узлов. В случае параллельных систем баз данных основными качественными характеристиками эффективности распараллеливания считаются ускорение и расширяемость, которые показывают возможность системы адаптироваться соответственно к увеличению количества узлов кластера и возрастанию объема обрабатываемых данных. Данные характеристики определяются следующим образом [26].

Пусть A и B — две различные конфигурации параллельной машины баз данных с фиксированной архитектурой, различающиеся количеством процессоров и ассоциированных с ними устройств (при этом все конфигурации предполагают пропорциональное наращивание модулей памяти и дисков) и задан некоторый тест Q. Тогда *ускорение* a_{AB}, получаемое при переходе от конфигурации A к конфигурации B, определяется формулой $a_{AB} = t_{QA}/t_{QB}$, где t_{QA} и t_{QB} — это время, затраченное конфигурациями A и B соответственно на выполнение теста Q. Ускорение позволяет определить эффективность наращивания системы на сопоставимых задачах.

Пусть теперь задан набор тестов Q_1, Q_2, ..., количественно превосходящих некоторый фиксированный тест Q в i раз, где i — номер соответствующего теста и конфигурации параллельной машины баз данных A_1, A_2, ..., превосходящие по степени параллелизма (количеству процессоров) некоторую минимальную конфигурацию A в j раз, где j — номер соответствующей конфигурации. Тогда *расширяемость* e_{km}, получаемая при переходе от конфигурации A_k к конфигурации A_m ($k < m$), определяется формулой $e_{km} = t_{Q_kA_k}/t_{Q_mA_m}$. Расширяемость позволяет измерить эффективность наращивания системы на бóльших задачах.

Говорят, что параллельная система хорошо масштабируема, если она демонстрирует ускорение и расширяемость, близкие к линейным. *Линейное ускорение* означает, что существует константа $k > 0$ такая, что $a_{AB} = kd_B/d_A$ для любых конфигураций A и B (где d — количество процессоров в соответствующей конфигурации). *Линейная расширяемость* означает, что расширяемость остается равной единице для всех конфигураций данной системной архитектуры.

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

Результаты данных экспериментов представлены на рис. 18: СУБД PargreSQL демонстрирует ускорение, близкое к линейному.

В экспериментах, исследовавших расширя-

Таблица 1. Аппаратная платформа экспериментов

Характеристика	Значение
Число узлов/процессоров/ядер	736/1472/8832
Тип процессора	Intel Xeon X5680
Оперативная память	3 Тб
Производительность пиковая	117 TFlops
Производительность LINPACK	100.4 TFlops

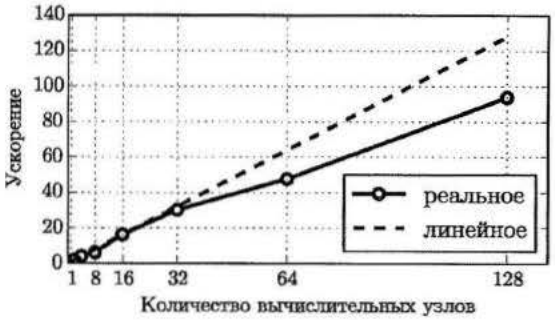

Рис. 18. Ускорение.

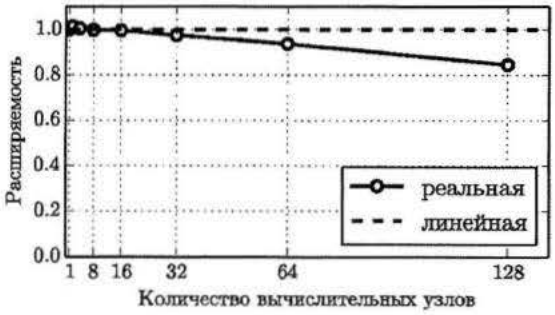

Рис. 19. Расширяемость СУБД PargreSQL.

емость, СУБД PargreSQL исполняла запрос, предполагающий выполнение операции естественного соединения двух отношений по общему атрибуту. Кортежи отношений равномерно распределены по узлам кластера. Размеры соединяемых отношений увеличивались пропорционально увеличению количества используемых узлов кластера с множителем 12 млн. и 0.3 млн. кортежей соответственно.

