Федеральное государственное автономное образовательное учреждение
высшего образования
«ЮЖНО-УРАЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
(национальный исследовательский университет)»

На правах рукописи

ЦЫМБЛЕР Михаил Леонидович

ИНТЕЛЛЕКТУАЛЬНЫЙ АНАЛИЗ ДАННЫХ В СУБД

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

Диссертация на соискание ученой степени
доктора физико-математических наук

Научный консультант:

СОКОЛИНСКИЙ Леонид Борисович,
доктор физ.-мат. наук, профессор

Челябинск — 2019

Оглавление

Введение **5**

1. Интеллектуальный анализ данных **14**

1.1. Задачи интеллектуального анализа данных 14

1.1.1. Задача кластеризации 15

1.1.2. Поиск шаблонов 20

1.1.3. Анализ временных рядов 23

1.2. Применение СУБД для интеллектуального анализа данных 31

1.2.1. Интеграция анализа данных в СУБД 32

1.2.2. СУБД на основе фрагментного параллелизма 36

1.3. Обзор работ по теме диссертации 41

1.3.1. Системы анализа данных в СУБД 41

1.3.2. Алгоритмы кластеризации 44

1.3.3. Алгоритмы поиска шаблонов 49

1.3.4. Алгоритмы анализа временных рядов 52

1.4. Выводы по главе 1 56

2. Кластеризация и поиск шаблонов **59**

2.1. Алгоритм *dbParGraph* кластеризации графа в параллельной СУБД 59

2.1.1. Проектирование алгоритма 60

2.1.2. Реализация алгоритма 65

2.1.3. Вычислительные эксперименты 73

2.2. Алгоритм *pgFCM* нечеткой кластеризации данных в параллельной СУБД 76

2.2.1. Проектирование алгоритма 76

2.2.2. Реализация алгоритма 82

2.2.3. Вычислительные эксперименты 86

2.3. Параллельный алгоритм *PDIC* поиска частых наборов 87

2.3.1. Проектирование алгоритма 88
2.3.2. Реализация алгоритма 89
2.3.3. Вычислительные эксперименты 93
2.4. Параллельный алгоритм *DDCapriori* поиска частых наборов . . . 97
2.4.1. Проектирование алгоритма 98
2.4.2. Реализация алгоритма 101
2.4.3. Вычислительные эксперименты 103
2.5. Выводы по главе 2 105

3. Анализ временных рядов 110
3.1. Параллельный алгоритм поиска похожих подпоследовательностей *PBM* 110
3.1.1. Проектирование алгоритма 110
3.1.2. Реализация алгоритма 115
3.1.3. Вычислительные эксперименты 122
3.2. Параллельный алгоритм поиска диссонансов *MDD* 131
3.2.1. Проектирование алгоритма 131
3.2.2. Реализация алгоритма 134
3.2.3. Вычислительные эксперименты 139
3.3. Выводы по главе 3 143

4. Интеграция в СУБД параллельных алгоритмов анализа данных 145
4.1. Базовые идеи и мотивационный пример 145
4.2. Системная архитектура 148
4.2.1. Внешний и внутренний интерфейсы 149
4.2.2. Управление буферным пулом 151
4.2.3. Библиотека параллельных алгоритмов 152
4.3. Методы реализации 153
4.3.1. Организация хранения системных данных 153
4.3.2. Подсистема *Frontend* 155
4.3.3. Подсистема *Backend* 156

4.4. Библиотека параллельных алгоритмов 157
4.4.1. Алгоритм *PBlockwise* вычисления матрицы евклидовых расстояний 158
4.4.2. Алгоритм *PPAM* кластеризации данных на основе техники медоидов 161
4.5. Вычислительные эксперименты 166
4.6. Выводы по главе 4 174

5. Интеграция в СУБД фрагментного параллелизма 177
5.1. Архитектура параллельной СУБД на базе PostgreSQL 177
5.1.1. Взаимодействие процессов СУБД 177
5.1.2. Обработка запроса 180
5.1.3. Модульная структура 181
5.1.4. Развертывание компонентов 183
5.2. Методы интеграции параллелизма в реляционную СУБД на примере PostgreSQL 184
5.2.1. Подсистема тиражирования 184
5.2.2. Оператор обмена (*exchange*) 186
5.2.3. Параллелизатор плана запроса 190
5.2.4. Обработка запросов на изменение данных 193
5.2.5. Хранение метаданных о фрагментации 196
5.2.6. Прозрачное портирование приложений 197
5.2.7. Мягкая модификация исходных текстов 198
5.3. Вычислительные эксперименты 199
5.4. Выводы по главе 5 203

Заключение 205

Литература 220

Введение

Актуальность темы

В настоящее время одним из феноменов, оказывающих существенное влияние на область методов обработки данных, являются *Большие данные* [30, 58]. В условиях современного информационного общества имеется широкий спектр приложений (социальные сети [169], электронные библиотеки [257], геоинформационные системы [156] и др.), в каждом из которых производятся неструктурированные данные, имеющие сверхбольшие объемы и высокую скорость прироста (от 1 Терабайта в день). Исследования аналитической компании IDC показывают, что мировой объем данных удваивается каждые два года и к 2020 г. достигнет 44 Зеттабайт (44 триллиона Гигабайт) [235][1].

В современном информационном обществе, однако, критичными являются не столько объемы и скорость прироста Больших данных, а наличие эффективных методов и алгоритмов *интеллектуального анализа данных*, которые позволяют извлекать из этих данных доступные для понимания знания, необходимые для принятия важных решений в различных сферах человеческой деятельности [30]. В 2016 г. в Бекманском отчете [28] ведущие мировые специалисты в области технологий обработки данных констатировали, что переход к умному обществу, управляемому данными, требует интегрированного и сквозного процесса от получения данных до извлечения из них полезных знаний.

Системы управления базами данных (СУБД) на основе *реляционной модели данных*, предложенной Э. Коддом (Edgar Codd) [64] в 70-х гг. XX века, остаются на сегодняшний день основным и наиболее популярным инструментом для управления данными. Феномен Больших данных порождает процессы очистки и структурирования данных, в результате которых

[1] Данный объем сопоставим с пятью миллиардами видеофильмов в разрешении высокой четкости, для непрерывного просмотра которых одному человеку потребовалось бы более одного миллиарда лет.

неструктурированные данные преобразуются в сверхбольшие базы и хранилища реляционных данных. Один из наиболее авторитетных ученых в области баз данных М. Стоунбрейкер (Michael Stonebraker) указывает [226], что для решения проблемы обработки сверхбольших данных необходимо использовать технологии СУБД. В отличие от файловой системы СУБД обеспечивают широкий спектр сервисов, необходимых для эффективного управления данными: отказоустойчивость, целостность и безопасность данных, исполнение запросов к данным на основе индексирования данных и управления буферным пулом и др.

Однако использование внешней по отношению к СУБД (stand-alone) программной системы для интеллектуального анализа данных в хранилище влечет за собой значительные накладные расходы, связанные с предварительным экспортом анализируемых данных из хранилища и импортом результатов анализа обратно в хранилище [173]. Указанных накладных расходов можно избежать, выполняя интеллектуальный анализ данных непосредственно в СУБД [174]. Кроме того, оставаясь в рамках СУБД, прикладной программист и конечный пользователь алгоритмов интеллектуального анализа данных получают без дополнительных накладных расходов ряд вышеупомянутых преимуществ, заложенных в архитектуре СУБД.

Эффективная обработка и анализ сверхбольших хранилищ данных требуют использования параллельных СУБД на платформе высокопроизводительных вычислительных систем [71, 102]. Параллельная СУБД строится на основе концепции *фрагментного параллелизма*, предполагающей разбиение таблиц базы данных на горизонтальные фрагменты, которые могут обрабатываться независимо на разных узлах многопроцессорной системы. В настоящее время в рейтинге TOP500 [227] самых мощных суперкомпьютеров мира доминируют кластерные вычислительные системы, занимая 88% позиций списка (ноябрь 2018 г.).

Однако существующие сегодня коммерческие СУБД на основе фрагментного параллелизма (Teradata [185], Greenplum [237], IBM DB2 Parallel Edition [39] и др.) имеют высокую стоимость и ориентированы на специ-

фические аппаратно-программные платформы. В то же время *свободные СУБД* (PostgreSQL [225], MySQL [244] и др.) являются надежной альтернативой проприетарным решениям [84, 194]. Свободные СУБД предоставляют открытый исходный код, который может быть модернизирован любым разработчиком, что делает возможным построение параллельной СУБД на основе свободной СУБД путем внедрения в код последней фрагментного параллелизма. При этом модернизация исходного кода подразумевает отсутствие масштабных изменений в коде существующих подсистем, что было бы равнозначно разработке параллельной СУБД «с нуля». Кроме того, в настоящее время разработка отечественных СУБД имеет государственную поддержку[2].

Одной из современных тенденций развития процессоров является увеличение количества вычислительных ядер вместо тактовой частоты [86]. В рамках данной тенденции производителями процессоров разработаны аппаратные архитектуры IBM Cell Broadband Engine (BE) [112], NVIDIA GPU (Graphic Processing Units) [184], Intel Many Integrated Core (MIC) [78]. Соответствующие *ускорители* (сопроцессоры и самостоятельные процессорные системы) обеспечивают от десятков до сотен процессорных ядер, поддерживающих векторную обработку данных, и значительно опережают традиционные процессоры по производительности, позволяя исполнять от сотен до тысяч параллельных нитей. В упомянутом выше Бекманском отчете [28] указывается, что для эффективной обработки и анализа данных соответствующие решения должны обеспечить полноценное и масштабируемое использование возможностей как многоядерных ускорителей, так и кластеров с узлами на базе таких вычислительных систем.

В соответствии с вышесказанным является *актуальной* проблема разработки новых подходов и методов интеграции интеллектуального анализа в реляционные системы баз данных, а также разработка и реализация в рамках предлагаемых подходов новых параллельных алгоритмов интел-

[2]Постановление Правительства РФ № 1236 от 16 ноября 2015 г. «Об установлении запрета на допуск программного обеспечения, происходящего из иностранных государств, для целей осуществления закупок для обеспечения государственных и муниципальных нужд».

лектуального анализа данных для кластерных вычислительных систем с узлами на базе современных многоядерных ускорителей.

Степень разработанности темы

Феномен больших данных, характерный для современного информационного общества, обуславливает неослабевающий интерес научного сообщества к тематике хранения и обработки больших массивов данных. Важным аспектом данной области исследований являются технологии интеллектуального анализа данных с применением современных многоядерных многопроцессорных вычислительных систем.