Результаты данных экспериментов отображены на рис. 19: расширяемость СУБД PargreSQL близка к линейной.

Таким образом, результаты экспериментов показывают, что СУБД PargreSQL демонстрирует масштабируемость, близкую к линейной.

4.2. *Тест TPC*

Стандартный тест TPC-C разработан консорциумом TPC (Transaction Processing Council) [27] для измерения производительности СУБД в ходе обработки смеси коротких транзакций. В рамках теста осуществляется моделирование деятельности типичного склада (прием заказов, управление учетом и распространением товаров и др.). В качестве меры производительности в тесте TPC-C используется коммерческая пропускная способность, отражающая количество обработанных в минуту заказов. Мера производительности выражается пиковой скоростью выполнения транзакций tpm-C (transactions-per-minute-C, количество транзакций в минуту).

В тесте использовалось от 1 до 30 параллельно работающих клиентов, выполняющих запросы к СУБД PargreSQL, запущенной на 12 узлах кластерной вычислительной системы. Размер базы данных составлял 12 "складов". Результаты исследования эффективности СУБД PargreSQL на тесте TPC-C приведены в табл. 2 в порядке убывания показателя tpm-C.

Данный результат позволил СУБД PargreSQL попасть в пятерку лидеров рейтинга TPC-C среди параллельных СУБД для кластеров на сентябрь 2013 г. (см. табл. 3).

Мы можем заключить, что параллельная СУБД PargreSQL представляет собой эффективное и относительно недорогое решение для организации хранения и обработки сверхбольших объемов данных, обладающее хорошей масштабируемостью.

5. ЗАКЛЮЧЕНИЕ

В данной работе была рассмотрена проблема организации обработки сверхбольших баз дан-

Таблица 2

К-во клиентов	tpm-C ↓	К-во клиентов	tpm-C ↓	К-во клиентов	tpm-C ↓	К-во клиентов	tpm-C ↓
29	2202531	24	2165413	16	1882353	8	1156626
26	2197183	23	2156250	15	1747572	7	1150684
30	2195122	22	2146341	14	1647058	5	857142
32	2194285	20	2068965	13	1529411	6	847058
27	2189189	19	2054054	12	1358490	4	657534
31	2188235	18	2037735	11	1346938	3	444444
28	2181818	21	2016000	10	1290322	2	328767
25	2173913	17	1961538	9	1270588	1	150000

Таблица 3

	Кластер / СУБД	К-во узлов / клиентов		tpm-C
1	SPARC SuperCluster with T3-4 Servers / Oracle Database 11g R2 Enterprise Edition w/RAC w/Partitioning	108	81	30 249 688
2	IBM Power 780 Server Model 9179-MHB / IBM DB2 9.7	24	96	10 366 254
3	Sun SPARC Enterprise T5440 Server Cluster / Oracle Database 11g Enterprise Edition w/RAC w/Partitioning	48	24	7 646 486
	СКИФ-Аврора ЮУрГУ / PargreSQL	12	29	2 202 531
4	HP Integrity rx5670 Cluster Itanium2/1.5 GHz-64p / Oracle Database 10g Enterprise Edition	64	80	1 184 893

ных на вычислительных системах с кластерной архитектурой. Предложен подход к решению данной проблемы, основанный на модернизации существующего исходного кода свободной последовательной СУБД для построения на ее основе параллельной СУБД для кластерных вычислительных систем путем внедрения фрагментного параллелизма. При этом модернизация исходного кода не является масштабной. Получаемая таким образом параллельная СУБД способна показать хорошую масштабируемость. Недостаток производительности по сравнению с коммерческими параллельными СУБД, ориентированными на специфические аппаратно-программные платформы, может быть преодолен добавлением в кластер новых вычислительных узлов с сохранением экономичности данного решения. Данный подход может быть применен для параллелизации практически любой СУБД с открытым кодом (PostgreSQL, MySQL и др.).

Описана архитектура и методы реализации параллельной СУБД PargreSQL, разработанной авторами на основе предложенного подхода путем внедрения фрагментного параллелизма в свободную СУБД PostgreSQL. Представлены результаты вычислительных экспериментов, показывающих близкие к линейным ускорение и расширяемость СУБД PargreSQL, а также ее приемлемую прозводительность на стандартном тесте TPC-C.