Среди российских исследователей наибольший вклад в развитие технологий баз данных внесли научные группы под руководством С.Д. Кузнецова, Б.А. Новикова, С.В. Зыкина, В.Э. Вольфенгагена. В областях высокопроизводительных вычислительных технологий и параллельных систем баз данных значимые результаты принадлежат российским научным группам, возглавляемым Вл.В. Воеводиным и Л.Б. Соколинским соответственно. Зарубежными учеными-классиками, работающими в области систем баз данных, являются П. Валдуриц (Patrick Valduriez), Д. ДеВитт (David DeWitt), М. Стоунбрейкер (Michael Stonebraker), С. Мэдден (Samuel Madden), Д. Абади (Daniel Abadi). Проблематика эффективных методов интеллектуального анализа временных рядов исследована в работах следующих ученых: И. Кеог (Eamon Keogh), К. Фалутсос (Christos Faloutsos), А. Муин (Abdulla Mueen), С. Лим (Seung-Hwan Lim), С. Ким (Sang-Wook Kim), Я. Мун (Yang-Sae Moon). Весомый вклад в решение проблемы интеграции интеллектуального анализа данных в СУБД, а также в разработку алгоритмов поиска шаблонов внесли Дж. Хан (Jiawei Han), Р. Агравал (Rakesh Agrawal), С. Сараваджи (Sunita Sarawagi), К. Ордонез (Carlos Ordonez), М. Заки (Mohammed Zaki). Вклад в разработку алгоритмов кластеризации данных внесли Л. Кауфман (Leonard Kaufman), Дж. Бездек

(James Bezdek), Дж. Карипис (George Karypis), В. Кумар (Vipin Kumar), С. Гуха (Sudipto Guha), Ж. Хуанг (Zhexue Huang) и др.

На сегодняшний день технологии баз данных и интеллектуального анализа данных остаются в фокусе интенсивных научных исследований и практических разработок. Одной из важных нерешенных проблем остается задача разработки методов интеграции интеллектуального анализа данных в реляционные СУБД, адаптированных для современных многопроцессорных и многоядерных аппаратных платформ.

Цель и задачи исследования

Цель данной работы состояла в разработке комплекса масштабируемых методов и параллельных алгоритмов для создания программной платформы интеллектуального анализа данных средствами СУБД с открытым кодом. Данная цель предполагает решение следующих *задач*.

1. Разработать методы и алгоритмы для внедрения фрагментного параллелизма в свободную последовательную реляционную СУБД. Проверить эффективность предложенных решений на СУБД PostgreSQL.

2. Разработать методы и алгоритмы для внедрения интеллектуального анализа данных в параллельную СУБД для современных многопроцессорных платформ с многоядерными ускорителями.

3. Разработать параллельные алгоритмы решения задач кластеризации, поиска шаблонов и анализа временных рядов средствами параллельной реляционной СУБД.

4. Провести вычислительные эксперименты с синтетическими и реальными данными, подтверждающие эффективность предложенных методов и алгоритмов.

Научная новизна

Научная новизна работы заключается в следующем.

1. Разработан оригинальный метод интеграции интеллектуального анализа данных в реляционную СУБД на основе пользовательских функций, инкапсулирующих параллельные аналитические алгоритмы для современных многоядерных процессоров.

2. Разработан оригинальный метод интеграции фрагментного параллелизма в последовательную свободную СУБД, не требующий масштабных изменений в исходном коде.

3. Впервые разработаны параллельные алгоритмы анализа временных рядов для вычислительных кластеров с многоядерными ускорителями.

4. Разработаны новые параллельные алгоритмы кластеризации данных сверхбольших объемов для параллельной реляционной СУБД.

5. Разработаны новые параллельные алгоритмы поиска частых наборов и кластеризации данных для многоядерных ускорителей.

Теоретическая и практическая значимость работы

Теоретическая ценность диссертационной работы состоит в следующем. В работе предложены методы, архитектурные решения и алгоритмы, позволяющие интегрировать параллельную обработку и анализ данных в последовательные реляционные СУБД: предложен подход к интеграции интеллектуального анализа данных в СУБД, предполагающий встраивание в СУБД аналитических алгоритмов, которые инкапсулируют параллельное исполнение на современных многоядерных ускорителях; предложен подход к разработке параллельной СУБД, предполагающий интегра-

цию фрагментного параллелизма в СУБД с открытым исходным кодом. В работе предложены параллельные алгоритмы решения различных задач интеллектуального анализа данных (кластеризация, поиск частых наборов, поиск похожих подпоследовательностей и диссонансов во временных рядах) для современных многоядерных ускорителей, обеспечивающих ускорение, близкое к линейному.

Практическая ценность диссертационной работы заключается в том, что предложенные методы интеграции параллелизма применены к свободной СУБД PostgreSQL и разработаны прототипы библиотеки интеллектуального анализа данных и прототип параллельной СУБД PargreSQL. Результаты, полученные в работе, могут быть использованы в создании коммерческих и свободно распространяемых программных продуктов, ориентированных на параллельную обработку и анализ данных с использованием свободной реляционной СУБД.

Методология и методы исследования

Проведенные в работе исследования базируются на реляционной модели данных, методах интеллектуального анализа данных и теории временных рядов. При разработке программных комплексов применялись методы объектно-ориентированного проектирования и язык UML, а также методы системного, модульного и объектно-ориентированного программирования. В реализации параллельных алгоритмов использованы методы параллельного программирования для общей и распределенной памяти на основе стандартов MPI и OpenMP, а также методы параллельных систем баз данных.

Структура и объем работы

Диссертация состоит из введения, пяти глав, заключения и библиографии. Объем диссертации составляет 260 страниц, объем библиографии —

274 наименования.

Содержание работы

Первая глава, «Интеллектуальный анализ данных», посвящена общим вопросам использования методов баз данных для интеллектуального анализа данных. В главе приводится обзор типовых задач анализа данных. Рассмотрены современные методы и подходы к интеграции интеллектуального анализа данных в реляционные СУБД. Дается обзор публикаций, наиболее близко относящихся к теме диссертации.

Во второй главе, «Кластеризация и поиск шаблонов», рассмотрены две задачи интеллектуального анализа данных: кластеризация и поиск шаблонов. В рамках первой задачи исследована проблематика использования реляционных СУБД для кластеризации больших объемов данных. Предложены алгоритмы *dbParGraph* для кластеризации графа и *pgFCM* для нечеткой кластеризации данных для параллельной СУБД на основе фрагментного параллелизма. В рамках второй задачи исследованы параллельные методы поиска шаблонов на многоядерных процессорах. Предложены следующие параллельные алгоритмы поиска частых наборов для многоядерных вычислителей: алгоритм *PDIC* для ускорителя Intel Xeon Phi и алгоритм *DDCapriori* для процессора IBM Cell BE. Представлены результаты вычислительных экспериментов, исследующих эффективность разработанных алгоритмов.

Третья глава, «Анализ временных рядов», посвящена методам интеллектуального анализа временных рядов на платформе современных многопроцессорных многоядерных вычислительных систем. Рассмотрены следующие задачи анализа временных рядов: поиск похожих подпоследовательностей во временном ряде и поиск диссонансов во временном ряде. Предложены алгоритмы решения указанных задач: алгоритм поиска похожих подпоследовательностей *PBM* для кластерных систем с узлами на базе многоядерных ускорителей и алгоритм поиска диссонансов *MDD* для мно-

гоядерного ускорителя. Представлены результаты вычислительных экспериментов, исследующих эффективность разработанных алгоритмов.

Четвертая глава, «Интеграция в СУБД параллельных алгоритмов анализа данных», представляет подход к интеграции интеллектуального анализа данных и реляционных СУБД. Указанный подход предполагает встраивание в СУБД аналитических алгоритмов, которые инкапсулируют параллельное исполнение на современных многоядерных ускорителях. Описана системная архитектура и методы реализации подхода для свободной СУБД PostgreSQL и многоядерных ускорителей. Приведены результаты вычислительных экспериментов, исследующих эффективность предложенного подхода.

В пятой главе, «Интеграция в СУБД фрагментного параллелизма», представлены методы, позволяющие внедрить фрагментный параллелизм в свободную реляционную СУБД посредством модификации ее открытых исходных кодов. Описаны архитектура и методы реализации параллельной СУБД *PargreSQL*, полученной путем распараллеливания свободной СУБД PostgreSQL. Представлены результаты вычислительных экспериментов, исследующих эффективность предложенного решения.

Заключение подводит итоги диссертационной работы и содержит описание ключевых отличий данного исследования от ранее выполненных родственных работ других авторов, а также рекомендации по использованию полученных результатов и направления дальнейших исследований в данной области.

Глава 1. Интеллектуальный анализ данных

В первой главе формулируются типовые задачи интеллектуального анализа данных: кластеризация, поиск шаблонов (ассоциативных правил) и анализ временных рядов. Рассматриваются известные подходы к использованию сервисов СУБД для интеллектуального анализа данных. Дается обзор публикаций, наиболее близко относящихся к теме диссертации.

1.1. Задачи интеллектуального анализа данных

Под *интеллектуальным анализом данных (Data Mining)* понимают совокупность алгоритмов, методов и программного обеспечения для обнаружения в данных ранее неизвестных, нетривиальных, практически полезных и доступных интерпретации знаний, необходимых для принятия стратегически важных решений в различных сферах человеческой деятельности [99]. В качестве синонима также используется термин *обнаружение знаний в базах данных (Knowledge Discovery in Databases)* [89]. Терминология введена Пятецким-Шапиро (Piatetsky-Shapiro) и Файядом (Fayyad) в 1991–1996 гг. в работах [89, 268].

Точкой отсчета интеллектуального анализа данных как самостоятельной научной области принято считать семинар Knowledge Discovery in Real Databases [197], проведенный Пятецким-Шапиро в рамках международной научной конференции по искусственному интеллекту IJCAI'89 (The 11th International Joint Conference on Artificial Intelligence) в 1989 г. [273].

В качестве основных причин, обусловивших возникновение области интеллектуального анализа данных, можно указать следующие вызовы того времени, отмеченные в отчетах по результатам встреч ведущих предста-

вителей исследовательского сообщества о состояниях и перспективах технологий обработки данных в Лагуна-Бич в 1989 г. [41] и Пало-Альто в 1990 г. [220]:

— появление новых типов информационных объектов и необходимость внедрения в СУБД соответствующих средств их аналитической обработки: изображение, документ, географическая карта и др.;

— накопление больших объемов ретроспективных данных вследствие удешевления систем хранения;

— необходимость внедрения в СУБД методов компьютерной поддержки полного производственного цикла (Computer Integrated Manufacturing) и информационного поиска (Information Retrieval).

Спустя 10 лет в Клермонтском отчете [31] указывалось, что методы и технологии интеллектуального анализа данных становятся центром прибыли (например, в 2007 г. общий объем сделок по поглощению компаний, поставляющих программное обеспечение для аналитической обработки данных, превысил 10 миллиардов долларов), что требует совершенствования соответствующих методов и алгоритмов.

В интеллектуальном анализе данных выделяют как типовые следующие задачи: кластеризация, поиск шаблонов и анализ временных рядов [89, 247], — которые рассматриваются в следующих разделах.

1.1.1. Задача кластеризации

Задача *кластеризации (clustering)* заключается в разбиении множества объектов сходной структуры на заранее неизвестные группы (кластеры) в зависимости от похожести свойств объектов. Формальное определение задачи кластеризации выглядит следующим образом.

Пусть заданы конечные множества: $X = \{x_1, x_2, \ldots, x_n\}$ — множество объектов d-мерного метрического пространства, для которых задана функ-

ция расстояния $\rho(x_i, x_j)$, и $C = \{c_1, c_2, \ldots, c_k\}$, где $k \ll n$ — набор уникальных идентификаторов (номеров, имен, меток) кластеров.

Алгоритм (четкой) кластеризации определяется как функция $\alpha : X \to C$, которая каждому объекту назначает уникальный идентификатор кластера. Алгоритм кластеризации выполняет разбиение множества X на непересекающиеся непустые подмножества (*кластеры*) таким образом, чтобы каждый кластер состоял из объектов, близких по метрике ρ, а объекты разных кластеров существенно отличались. *Алгоритм нечеткой кластеризации* позволяет одному и тому же объекту принадлежать одновременно всем кластерам (с различной степенью принадлежности).

Основные алгоритмы кластеризации

Существующие алгоритмы кластеризации подразделяются на разделительные, иерархические, плотностные и решеточные [99].

Разделительная (partitioning) кластеризация предполагает начальное разбиение исходного множества объектов на кластеры (возможно, выполняемое случайным образом), при котором в каждом кластере имеется, по крайней мере, один объект и каждый объект принадлежит в точности одному кластеру. После выполнения начального разбиения разделительный алгоритм итеративно осуществляет перемещения объектов между кластерами с целью улучшить начальное разбиение (чтобы объекты из одного кластера были более «близкими», а из разных кластеров — более «далекими» друг другу). При этом, поскольку поиск всех возможных разбиений может привести к большим накладным расходам, вместо него используются эвристики.

В алгоритме *k-Means* [144] при улучшении разбиения каждый кластер представляется посредством среднего значения координат объектов в кластере. Для представления кластеров в разделительных алгоритмах могут использоваться также медиана или мода координат объектов (алгоритмы *k-Medians* [97] и *k-Modes* [56, 108] соответственно).

Алгоритмы *k-Medoids* [168] и *PAM (Partitioning Around Medoids)* [115] в качестве представления каждого кластера используют тот объект подвергаемого кластеризации множества, который находится ближе остальных к центру кластера. Техника медоидов направлена на повышение устойчивости алгоритма к выбросам и шумам в данных (робастности) и применяется в широком спектре приложений, связанных с кластеризацией: сегментирование медицинских и спутниковых изображений, анализ ДНК-микрочипов и текстов и др.

Иерархическая кластеризация заключается в последовательном разбиении исходного множества объектов по уровням иерархии. В зависимости от дисциплины, в соответствии с которой выполняется разбиение, иерархические алгоритмы кластеризации подразделяются на агломеративные и дивизимные.

Агломеративный алгоритм кластеризации начинает работу в предположении, что каждый исходный объект образует отдельный кластер, и затем выполняет слияние близких друг к другу объектов или кластеров до тех пор, пока не будет получен единственный кластер или не будет выполнено условие завершения слияния. Примером агломеративного подхода является алгоритм *AGNES* [115]. *Дивизимный алгоритм* кластеризации, напротив, стартует, предполагая, что все исходные объекты входят в один кластер, и затем итеративно выполняет его разбиение на менее мощные кластеры до тех пор, пока не будут получены кластеры-синглтоны или не будет выполнено условие завершения слияния. Дивизимный подход реализован в алгоритме *DIANA* [115].

Плотностная (density-based) кластеризация предполагает добавление объектов (называемых в контексте плотностных методов точками) в кластер до тех пор, пока плотность (количество) соседних точек не превысит некоторого наперед заданного значения порога концентрации. В соответствии с этим в окрестности каждой точки кластера должно находиться некоторое минимальное количество других точек. Плотностная кластеризация может использоваться для нахождения аномалий и класте-

ров произвольной формы (в отличие от разделительных алгоритмов, которые приспособлены для нахождения кластеров сферической формы). Типичным представителем плотностной кластеризации является алгоритм *DBSCAN* [83], осуществляющий построение кластера как множества связанных (density-connected) точек, которое имеет наибольшую мощность.

Определенным недостатком плотностных алгоритмов можно считать их чувствительность к входным параметрам (радиус окрестности и порог концентрации точек в окрестности), которые устанавливаются эмпирическим путем и трудно определяются для реальных данных, имеющих объекты с большим количеством атрибутов.

Решеточная (grid-based) кластеризация предполагает разбиение пространства исходных данных на конечное число ячеек, формирующих решеточную структуру, над которой выполняются операции, необходимые для кластеризации. Алгоритм *STING* [240] использует статистическую информацию, хранящуюся в прямоугольных ячейках решетки. Статистические данные о ячейках верхних уровней вычисляются на основе статистических данных о ячейках нижних уровней. Для кластеризации используются следующие статистические данные: количество точек в ячейке, минимальное, максимальное, среднее значение атрибутов и др.

Нечеткая (fuzzy) кластеризация предполагает, что каждого элемента кластеризуемого множества объектов вычисляется степень его принадлежности (responsibility) каждому из кластеров. Алгоритм нечеткой кластеризации *Fuzzy c-Means (FCM)*, обобщающий в этом смысле вышеупомянутый алгоритм разделительной кластеризации *k-Means* [146], предложен Данном (Dunn) [77] и впоследствии улучшен Бездеком (Bezdek) [43].

Кластеризация графов

Одной из областей приложения методов интеллектуального анализа данных являются задачи аналитической обработки *сверхбольших графов*, в которых количество вершин и ребер имеет порядок от миллионов, что не позволяет разместить такие графы целиком в оперативной памяти. По-

добные графы возникают при моделировании сложных систем различного рода с большим количеством связей между элементами: химические соединения [238], биологические и социальные сети [231], Web [74], версионирование сложного программного обеспечения [79] и др. В соответствии с этим исследователи обосабливают задачу кластеризации графовых данных [29].

Разбиение множества вершин графа на кластеры осуществляется на основе весов ребер этого графа, трактуемые как расстояния между соседними вершинами. В теории графов задача кластеризации известна как задача *разбиения графа (graph partitioning)* и формально определяется следующим образом [87].

Пусть имеется граф $G = (N, E)$, где N — множество взвешенных вершин, E — множество взвешенных ребер[1], и дано целое число $p > 0$. Тогда *p-разбиением* графа G называются такие p подмножеств вершин N_1, ..., N_p, $N_i \subseteq N$, для которых выполняются следующие условия:

1. $\bigcup\limits_{i=1}^{p} N_i = N$ и $N_i \cap N_j = \varnothing \ \forall i \neq j$;
2. $w(N_i) \approx \frac{w(N)}{p} \ \forall i = 1, 2, \ldots, p$ (где $w(N_i)$ и $w(N)$ обозначают суммы весов вершин из N_i и N соответственно);
3. величина $W_{cut} = \sum\limits_{(u,v) \in E_{cut}} w(u, v)$ минимальна,
   где *разрез* $E_{cut} = \{(u, v) \in E \mid u \in N_i, v \in N_j, 1 \leqslant i, j \leqslant p, i \neq j\}$.

Типичным является случай 2-разбиения (p=2), или *бисекции*, когда граф разбивается на две части. Для разбиения графа на большее количество частей бисекция рекурсивно применяется к найденным частям [87].

Одним из первых алгоритмов разбиения графа является *алгоритм Кернигана—Лина* [118]. Данный алгоритм использует технику поиска соседей для нахождения оптимального разбиения графа. На первом шаге алгоритма берется случайное разбиение графа. Далее, на каждой итерации осуществляется попытка обмена пары вершин в двух кластерах и определение

[1] Для невзвешенных графов веса полагают равными единице [87].

общего веса разреза. Если уменьшение общего веса разреза имело место, то обмен вершин выполняется. В противном случае берется следующая пара вершин. Этот процесс повторяется до нахождения оптимального решения, которое может не представлять собой глобальный оптимум, но являться лишь локальным оптимумом исходных данных.

1.1.2. Поиск шаблонов

Задача *поиска шаблонов (pattern mining)* или *ассоциативных правил (association rule mining)* заключается в нахождении часто повторяющихся зависимостей в заданном конечном наборе объектов. Данная задача может быть разбита на две последовательно выполняемые подзадачи: поиск всех частых наборов и генерация устойчивых ассоциативных правил на основе найденных частых наборов [34]. Формальное определение задачи поиска шаблонов выглядит следующим образом.

Пусть дано конечное множество *объектов* $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$, любое непустое его подмножество называют *набором*. Набор из k объектов ($1 \leqslant k \leqslant m$) называют *$k$-набором*.

Пусть имеется множество *транзакций* $\mathcal{D}$, в котором каждая транзакция представляет собой пару $(tid; I)$, где tid — уникальный идентификатор транзакции, $I \subseteq \mathcal{I}$ — набор.

Поддержкой набора $I \subseteq \mathcal{I}$ является доля транзакций $\mathcal{D}$, содержащих данный набор:

$$support(I) = \frac{|\{T \in \mathcal{D} \mid I \subseteq T.I\}|}{|\mathcal{D}|}. \tag{1.1}$$

Наперед задаваемое *пороговое значение поддержки* $minsup$ является параметром задачи. Набор, поддержка которого не ниже $minsup$, называют *частым*, иначе набор называют *редким*.

Обозначим множество всех частых k-наборов как $\mathcal{L}_k$, тогда решением подзадачи *поиска всех частых наборов* будет множество $\mathcal{L} = \cup_{k=1}^{k_{max}} \mathcal{L}_k$, где k_{max} — максимальное количество объектов в частом наборе.

Шаблоном называют запись вида $A \to B$ (где A называют *антецедентом*, а B — *консеквентом* шаблона), которая означает соблюдение следующих условий:

$$\begin{gathered} A \subseteq \mathcal{I}, B \subseteq \mathcal{I}, A \cap B = \varnothing \\ \forall T \in \mathcal{D}\ (A \subseteq T.I \Rightarrow B \subseteq T.I). \end{gathered} \tag{1.2}$$

Иными словами, шаблон показывает, что из наличия в некоторой транзакции набора-антецедента следует наличие в данной транзакции также набора-консеквента.

Поддержка шаблона $A \to B$ вычисляется как доля транзакций в $\mathcal{D}$, в которых содержатся оба набора A и B (обозначается как $A \cup B$):

$$support(A \to B) = \frac{|\{T \in \mathcal{D} \mid A \cup B \subseteq T.I\}|}{|\mathcal{D}|}. \tag{1.3}$$

Достоверность шаблона $A \to B$ вычисляется как отношение количества транзакций в $\mathcal{D}$, содержащих оба набора A и B, к количеству транзакций, которые содержат набор A:

$$confidence(A \to B) = \frac{|\{T \in \mathcal{D} \mid A \cup B \subseteq T.I\}|}{|\{T \in \mathcal{D} \mid A \subseteq T.I\}|}. \tag{1.4}$$

Наперед задаваемое *пороговое значение достоверности* $minconf$ является параметром задачи. Шаблон будет *устойчивым (strong rule)*, если его поддержка и достоверность не ниже пороговых значений $minsup$ и $minconf$ соответственно.

Устойчивые шаблоны генерируются на основе множества всех частых наборов $\mathcal{L}$ в соответствии со следующим алгоритмом [99] (см. алг. 1.1).

Вначале множество всех устойчивых шаблонов $\mathcal{S}$ полагается пустым. Далее для каждого набора ℓ из множества всех частых наборов $\mathcal{L}$ генерируется его *булеан* $\mathcal{P}(\ell)$ — множество всех подмножеств набора ℓ. Затем для каждого непустого элемента s из $\mathcal{P}(\ell)$ в множество $\mathcal{S}$ будет добавлен шаблон с антецедентом s и консеквентом $\ell \setminus s$ в том случае, если соотношение значений поддержек наборов ℓ и s не ниже порогового значения $minconf$.