В качестве возможных направлений дальнейших исследований интересными представляются следующие задачи.

1. Внедрение в последовательную СУБД с открытым кодом, наряду с фрагментным параллелизмом, поддержки репликации данных на основе метода частичного зеркалирования данных [28]) и оценок коммуникационных затрат обработки фрагментированных отношений [29], и разработка подсистемы балансировки загрузки получаемой параллельной СУБД.

2. Внедрение в параллельную СУБД, полученную путем модификации исходных текстов последовательной СУБД, эффективных методов управления буферным пулом, ориентированных на параллельные системы баз данных без совместного использования ресурсов [30].

3. Адаптация предложенных методов и алгоритмов для кластерных систем, узлы которых оснащены многоядерными ускорителями, на основе использования модели DMM многопроцессорных систем баз данных [31].

СПИСОК ЛИТЕРАТУРЫ

1. *Соколинский Л.Б.* Обзор архитектур параллельных систем баз данных // Программирование. 2004. N 6. С. 49–63.

2. *Лепихов А.В., Соколинский Л.Б.* Обработка запросов в СУБД для кластерных систем // Программирование. 2010. N 4. С. 25–39.

3. *Page J.* A Study of a Parallel Database Machine and its Performance the NCR/Teradata DBC/1012 // Proceedings of the 10th British National Conference on Databases, BNCOD, Aberdeen, Scotland, July 6–8, 1992. Lecture Notes in Computer Science. Springer. 1992. P. 115–137.

4. *Waas F.M.* Beyond Conventional Data Warehousing — Massively Parallel Data Processing with Greenplum Database // Proceedings of the 2nd International Workshop on Business Intelligence for the Real-Time Enterprise, BIRTE 2008, in conjunction with VLDB'08, August 24, 2008, Auckland, New Zealand. 2008.

5. *Baru C.K., Fecteau G., Goyal A., et al.* An Overview of DB2 Parallel Edition // Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22–25, 1995. ACM, 1995. P. 460–462.

6. *Akal F., Böhm K., Schek H.-J.* OLAP Query Evaluation in a Database Cluster: A Performance Study on Intra-Query Parallelism // Proceedings of the 6th East European Conference on Advances in Databases and Information Systems, ADBIS 2002, Bratislava, Slovakia, September 8–11, 2002. Lecture Notes in Computer Science. Springer, 2002. P. 218–231.

7. *Ronström M., Oreland J.* Recovery Principles in MySQL Cluster 5.1 // Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 – September 2, 2005. ACM, 2005. P. 1108–1115.

8. *Pruscino A.* Oracle RAC: Architecture and Performance // Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9–12, 2003. ACM, 2003. P. 635.

9. *Paes M., Lima A.A.B., Valduriez P., Mattoso M.* High-Performance Query Processing of a Real-World OLAP Database with ParGRES // High Performance Computing for Computational Science, VECPAR 2008: 8th International Conference, Toulouse, France, June 24–27, 2008. Revised Selected Papers. Lecture Notes in Computer Science. Springer, 2008. P. 188–200.

10. *Ngamsuriyaroj S., Pornpattana R.* Performance Evaluation of TPC-H Queries on MySQL Cluster // 24th IEEE International Conference on Advanced Information Networking and Applications Workshops, WAINA 2010, Perth, Australia, April 20–13, 2010. IEEE Computer Society, 2010. P. 1035–1040.

11. *Evdoridis T., Tzouramanis T.* A Generalized Comparison of Open Source and Commercial Database Management Systems // Database Technologies: Concepts, Methodologies, Tools, and Applications. Chapter 23. IGI Global, 2009. P. 294–308.

12. *Paulson L.D.* Open Source Databases Move into the Marketplace // Computer, 2004. V. 37. N 7. P. 13–15.

13. *Гавриш Е.В., Колтаков А.В., Медведев А.А., Соколинский Л.Б.* Параллельная СУБД с открытым исходным кодом для кластерных вычислительных систем // Вестник ЮУрГУ. Серия "Вычислительная математика и информатика". 2013. Т. 2. N 3. С. 81–91.