Алг. 1.1. StrongPatterns(in $\mathcal{L}$, out $\mathcal{S}$)

```
1: S ← ∅
2: for all ℓ ∈ L do
3:     for all s ∈ P(ℓ) ∖ ∅ do
4:         if support(ℓ)/support(s) ⩾ minconf then
5:             rule_new ← "s → ℓ ∖ s"
6:             S ← S ∪ rule_new
7:         end if
8:     end for
9: end for
```

Классическим алгоритмом решения задачи поиска частых наборов является алгоритм *Apriori* [34]. Идея алгоритма заключается в итеративной генерации множества *кандидатов* в частые наборы и последующем отборе кандидатов с подходящим значением поддержки. Итерация осуществляется по k, количеству объектов в наборах-кандидатах, начиная с $k = 1$. В алгоритме используется следующее свойство *антимонотонности поддержки* (принцип *a priori*), которое позволяет исключать из рассмотрения заведомо редкие наборы: если k-набор является редким, то содержащий его $(k + 1)$-набор также является редким.

Узким местом алгоритма *Apriori* является операция генерации и проверки наборов-кандидатов, поскольку при достаточно больших значениях k и малых значениях $minsup$ имеют место значительные накладные расходы на поддержку наборов-кандидатов и повторяющиеся операции сканирования множества транзакций и подсчета поддержки. Впоследствии был разработан ряд улучшений алгоритма Apriori, связанных с сокращением количества наборов-кандидатов, количества просматриваемых транзакций и количества операций сканирования: алгоритмы *AprioriTid* [34], *DHP* [190], *Partition* [214], *DIC* [50], *Eclat* [255] и др.

Одним из наиболее эффективных алгоритмов решения задачи поиска частых наборов является алгоритм *FP-Growth* [101]. Данный алгоритм использует стратегию «разделяй и властвуй», избегая итеративной генерации наборов-кандидатов. На первой фазе алгоритма за две операции полного

сканирования множества транзакций выполняется построение специальной структуры данных, *FP-дерева (FP tree, frequent pattern tree)*, которое в компактном виде хранит наборы и их поддержку. На второй фазе с помощью рекурсивного обхода построенного дерева осуществляется генерация устойчивых шаблонов.

Алгоритм FP-Growth является существенно более эффективным, чем алгоритмы на основе принципа Apriori. Однако он предъявляет большие требования к объему необходимой оперативной памяти и не является масштабируемым для случая сверхбольших объемов исходных данных [259]. Продолжением исследований по улучшению подхода FP-Growth являются алгоритмы *AFOPT* [142], *OpportuneProject* [143] и др.

1.1.3. Анализ временных рядов

Временны́е ряды представляют собой важный класс темпоральных данных, получаемых в широком спектре предметных областей: мониторинг показателей функциональной диагностики организма человека [4], моделирование климата [1], финансовое прогнозирование [3] и др.

Под *временным рядом (time series)* понимают последовательность хронологически упорядоченных вещественных значений. Наличие у каждого элемента ряда временно́й метки, неотделимой от значения, существенным образом влияет на постановку и решение задач интеллектуального анализа временных рядов [81, 92].

Поиск похожих подпоследовательностей (subsequence matching) во временном ряде [92] является одной из важных задач интеллектуального анализа временных рядов. Поиск похожих подпоследовательностей предполагает нахождение участков данного ряда, которые являются похожими на заданный ряд существенно меньшей длины в смысле заданной меры схожести. Данная проблема была поставлена впервые в 1993 г. в работе [32] Агравалом (Agrawal) и др. и далее на протяжении более чем десятилетия

рассматривалась в качестве основной проблемы интеллектуального анализа временных рядов [81].

Другой важной задачей интеллектуального анализа временных рядов является *поиск диссонансов (discord)* временного ряда. Диссонанс является уточнением понятия аномальной подпоследовательности временного ряда. Задача поиска диссонансов встречается в широком спектре предметных областей, связанных с временными рядами: медицина [116], экономика [3], моделирование климата [1] и др. Понятие диссонанса предложено Кеогом (Keogh) в 2005 г. в работе [116] и определяется как подпоследовательность ряда, которая имеет максимальное расстояние до ближайшего соседа. Ближайшим соседом подпоследовательности является та подпоследовательность ряда, которая не пересекается с данной и имеет минимальное расстояние до нее.

Дадим ниже определения, необходимые для формальной постановки указанных задач, в соответствии с работой [81].

Временной ряд T представляет собой последовательность числовых значений, каждое из которых ассоциировано с отметкой времени, взятых в хронологическом порядке: $T = (t_1, t_2, \ldots, t_m)$, $t_i \in \mathbb{R}$. Число m обозначается $|T|$ и называется длиной ряда.

Подпоследовательность $T_{i,n}$ временного ряда T представляет собой непрерывное подмножество T из n элементов, начиная с позиции i: $T_{i,n} = (t_i, t_{i+1}, \ldots, t_{i+n-1}), 1 \leqslant n \ll m, 1 \leqslant i \leqslant m - n + 1$. Множество всех подпоследовательностей ряда T, имеющих длину n, обозначается как S_T^n.

Мерой схожести $\mathcal{D}$ между двумя временными рядами[2] X и Y называется вещественная неотрицательная функция, вычисляющая расстояние между данными рядами: $\mathcal{D}(X, Y) \geqslant 0$.

[2]Данное определение естественным образом продолжается на случай, когда схожесть определяется между подпоследовательностями, принадлежащими одному временному ряду или разным временным рядам.

Задача поиска похожих подпоследовательностей

Пусть фиксирована мера схожести $\mathcal{D}$, имеется временной ряд T и заданный пользователем существенно более короткий временной ряд Q, называемый *поисковым запросом (query)* ($m = |T| \gg |Q| = n$). Тогда *наиболее похожей подпоследовательностью (best match)* называют такую подпоследовательность $T_{i,n}$, форма которой максимально похожа на запрос Q в смысле меры $\mathcal{D}$:

$$\exists i \; \forall k \; \mathcal{D}(Q, T_{i,n}) \leqslant \mathcal{D}(Q, T_{k,n}), \; 1 \leqslant i, k \leqslant m - n + 1. \tag{1.5}$$

В качестве одного из первых подходов к решению проблемы поиска похожих подпоследовательностей было предложенное Фалутсосом (Faloutsos) и др. в 1994 г. создание индекса подпоследовательностей исходного временного ряда [85]. Временной ряд преобразуется в набор многомерных прямоугольников, по которым строится пространственный индекс в многомерном пространстве, а поиск подпоследовательностей заключается в выполнении пространственных запросов к этому индексу.

На основе данного подхода впоследствии были разработаны улучшенные алгоритмы поиска похожих подпоследовательностей. Мун (Moon) и др. в 2001 г. предложили алгоритм *DualMatch* [161] и его улучшенную версию *GeneralMatch* [160], которые основаны на разделении временного ряда на непересекающиеся окна и разделении запроса на скользящие окна для создания пространственного индекса. Далее Лим (Lim) и др. в 2007 г. разработал метод поиска похожих подпоследовательностей, в котором для увеличения производительности алгоритма DualMatch используется интерполяция индекса [137, 138].

Вышеупомянутые алгоритмы используют при поиске евклидово расстояние в качестве меры схожести. Однако, в настоящее время *динамическая трансформация временно́й шкалы (Dynamic Time Warping, DTW)*, предложенная Берндтом (Berndt) и Клиффордом (Clifford) в 1994 г. в работе [40], рассматривается научным сообществом как наилучшая мера схожести для

подавляющего большинства приложений интеллектуального анализа временных рядов из различных предметных областей [72, 202, 250].

Формальное определение меры схожести DTW выглядит следующим образом [40]. Пусть имеются два временных ряда $X = (x_1, x_2, \ldots, x_m)$ и $Y = (y_1, y_2, \ldots, y_m)$. Тогда

$$DTW(X, Y) = d(m, m),$$
$$d(i, j) = (x_i - y_j)^2 + \min \begin{cases} d(i-1, j) \\ d(i, j-1) \\ d(i-1, j-1) \end{cases}, \tag{1.6}$$
$$d(0, 0) = 0,\ d(i, 0) = d(0, j) = \infty,\ 1 \leqslant i \leqslant m,\ 1 \leqslant j \leqslant m.$$

Мера[3] DTW позволяет сравнивать ряды, а также подпоследовательности рядов, которые смещены вдоль временной оси, сжаты, растянуты друг относительно друга или имеют разные длины. Далее рассматривается случай аргументов равной длины, поскольку это упрощает изложение и не ограничивает общность [200].

В формуле (1.6) матрица $(d_{ij}) \in \mathbb{R}^{m \times m}$ выражает соответствие между точками сравниваемых временных рядов и называется *матрицей трансформации. Путь трансформации (warping path)* P представляет собой последовательность элементов матрицы трансформации, которая определяет соответствие между временными рядами Q и C. Пусть элемент p_t пути трансформации P определяется как $p_t = (i, j)_t$, тогда

$$P = (p_1, p_2, \ldots, p_t, \ldots, p_T),\ m \leqslant T \leqslant 2n - 1. \tag{1.7}$$

На путь трансформации накладывается ряд ограничений: путь должен начинаться и заканчиваться в диагонально противоположных элементах матрицы трансформации, шаги пути ограничены соседними ячейками, а точки пути должны быть монотонно разнесены во времени.

[3] DTW *не* определяет метрику, поскольку для нее выполняются аксиомы тождества и симметрии, но не выполняется аксиома треугольника [40, 200].

Мера DTW имеет квадратичную вычислительную сложность и ее вычисление может доходить до 80% от общего времени выполнения поиска похожих подпоследовательностей [258].

Объем вычислений при подсчете меры схожести DTW может быть уменьшен за счет огрубления схожести. Для этого на путь трансформации могут налагаться дополнительные ограничения. Одним из наиболее часто применяемых ограничений является т.н. *полоса Сако—Чиба* [210], не позволяющая пути трансформации отклоняться более чем на r элементов от диагонали матрицы трансформации, где наперед задаваемый параметр r $(1 \leqslant r \leqslant n)$ называется *шириной полосы Сако—Чиба*. Уменьшение ширины полосы Сако—Чиба позволяет уменьшить объем вычислений при подсчете меры схожести DTW ценой более грубого определения схожести подпоследовательностей и поискового запроса.

В работах Парка (Park) [191] и Кима (Kim) [121] 2000–2002 гг. исследуется применение техники индексирования в поиске похожих подпоследовательностей на основе меры DTW. Используется преобразование значений временного ряда из непрерывных в дискретные, по которым в дальнейшем строится индекс на основе суффиксного дерева [66]. Лим (Lim) и др. [137] в 2006 г. для ускорения вычислений предложили технику индексирования, которая требует заранее задавать длину поискового запроса, что не всегда приемлемо.

Сакураи (Sakurai) и др. в 2007 г. разработали алгоритм *SPRING* [211], который использует технику повторного использования вычислений. Однако, данная техника ограничивает приложение алгоритма, поскольку повторное использование данных предполагает использование последовательностей, не подвергаемых нормализации.