14. *Abouzeid A., Bajda-Pawlikowski K., Abadi D.J., et al.* HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads // Proceedings of the VLDB Endowment. 2009. V. 2. N 1. P. 922–933.

15. *Dean J., Ghemawat S.* MapReduce: Simplified Data Processing on Large Clusters // Communications of the ACM (CACM). 2008. V. 51. N 1. P. 107–113.

16. *White T.* Hadoop — The Definitive Guide: MapReduce for the Cloud. O'Reilly, 2009. 524 p.

17. *Соколинский Л.Б.* Организация параллельного выполнения запросов в многопроцессорной машине баз данных с иерархической архитектурой // Программирование. 2001. N 6. С. 13–29.

18. *Stonebraker M., Kemnitz G.* The POSTGRES Next-generation Database Management System // Communications of the ACM. 1991. V. 34. N 10. P. 78–92.

19. *Пан К.С., Цымблер М.Л.* Разработка параллельной СУБД на основе последовательной СУБД PostgreSQL с открытым исходным кодом // Вестник ЮУрГУ. Серия "Математическое моделирование и программирование", 2012. N 18(277). Вып. 12. С. 112–120.

20. *Pan C., Zymbler M.* Taming Elephants, or How to Embed Parallelism into PostgreSQL // Database and Expert Systems Applications – 24th International Conference, DEXA 2013, Prague, Czech Republic, August 26–29, 2013. Proceedings, Part I. Springer, 2013. Lecture Notes in Computer Science. V. 8055. P. 153–164.

21. *Zhou J.* Hash Join // Encyclopedia of Database Systems / Ed. by Liu L., Özsu M.T. Springer, 2009. P. 1288–1289.

22. *Zhou J.* Nested Loop Join // Encyclopedia of Database Systems / Ed. by Liu L., Özsu M.T. Springer, 2009. P. 1895.

23. *Zhou J.* Sort-Merge Join // Encyclopedia of Database Systems / Ed. by Liu L., Özsu M.T. Springer, 2009. P. 2673–2674.

24. *Gropp W.* MPI 3 and Beyond: Why MPI Is Successful and What Challenges It Faces // Proceedings of the 19th European MPI Users' Group Meeting, EuroMPI 2012, Vienna, Austria, September 23–26, 2012. Lecture Notes in Computer Science. Springer, 2012. P. 1–9.

25. *Московский А.А., Перминов М.П., Соколинский Л.Б., Черепенников В.В., Шамакина А.В.* Исследование производительности суперкомпьютеров семейства "СКИФ Аврора" на индустриальных задачах // Вестник ЮУрГУ. Серия "Математическое моделирование и программирование". 2010. N 35(211). С. 66–78.

26. *Соколинский Л.Б.* Параллельные системы баз данных. М.: Изд-во МГУ, 2013. 184 с.

27. *Nambiar R.O., Poess M., Masland A., et al.* TPC Benchmark Roadmap 2012 // Proceedings of the 4th TPC Technology Conference, TPCTC 2012, Istanbul, Turkey, August 27, 2012, Revised Selected Papers. Lecture Notes in Computer Science. Springer, 2013. P. 1–20.

28. *Костенецкий П.С., Лепихов А.В., Соколинский Л.Б.* Технологии параллельных систем баз данных для иерархических многопроцессорных сред // Автоматика и телемеханика. 2007. N 5. С. 112–125.

29. *Губин М.В., Соколинский Л.Б.* Об оценке коммуникационных затрат при обработке фрагментированного отношения для равномерного распределения // Вестник ЮУрГУ. Серия "Вычислительная математика и информатика", 2013. Т. 2. N 1. С. 33–43.

30. *Соколинский Л.Б.* Эффективный алгоритм замещения страниц для буферизации обменов с дисками в параллельной системе баз данных без совместного использования ресурсов // Вычислительные методы и программирование. 2002. Т. 3. N 1. С. 113–130.

31. *Костенецкий П.С., Соколинский Л.Б.* Моделирование иерархических многопроцессорных систем баз данных // Программирование. 2013. Т. 39. N 1. С. 2–33.