Алгоритм *UCR-DTW* поиска самой похожей подпоследовательности, предложенный Кеогом (Keogh) и др. в 2012 г. работе [200], является на сегодня, вероятно, самым быстрым последовательным алгоритмом поиска похожих подпоследовательностей [213]. Данный алгоритм интегрирует большое количество существующих техник ускорения вычисления меры

DTW.

Алгоритм *UCR-DTW* предполагает сравнение поискового запроса и подпоследовательностей, подвергнувшихся процедуре z-нормализации [200], которая позволяет сравнивать формы рядов, отличных по амплитуде. После нормализации среднее арифметическое временного ряда приблизительно равно 0, а среднеквадратичное отклонение близко к 1.

Z-нормализацией временного ряда T называется временной ряд $\hat{T} = (\hat{t}_1, \ldots, \hat{t}_m)$, элементы которого вычисляются следующим образом:

$$\hat{t}_i = \frac{t_i - \mu}{\sigma},\ \mu = \frac{1}{m}\sum_{i=1}^{m} t_i,\ \sigma = \sqrt{\frac{1}{m}\sum_{i=1}^{m} t_i^2 - \mu^2}. \tag{1.8}$$

Алгоритм *UCR-DTW* использует также *каскадное применение нижних границ схожести. Нижняя граница схожести (lower bound, LB)* представляет собой функцию $LB : S_T^n \times S_T^n \to \mathbb{R}$, вычислительная сложность которой меньше вычислительной сложности меры DTW ($O(n^2)$). Нижние границы используются для отбрасывания кандидатов, заведомо непохожих на запрос, без вычисления меры DTW и позволяют существенно сократить объем вычислений [72].

Техника использования нижних границ схожести выглядит следующим образом. Алгоритм *UCR-DTW* выполняет сканирование подпоследовательностей исходного временного ряда, с $T_{1,n}$ до $T_{m-n+1,n}$, с шагом 1. Значение схожести текущей подпоследовательности $T_{i,n}$ и поискового запроса Q обозначают как *bsf (best-so-far)*. Если нижняя граница для рассматриваемой подпоследовательности превышает порог bsf, то значение меры DTW для данной подпоследовательности также превысит bsf, и $T_{i,n}$ заведомо непохожа на запрос.

Алгоритм инициализирует bsf значением $+\infty$ и на i-м шаге поиска пытается улучшить (уменьшить) значение bsf, вычисляя его следующим

образом:

$$bsf_{(i)} = \min\Big(bsf_{(i-1)}, \begin{cases} +\infty & , LB(Q, T_{i,n}) > bsf_{(i-1)} \\ DTW(Q, T_{i,n}) & , otherwise \end{cases}\Big) \quad (1.9)$$

Для повышения эффективности отбрасывания заведомо непохожих подпоследовательностей алгоритм *UCR-DTW* применяет каскад следующих нижних границ [201]: $LB_{Kim}FL$ [120], $LB_{Keogh}EQ$ [90], $LB_{Keogh}EC$ [200].

Нижняя граница схожести $LB_{Kim}FL$ представляет собой Евклидово расстояние между первой и последней парами точек поискового запроса Q и подпоследовательности C:

$$LB_{Kim}FL(Q, C) = ED(q_1, c_1) + ED(q_n, c_n). \quad (1.10)$$

Евклидово расстояние определяется следующим образом:

$$ED(Q, C) = \sqrt{\sum_{i=1}^{n}(q_i - c_i)^2}. \quad (1.11)$$

Нижняя граница $LB_{Keogh}EC$ показывает схожесть между *оболочкой (envelope)* E поискового запроса Q и подпоследовательностью C и вычисляется следующим образом:

$$LB_{Keogh}EC(Q, C) = \sum_{i=1}^{n} \begin{cases} (c_i - u_i)^2, & if\ c_i > u_i \\ (c_i - \ell_i)^2, & if\ c_i < \ell_i \\ 0, & otherwise \end{cases}. \quad (1.12)$$

В формуле (1.12) последовательности $U = (u_1, .., u_n)$ и $L = (\ell_1, .., \ell_n)$ обозначают *верхнюю* и *нижнюю границы оболочки* запроса Q соответственно, которые вычисляются следующим образом:

$$\begin{aligned} u_i &= \max(q_{i-r}, \ldots, q_{i+r}), \\ \ell_i &= \min(q_{i-r}, \ldots, q_{i+r}). \end{aligned} \quad (1.13)$$

Нижняя граница $LB_{Keogh}EQ$ представляет собой Евклидово расстояние между запросом Q и оболочкой кандидата C, то есть по сравнению с $LB_{Keogh}EC$ роли запроса и подпоследовательности меняются местами:

$$LB_{Keogh}EQ(Q, C) := LB_{Keogh}EC(C, Q). \tag{1.14}$$

Задача поиска диссонансов

Дадим формальное определение задачи поиска диссонансов в соответствии с терминологией и обозначениями в работе [117].

Подпоследовательности $T_{i,n}$ и $T_{j,n}$ ряда T называются *непересекающимися (non-self match)*, если $|i - j| \geqslant n$. Подпоследовательность, которая является непересекающейся к данной подпоследовательности C, обозначается как M_C.

Пусть функция $Dist : S_T^n \times S_T^n \to \mathbb{R}$ удовлетворяет аксиомам тождества ($\forall X \in S_T^n \; Dist(X, X) = 0$) и симметрии ($\forall X, Y \in S_T^n \; Dist(X, Y) = Dist(Y, X)$). Тогда подпоследовательность D ряда T является *диссонансом (discord)*, если $\forall C, M_C \in T \; \min(Dist(D, M_D)) > \min(Dist(C, M_C))$. Другими словами, некая подпоследовательность ряда является диссонансом, если она имеет максимальное расстояние до ближайшей непересекающейся с ней подпоследовательностью.

Метод поиска диссонансов во временном ряде, предложенный в работах [116, 117], предполагает предварительную z-нормализацию исходного ряда по формулам (1.8) и использование евклидовой метрики (1.11) в качестве функции расстояния.

Подпоследовательности ряда подвергаются *кусочно-агрегатной аппроксимации (PAA, Piesewise Aggregate Approximation)* [140]. Для подпоследовательности $C = (c_1, c_2, \ldots, c_n)$ кусочно-агрегатным представлением (PAA-представлением) является вектор $\overline{C} = (\overline{c}_1, \ldots, \overline{c}_w)$, где *степень агрегации* $w \leqslant n$ — параметр, а координаты вектора вычисляются следующим обра-

зом:

$$\overline{c}_i = \frac{w}{n} \cdot \sum_{j=\frac{n}{w}\cdot(j-1)+1}^{\frac{n}{w}\cdot j} c_j. \tag{1.15}$$

Далее PAA-представление подвергается кодированию с помощью *символьной агрегатной аппроксимации (SAX, Symbolic Aggregate ApproXimation)* [140]. Символьным представлением (SAX-представлением) подпоследовательности $C = (c_1, c_2, \ldots, c_n)$ является *слово* $\hat{C} = (\hat{c}_1, \hat{c}_2, \ldots, \hat{c}_w)$, получаемое следующим образом. Пусть имеется символьный алфавит $\mathcal{A} = (\alpha_1, \alpha_2, \ldots, \alpha_{|\mathcal{A}|})$, где запись $|\mathcal{A}|$ означает мощность алфавита (α_1 ='a', α_2 ='b' и т.д.). Тогда

$$\hat{c}_i = \alpha_i \Leftrightarrow \beta_{j-1} \leqslant \hat{c}_i < \beta_j. \tag{1.16}$$

В формуле (1.16) числа β_i представляют собой *точки разделения (breakpoints)* [140], определяемые как упорядоченный список чисел $\mathcal{B} = (\beta_0, \beta_1, \ldots, \beta_{|\mathcal{A}|-1}, \beta_{|\mathcal{A}|})$, где $\beta_0 = -\infty$ и $\beta_{|\mathcal{A}|} = +\infty$, а площадь под кривой нормального распределения $N(0,1)$ между β_i и β_{i+1} равна $\frac{1}{|\mathcal{A}|}$. Точки разделения для различных значений параметра мощности алфавита могут быть получены из статистических таблиц [116, 117].

Последовательная реализация поиска диссонансов, предложенная Кеогом и др. [116, 117], представлена в алг. 1.2.

1.2. Применение СУБД для интеллектуального анализа данных

В настоящее время интеграция методов и алгоритмов интеллектуального анализа данных в реляционные СУБД является одной из актуальных задач в области методов управления данными [174, 180, 226].

Алг. 1.2. HOTSAX(IN T, n; OUT $pos_{bsf}, dist_{bsf}$)

```
1: for all C_i ∈ S_T^n do
2:     dist_min ← ∞
3:     for all C_j ∈ S_T^n and |i − j| ⩾ n do
4:         dist ← ED(C_i, C_j)
5:         if dist < dist_bsf then
6:             break
7:         end if
8:         if dist < dist_min then
9:             dist_min ← dist
10:        end if
11:    end for
12:    if dist_min > dist_bsf then
13:        dist_bsf ← dist_min
14:        pos_bsf ← i
15:    end if
16: end for
17: return {pos_bsf, dist_bsf}
```

Далее будут рассмотрены современные подходы к интеграции интеллектуального анализа данных в СУБД и методы параллельной обработки реляционных баз данных на основе концепции фрагментного параллелизма.

1.2.1. Интеграция анализа данных в СУБД

Исследования в области интеграции интеллектуального анализа данных в реляционные системы баз данных начаты в конце XX в. в работах Агравала (Agrawal) и Сараваджи (Sarawagi) [33,212], где предложен термин «связывание» (coupling) интеллектуального анализа данных и СУБД.

Хан (Han) предложил [99] различать следующие виды интеграции интеллектуального анализа данных в СУБД: слабое связывание, среднее связывание и сильное связывание.

При *слабом связывании (loose coupling)* система интеллектуального анализа данных использует сервисы СУБД для экспорта исходных данных из

хранилища и импорта результатов анализа обратно в хранилище данных. Данный подход использует большинство современных открытых систем для интеллектуального анализа данных: KNIME [42], Weka [88] и др.

При *среднем связывании (semitight coupling)* система интеллектуального анализа данных использует возможности СУБД для реализации некоторых примитивных операций, часто используемых при подготовке данных для интеллектуального анализа. В качестве таких операций могут фигурировать индексирование, соединение отношений, построение гистограмм, статистические вычисления (поиск максимума и минимума, среднего отклонения) и др. Помимо этого СУБД может обеспечивать хранение предварительно вычисленных и часто используемых промежуточных результатов интеллектуального анализа.

При *сильном связывании (tight coupling)* система интеллектуального анализа данных рассматривается как функциональная единица СУБД. В этом случае запросы и функции интеллектуального анализа данных оптимизируются на основе использования структур данных, схем индексирования и методов обработки запросов, встроенных в СУБД. Данный подход предпочтителен с точки зрения удобства прикладного программиста и конечного пользователя, но одновременно является наиболее трудоемким в реализации [99]. В рамках данного подхода осуществляется расширение стандартного синтаксиса SQL конструкциями для интеллектуального анализа данных либо разработка самостоятельного языка запросов для СУБД.

```
1 find association rules as HealthRuleSet
2   related to Salary, Age, isSmoker, Disease
3   from HealthDB
4   where Disease='Pneumonia' and Age>60
5     with support threshold=0.05
6     with confidence threshold=0.07
```

Рис. 1.1. Пример запроса на языке *DMQL*

В 1996 г. Хан (Han) и др. предложили язык *DMQL* [98], который предоставляет SQL-подобный синтаксис для записи запросов интеллектуального анализа данных. Примитивы *DMQL* позволяют определить данные, подле-

жащие анализу, решаемую задачу интеллектуального анализа (классификация, поиск ассоциативных правил и др.), семантические иерархии в анализируемых данных и пороговые значения параметров задачи (поддержка и др.). Пример запроса на языке *DMQL* приведен на рис. 1.1.

Язык *DMQL* позднее послужил основой для разработки целого ряда языков запросов интеллектуального анализа данных: язык для анализа временных данных *TQML* [61] Чена (X. Chen), 1998 г.; язык для анализа географических данных *GMQL* [100] Хана (Han), 1997 г.; язык для анализа пространственно-временных данных *ST-DMQL* [48] Богорны (Bogorny), 2009 г.

```
1 mine rule HealthRuleSet as
2   select distinct 1..n Disease as body,
3     1..1 isSmoker as head
4   from HealthDB
5   where body.Disease='Pneumonia' and body.Age>60
6   extracting rules with
7     support: 0.1
8     confidence: 0.3
```

Рис. 1.2. Пример запроса на языке *MINE RULE*

Meo (Meo) и др. в 1996 г. предложили SQL-подобный оператор *MINE RULE* [157], который предназначен для решения задачи поиска ассоциативных правил. Пример запроса с использованием оператора *MINE RULE* показан на рис. 1.2.

Позднее в 1999 г. в работе [109] Имилински (Imielinski) описал язык *MSQL*, представляющий собой расширение SQL для решения задачи поиска ассоциативных правил. В отличие от *DMQL*, язык *MSQL* предполагает не только нахождение ассоциативных правил, но и предоставляет возможность их сохранения в базе данных для последующих запросов. Соответствующие примеры запросов приведены на рис. 1.3.

Первое десятилетие XXI в. исследования в области языков запросов интеллектуального анализа данных представлено следующими работами. В корпорации Microsoft разработаны стандарт OLE DB for Data Mining и язык запросов *DMX (Data Mining Extensions)* [232], используемые в ее про-

```
GetRules(HealthDB)
  into HealthRuleSet R
  where R.Body in {(Disease=*), (Age=*), (Salary=*)}
    and R.Body has {(Disease='Pneumonia'), (Age>60)}
    and R.Consequent in {(isSmoker=*)}
    and Support>0.1
    and Confidence>0.7

SelectRules(HealthRuleSet)
  where Body has {(Disease='Pneumonia')}
    and {(Salary>0) and (Salary<=1000)}
    and Support>0.1
    and Confidence>0.7
```

Рис. 1.3. Пример запроса на языке MSQL

```
select
  PredictAssociation([HealthMiningModel].[AssocLines],
  INCLUDE_STATISTICS, 3)
  from [HealthMiningModel]
  natural prediction join (
    select
      60 as [Age],
      TRUE as [isSmoker],
      'Pneumonia' as [Disease]) as [AssocLines]
```

Рис. 1.4. Пример запроса на языке *DMX*

дукте MS SQL Server Analysis Services. Стандарт специфицирует интерфейс программирования приложений (Application Programming Interface, API) интеллектуального анализа данных. Язык *DMX* имеет SQL-подобный синтаксис (операторы определения и манипулирования данными и др.), однако его операндами являются не реляционные отношения, а модели интеллектуального анализа данных. Под моделью интеллектуального анализа данных понимается сочетание самих данных, алгоритма интеллектуального анализа данных и коллекции значений параметров и фильтров, управляющих использованием и обработкой данных. Пример запроса на языке *DMX* показан на рис. 1.4.

Малерба (Malerba) и др. в 2004 г. разработали язык *SDMQL* [148], который предназначен для проведения интеллектуального анализа пространственных данных и поддерживает запросы, связанные с классификацией и

поиском ассоциативных правил.

В 2013 г. Сан (Sun) и др. в работе [229] предложили расширить язык SQL оператором *CLUSTER BY* для кластеризации данных. Данная конструкция подразумевает выполнение группировки строк результата запроса в соответствии со специфицированным алгоритмом кластеризации, в отличие от стандартного оператора `GROUP BY`, где группировка осуществляется по точному совпадению значений в полях записей. Силва (Silva) и др. в работе [221] предложили схожий по назначению оператор *SIMILAR GROUP BY*, реализованный авторами в СУБД PostgreSQL.

В 2016 г. Краузе (Krause) и др. в работе [128] описали язык запросов для поиска шаблонов в графах, имеющий SQL-подобный синтаксис, и соответствующую систему интеллектуального анализа данных.

1.2.2. СУБД на основе фрагментного параллелизма

Вычислительная система с кластерной архитектурой (кластер) представляет собой набор рабочих станций, объединенных в систему с помощью одной из стандартных сетевых технологий на базе шинной архитектуры или коммутатора. Популярность вычислительных кластеров обусловлена тем, что данные системы обладают хорошим соотношением «цена/производительность», поскольку состоят из массово производимых и продающихся на рынке компонент. Исследования последних лет показывают, что кластеры могут быть эффективно использованы для хранения и обработки сверхбольших хранилищ данных [124, 133, 139].

Параллельные системы баз данных [71] обеспечивают распределенную обработку запросов на кластерных вычислительных системах [30, 124]. Базисной концепцией параллельных систем баз данных является *фрагментный параллелизм*. Начало исследованиям параллельной обработки баз данных на основе фрагментного параллелизма положили работы Граэфа (Graefe) [95] и ДеВитта (DeWitt) [71]. В соответствии с концепцией фрагмент-

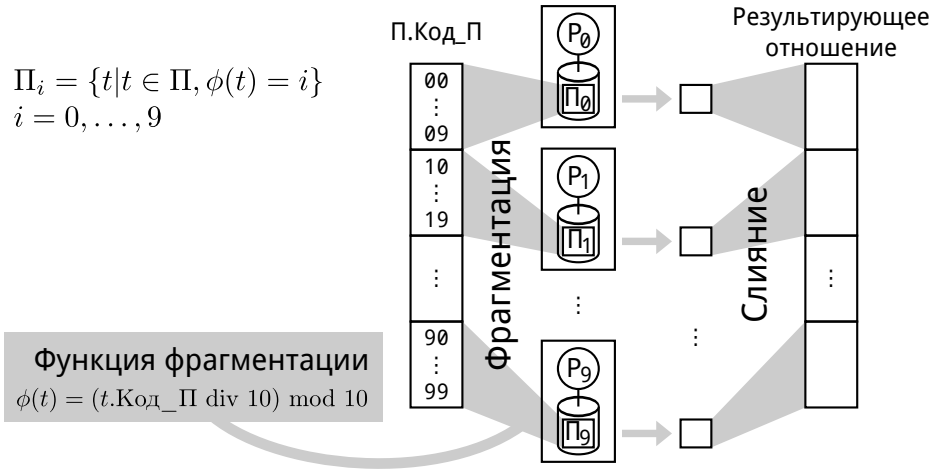


Рис. 1.5. Обработка запроса на основе фрагментного параллелизма

ного параллелизма *(partitioned parallelism)* [154] параллельная обработка запросов к базе данных выглядит следующим образом (см. рис. 1.5).

Все отношения реляционной базы данных подвергаются *горизонтальной фрагментации*, которая заключается в физическом распределении кортежей отношения по дискам узлов кластерной системы. Способ фрагментации отношения R определяется *функцией фрагментации*

$$\varphi_R : \{t \mid t \in R\} \to \{0, 1, \ldots, F-1\},$$

которая для кортежа отношения выдает номер вычислительного узла, где должен храниться этот кортеж. Величина F (количество вычислительных узлов кластера) называется *степенью фрагментации*.

Л.Б. Соколинским введена *атрибутная фрагментация* [23], предполагающая, что $\forall r \in R \varphi_R(r) = f_A(r.A)$, где $f_A : D_A \to \{0, \ldots, k-1\}$ является функцией фрагментации, определенной на домене атрибута A. Атрибутная фрагментация допускается основными реляционными операциями (естественное соединение, группировка, удаление дубликатов).

Запрос параллельно выполняется на всех вычислительных узлах в виде набора *параллельных агентов* [5], каждый из которых обрабатывает выделенные ему фрагменты отношений на соответствующем вычислительном узле. Обработка запроса состоит из трех этапов (см. рис. 1.6). На первом

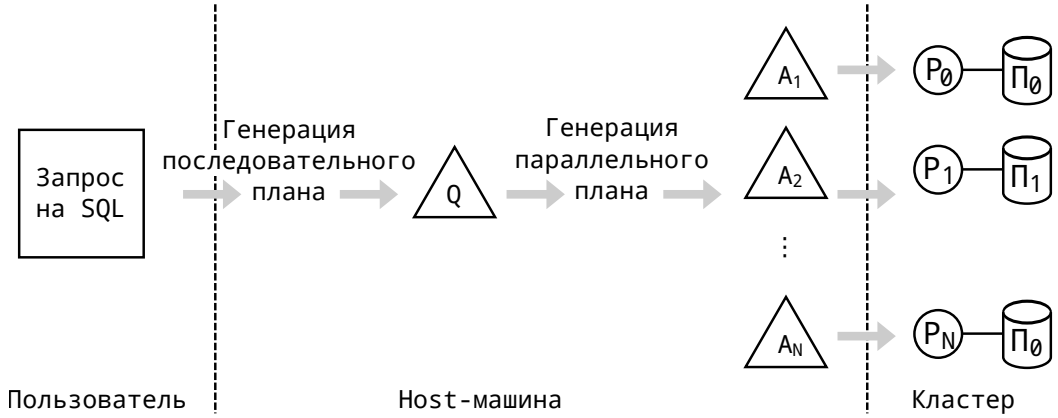


Рис. 1.6. Схема обработки запроса в параллельной СУБД. Q — последовательный физический план, A_i — параллельный агент, P_i — вычислительный узел, D_i — диск

этапе SQL-запрос передается пользователем на выделенную host-машину (роль которой может играть любой узел вычислительного кластера), где транслируется в некоторый последовательный физический план [57].

На втором этапе последовательный физический план преобразуется в параллельный план, представляющий собой совокупность параллельных агентов. Это достигается путем вставки оператора обмена *exchange* в соответствующие места плана запроса. Структура оператора обмена и схема построения параллельного плана запроса обсуждаются ниже.

На третьем этапе параллельные агенты пересылаются с host-машины на соответствующие вычислительные узлы, где интерпретируются исполнителем запросов. Результаты выполнения агентов объединяются корневым оператором *exchange* на нулевом узле, откуда передаются на host-машину, выдающую итоговый результат.

Независимая обработка экземплярами параллельной СУБД фрагментов базы данных, расположенных на различных узлах кластерной вычислительной системы, не исключает необходимость обменов кортежами между экземплярами СУБД в процессе выполнения запроса. Например, обмены требуются при выполнении операции естественного соединения двух отношений по общему атрибуту, когда кортежи, для которых выполняется условие соединения, хранятся в разных фрагментах базы данных (на

разных вычислительных узлах кластерной системы). Для разрешения подобных ситуаций строится *параллельный план запроса*, представляющий собой последовательный план, в нужные места которого вставляется специальный оператор обмена.

Оператор обмена (*exchange*) впервые исследован Граэфом (Graefe) в работе [94] для параллельных СУБД, построенных на базе вычислительных систем с общей памятью. Позднее Л.Б. Соколинским [222] описана структура и принципы реализации оператора обмена для параллельных СУБД на базе вычислительных систем с кластерной архитектурой.

Оператор *exchange* инкапсулирует обмены данными между экземплярами СУБД, запущенных на различных узлах кластерной вычислительной системы. Реализация данного оператора, так же как и у других операторов физической алгебры в модифицируемой СУБД, основана на итераторной модели. Оператор обмена имеет два дополнительных атрибута: порт и функция пересылки. Атрибут *порт* обеспечивает идентификацию операторов обмена в рамках одного плана запроса.

Функция пересылки $\psi(t)$ возвращает номер вычислительного узла кластерной системы, на котором должен быть обработан данный кортеж t. Если кортеж t требуется обработать на текущем вычислительном узле, то кортеж передается в родительский узел плана запроса, иначе выполняется пересылка данного кортежа на вычислительный узел с номером $\psi(t)$ (где получение этого кортежа будет осуществляться оператором обмена с идентичным номером порта).

На рис. 1.7 приведена структура оператора обмена (стрелки отражают направление следования кортежей). Операторы *split*, *scatter*, *gather* и *merge*, составляющие оператор обмена, также реализуются на основе итераторной модели.

Оператор *split* представляет собой бинарный оператор, который разделяет входные кортежи на две категории: «свои» и «чужие». «Свои» кортежи будут обрабатываться на текущем вычислительном узле и помещаются в выходной буфер оператора *split*. «Чужие» кортежи помещаются в вы-

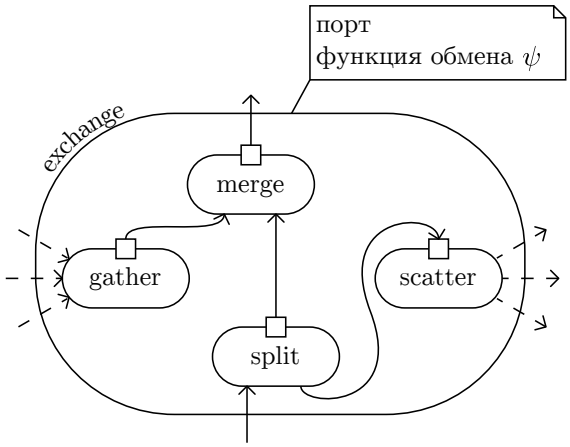


Рис. 1.7. Структура оператора обмена

ходной буфер оператора *scatter*, поскольку они должны быть отправлены для обработки на другие вычислительные узлы. Оператор *scatter* представляет собой нульарный оператор, который вычисляет значение функции пересылки для кортежей своего выходного буфера и отправляет их на соответствующие вычислительные узлы, используя указанный порт обмена. Оператор *gather* представляет собой нульарный оператор, который выполняет получение кортежей со всех процессорных узлов, отличных от текущего, по указанному порту обмена, помещая их в свой выходной буфер. Оператор *merge* определяется как бинарный оператор, который попеременно извлекает кортежи из выходных буферов операторов *gather* и *split*, помещая их в свой выходной буфер.

Оригинальный исполнитель запросов последовательной СУБД будет выполнять оператор обмена подобно другим операторам физической алгебры, не подразумевая скрытого в нем параллелизма. Параллельное выполнение запроса будет иметь место в силу построения корректного параллельного плана запроса.

Параллельный план запроса представляет собой последовательный план запроса, в котором между внутренними узлами добавлены узлы с операторами обмена. Оператор обмена вставляется в те места последовательного плана, где требуется обеспечить пересылки кортежей между вычислительными узлами по правилу, заданному функцией пересылки, чтобы деятель-

ность исполнителя запросов позволила получить корректный результат запроса.

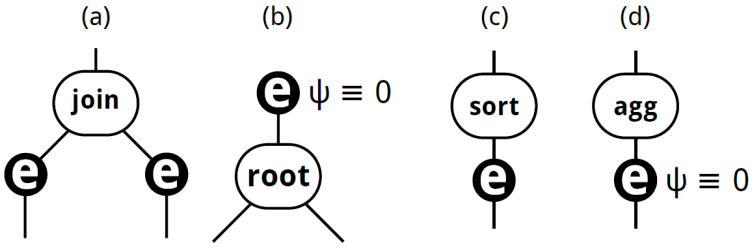


Рис. 1.8. Параллельный план запроса

На рис. 1.8 показаны примеры параллельных планов запроса, содержащих оператор обмена. При выполнении соединения требуется, чтобы оба соединяемых отношения были фрагментированы по атрибуту соединения, поскольку для того, чтобы условие соединения выполнилось, соединяемые кортежи должны располагаться на одном вычислительном узле (случай a). Оператор обмена также вставляется в корень плана, чтобы обеспечить сборку конечного результата на узле-координаторе (случай b). Поскольку порядок кортежей, поступающих от оператора обмена, в общем случае неизвестен, вставка оператора обмена производится под оператором сортировки (случай c). Для него подбирается функция пересылки, зависящая от атрибута, по которому производится сортировка, кроме случая, когда оператор сортировки является корневым. Аналогичным образом производится вставка оператора обмена ниже оператора агрегации (случай d).

1.3. Обзор работ по теме диссертации

В данном разделе делается обзор научных работ, наиболее близко относящихся к теме диссертации.

1.3.1. Системы анализа данных в СУБД

Ванг (Wang) и др. разработали систему интеллектуального анализа данных *ATLAS* [239]. Данная система поддерживает одноименный язык,

являющийся надстройкой над SQL. Язык ATLAS добавляет в SQL поддержку пользовательских функций и функций, возвращающих в качестве значения реляционную таблицу. На языке ATLAS реализованы алгоритм поиска шаблонов Apriori, алгоритм кластеризации DBSCAN и классификация посредством деревьев решений.

Хеллерштейн (Hellerstein) и др. разработали библиотеку *MADlib* [103] с открытым исходным кодом для интеллектуального анализа данных в реляционных СУБД PostgreSQL и Greenplum. MADlib предоставляет алгоритмы, адаптированные для использования в реляционной СУБД и не требующие экспорта и импорта данных внешних аналитических приложений. Реализация MADlib выполнена на SQL и языке программирования Python.

В работе [176] описана техника реализации хранимых функций, выполняющих агрегатные операции относительно столбцов реляционных таблиц (в противоположность стандартным конструкциям SQL, выполняющим агрегатные операции над строками), которые используются в SQL-реализациях алгоритмов интеллектуального анализа данных.

В цикле работ [45–47] Блокилом (Blockeel), Гоэталсом (Goethals) и др. предложен подход к интеллектуальному анализу данных в СУБД на основе виртуальных аналитических представлений. *Виртуальное аналитическое представление (virtual mining view)* создается как именованный запрос к таблицам базы данных и другим представлениям, который обеспечивает логическое хранение (в отличие от физического хранения таблиц базы данных) результатов интеллектуального анализа данных. Описано построение виртуальных аналитических представлений для поиска шаблонов, классификации с помощью деревьев решений и др.

Научная группа под руководством Ордонеза провела следующие исследования в данном направлении. В работе [182] предложена реализация метода главных компонент в виде хранимых процедур для параллельной СУБД, использующих функции библиотеки Intel Math Kernel Library. В работе [179] предложен способ ускорения вычисления Байесовской модели для выбора переменной в приложении к алгоритму линейной регрессии на

основе использования хранимых процедур, адаптированных для применения в параллельной СУБД.

В работе [181] Ордонезом и др. предложена облачная система интеллектуального анализа данных на основе реляционной СУБД. На локальной машине запускается реляционная СУБД, подключающаяся к облаку. База данных хранится и обрабатывается в облаке, а в локальную СУБД передаются только результаты анализа. Помимо возможностей обработки данных только на локальной машине или только в облаке, система поддерживает режим гибридного исполнения, когда выполняется распределение вычислительной нагрузки между облаком и локальной СУБД.

Ордонезом и др. также разработана система для интеллектуального анализа данных, основанная на использовании реляционной СУБД и хранимых процедур [178]. Система является промежуточным звеном между конечным пользователем и СУБД. С помощью графического интерфейса конечный пользователь специфицирует задачу интеллектуального анализа данных, ее параметры и таблицы исходных данных. Система осуществляет генерацию текста и запуск соответствующей хранимой процедуры, которая выполняет интеллектуальный анализ, и последующую визуализацию результатов. Хранимые процедуры реализуют алгоритмы решения различных задач интеллектуального анализа данных.

Махаян (Mahajan) и др. разработали аналитическую систему DAnA [147], которая выполняет автоматическое преобразование запросов на выполнение анализа данных в код для выполнения на вычислительных системах FPGA. Реализация данного преобразования выполняется с помощью пользовательской функции на SQL, использующей язык Python. Система DAnA предполагает интеграцию в СУБД на основе специализированных аппаратных устройств, называемых *страйдерами (striders)*. Страйдер имеет прямой интерфейс доступа к буферному пулу СУБД и выполняет извлечение, очистку и обработку кортежей данных, которые затем передаются на FPGA для параллельного исполнения аналитического алгоритма.

Речкалов Т.В. в работе [20] предложил использование XML-разметки

алгоритма интеллектуального анализа данных, выраженного на языке SQL. Разметка позволяет выполнить автоматическую генерацию хранимых процедур на языке SQL, реализующих данный алгоритм, в зависимости от специфицированных пользователем таблиц исходных данных и параметров алгоритма.

1.3.2. Алгоритмы кластеризации

Кластеризация на базе техники медоидов

Алг. 1.3. PAM(IN X, k; OUT C)

▷ Фаза *Build*

1: Инициализировать C

▷ Фаза *Swap*

2: **repeat**
3: Найти $\{T_{min}, c_{min}, x_{min}\}$
4: Поменять местами c_{min} и x_{min}
5: **until** $T_{min} < 0$
6: **return** C

Алгоритм *PAM* [115] представляет собой разделительный алгоритм кластеризации, в котором в качестве центров кластеров выбираются только кластеризуемые объекты (медоиды). Реализация *PAM* представлена в алг. 1.3. Алгоритм состоит из двух фаз: *Build* и *Swap*. На фазе *Build* выполняется инициализация множества медоидов, которые затем уточняются на фазе *Swap*.

На фазе *Build* в качестве первого медоида c_1 выбирается объект, сумма расстояний от которого до всех остальных объектов является наименьшей:

$$c_1 = \arg \min_{1 \leqslant h \leqslant n} \sum_{j=1}^{n} \rho(x_h, x_j). \tag{1.17}$$

Для выбора остальных медоидов используется целевая функция $E : X \times X \to \mathbb{R}$, определяемая как сумма расстояний от каждого объекта до ближайшего ему медоида:

$$E = \sum_{j=1}^{n} \min_{1 \leqslant i \leqslant k} \rho(c_i, x_j). \tag{1.18}$$

При выборе каждого последующего медоида производится вычисление целевой функции относительно объектов, ранее выбранных в качестве медоидов, и каждого из еще не выбранных объектов, для минимизации целевой функции:

$$c_2 = \arg \min_{1 \leqslant h \leqslant n} \sum_{j=1}^{n} \min(\rho(c_1, x_j), \rho(x_h, x_j)), \tag{1.19}$$

$$c_3 = \arg \min_{1 \leqslant h \leqslant n} \sum_{j=1}^{n} \min(\min_{1 \leqslant \ell \leqslant 2} (\rho(c_\ell, x_j)), \rho(x_h, x_j)), \tag{1.20}$$

$$\ldots$$

$$c_k = \arg \min_{1 \leqslant h \leqslant n} \sum_{j=1}^{n} \min(\min_{1 \leqslant \ell \leqslant k-1} (\rho(c_\ell, x_j)), \rho(x_h, x_j)). \tag{1.21}$$

На фазе *Swap* алгоритм пытается изменить множество медоидов C таким образом, чтобы улучшить значение целевой функции E. *PAM* выполняет поиск пары объектов (c_{min}, x_{min}), минимизирующих целевую функцию. Для этого перебираются все пары (c_i, x_h), где c_i — медоид, а x_h не является медоидом, и вычисляется изменение целевой функции при исключении c_i из множества медоидов и включении вместо него объекта x_h. Указанное изменение обозначается как T_{ih}, а его минимальное значение, достигаемое на паре (c_{min}, x_{min}), как T_{min}. Если $T_{min} > 0$, тогда множество C не может быть улучшено, и алгоритм завершается.

Для вычисления T_{ih} используются множества $D, S \in \mathbb{R}^n$, определяемые следующим образом:

$$\begin{aligned} D &= \{d_i \mid d_i = \rho(x_i, c_{min_1}), c_{min_1} := \arg\min_{c_j \in C} \rho(x_i, c_j)\} \\ S &= \{s_i \mid s_i = \rho(x_i, c_{min_2}), c_{min_2} := \arg\min_{c_j \in C \setminus c_{min_1}} \rho(x_i, c_j)\}. \end{aligned} \tag{1.22}$$

Иными словами, D — это множество расстояний от каждого объекта до ближайшего медоида, S — множество расстояний от каждого объекта до второго ближайшего медоида. Вклад не-медоида x_j в вычисление T_{ih} при замене c_i на x_h обозначается как δ_{jih}, и T_{ih} вычисляется следующим образом:

$$T_{ih} = \sum_{j=1}^{n} \delta_{jih}. \tag{1.23}$$

Величина δ_{jih} вычисляется, как показано в алг. 1.4.

Алг. 1.4. δ_{jih}(IN x_j, c_i, x_h, d_j, s_j; OUT δ)

```
if ρ(x_j, c_i) > d_j and ρ(x_j, x_h) > d_j then
    δ ← 0
else if ρ(x_j, c_i) = d_j then
    if ρ(x_j, x_h) < s_j then
        δ ← ρ(x_j, x_h) − d_j
    else
        δ ← s_j − d_j
    end if
else if ρ(x_j, x_h) < d_j then
    δ ← ρ(x_j, x_h) − d_j
end if
return δ
```

Эспеншейд (Espenshade) и др. в работе [82] и Кохлофф (Kohlhoff) и др. в работе [123] рассмотрели параллельные реализации для GPU алгоритма *k-Medoids*, идеологически близкого к *PAM*. Параллельная версия алгорит-

ма *k-Means* для многоядерных систем Intel Xeon Phi рассматривалась в работах Ли (Lee) и др. [132], Ву (Wu) и др. [246] и Яроша (Jaros) и др. [110].

В работе [131] Курте (Kurte) представлен алгоритм *GPUPAM* — параллельная реализация алгоритма *PAM* для графического процессора. Алгоритм *GPUPAM* предполагает создание двумерного массива нитей CUDA размера $k \times (n - k)$ для подсчета целевой функции, где каждая нить выполняет задачу обмена местами медоида и не-медоида.

Нечеткая кластеризация данных

В области параллельных алгоритмов нечеткой кластеризации данных можно выделить следующие исследования. Людвиг (Ludwig) предложил параллельный алгоритм *MR-FCM* в работе [145] на основе парадигмы MapReduce [68]. Алгоритм использует два потока работ (MapReduce job): первый вычисляет координаты центроидов, второй — расстояния между кластеризуемыми объектами, необходимые для обновления степеней принадлежности объектов кластерам. Эксперименты, однако, показывают [145], что *MR-FCM* уступает по производительности алгоритму нечеткой кластеризации данных *Mahout Fuzzy k-Means (Mahout FKM)* [249], входящему в состав библиотеки распределенных алгоритмов анализа данных Apache Mahout [164].

Хидри (Hidri) и др. предложили алгоритм *WCFC (Weighted Consensus Fuzzy Clustering)* в работе [105]. В алгоритме данные распределяются по узлам вычислительного кластера. Фрагмент на каждом из узлов разделяется на сегменты, где размер сегмента допускает его загрузку в оперативную память узла. Далее локально выполняется кластеризация сегмента. Результаты локальной кластеризации сегментов каждого фрагмента пересылаются на узел-мастер, который выполняет их агрегацию и преобразование в результат кластеризации исходного набора данных в целом. Алгоритм *WCFC* показывает лучшую производительность в экспериментах [105], чем алгоритм *MR-FCM* [145].

Гадири (Ghadiri) и др. в работе [93] предложили алгоритм *BigFCM*, основанный на парадигме MapReduce [68] и реализованный на платформе Hadoop [67]. Эксперименты показали [93], что алгоритм *BigFCM* существенно опережает по производительности алгоритм *Mahout FKM* [249].

Кластеризация данных в СУБД и разбиение графов

Исследования по реализации алгоритмов **кластеризации данных на SQL** представлены следующими работами. Ордонезом (Ordonez) в работах [171, 172] 2004–2006 гг. на языке SQL реализован классический алгоритм кластеризации *k-Means*. Им же в сотрудничестве с коллегами в работах [151, 175] выполнена реализация на SQL алгоритма кластеризации *EM (Expectation-Maximization)* [70].

В работе [134] Лепиниоти (Lepinioti) в 2007 г. разработал алгоритм иерархической кластеризации *Cobweb/IDX*. Реализация выполнена на языке PL/SQL для СУБД Oracle. Данный алгоритм является инкрементальным (поддерживает кластеризацию по мере появления новых данных).

Одним из эффективных алгоритмов кластеризации графов является *алгоритм Кариписа—Кумара* [114], основанный на алгоритме Кернигана—Лина [118]. Алгоритм Кариписа—Кумара предполагает многоуровневую схему разбиения графа (multilevel partitioning), которая состоит из трех стадий: огрубление, начальное разбиение, уточнение. Огрубление (coarsening) заключается в уменьшении количества вершин и ребер в графе путем «склеивания» сильно связанных вершин и удаления возникающих при этом петель. В рамках начального разбиения огрубленный граф подвергается анализу обычными алгоритмами, результатом которого является грубое разбиение исходного графа. На финальной стадии уточнения (uncoarsening) выполняется трансформация грубого разбиения в разбиение исходного графа с помощью какой-либо эвристики.

На сегодня разработано большое количество как последовательных, так и параллельных алгоритмов разбиения графов (см., например, обзор [87]). Разбиение графа на основе генетических алгоритмов исследовано в рабо-

те [119]. В работе [228] обсуждается параллельный алгоритм разбиения графов для многоядерных процессоров. В работе [59] предложен подход к решению задачи разбиения сверхбольших графов на базе концепции облачных вычислений.

Одними из наиболее цитируемых разработок для параллельного разбиения сверхбольших графов являются системы *ParMETIS* [113] и PT-Scotch [63], свободно распространяемые на уровне исходных кодов.

В работе [234] представлен параллельный алгоритм многоуровневого разбиения графа, использующий хранение данных на диске. Данный алгоритм в экспериментах [234], однако, уступает по производительности системе ParMETIS.

Имеются распределенные алгоритмы разбиения графа, основанные на предварительном выявлении сообществ в графе [192,256]. Под сообществом (community) графа понимают подмножество вершин этого графа, в котором вершины плотно связаны между собой и редко связаны с другими частями графа [242]. Данные алгоритмы реализуются на основе парадигмы MapReduce [68] и каркасов Spark [254] либо Hadoop [67].

1.3.3. Алгоритмы поиска шаблонов

Алгоритм *Dynamic Itemset Counting (DIC)* [50] является модификацией классического алгоритма поиска частых наборов *Apriori* [34]. По сравнению с алгоритмом-предшественником, *DIC* (см. алг. 1.5) пытается сократить количество проходов по множеству транзакций и сохранить при этом относительно небольшое количество наборов, поддержка которых подсчитывается в рамках одного прохода.

DIC выполняет логическое разбиение множества транзакций $\mathcal{D}$ на блоки, состоящие из $\lceil \frac{|\mathcal{D}|}{M} \rceil$ транзакций, где число транзакций в блоке M ($1 \leqslant M \leqslant |\mathcal{D}|$) является параметром алгоритма.

DIC поддерживает четыре вида наборов, для названия которых используются метафоры: *пунктирные окружности*, *пунктирные квадраты*,

Алг. 1.5. DIC(IN $\mathcal{D}$, $minsup$, M, OUT $\mathcal{L}$)

▷ Инициализация множеств наборов

1: $SolidBox \leftarrow \varnothing$; $SolidCircle \leftarrow \varnothing$; $DashedBox \leftarrow \varnothing$
2: $DashedCircle \leftarrow \mathcal{I}$
3: **while** $DashedCircle \cup DashedBox \neq \varnothing$ **do**

▷ Чтение транзакций

4: Read($\mathcal{D}, M, Chunk$)
5: **if** EOF($\mathcal{D}$) **then**
6: Rewind($\mathcal{D}$)
7: **end if**
8: **for all** $T \in Chunk$ **do**

▷ Подсчет поддержки наборов

9: **for all** $I \in DashedCircle \cup DashedBox$ **do**
10: **if** $I \subseteq T$ **then**
11: $support(I) \leftarrow support(I) + 1$
12: **end if**
13: **end for**

▷ Генерация наборов-кандидатов

14: **for all** $I \in DashedCircle$ **do**
15: **if** $support(I) \geqslant minsup$ **then**
16: $DashedBox \leftarrow DashedBox \cup I$
17: **for all** $i \in \mathcal{I}$ **do**
18: $C \leftarrow I \cup i$
19: **if** $\forall s \subseteq C \; s \in SolidBox \cup DashedBox$ **then**
20: $DashedCircle \leftarrow DashedCircle \cup C$
21: **end if**
22: **end for**
23: **end if**
24: **end for**

▷ Проверка завершения полного прохода

25: **for all** $I \in DashedCircle \cup SolidBox$ **do**
26: **if** IsPassCompleted(I) **then**
27: **switch** Shape(I)
28: *dashed*: $DashedBox \leftarrow DashedBox \cup I$
29: *solid*: $SolidBox \leftarrow SolidBox \cup I$
30: **end switch**
31: **end if**
32: **end for**
33: **end for**
34: **end while**
35: $\mathcal{L} \leftarrow SolidBox$
36: **return** $\mathcal{L}$

сплошные окружности и *сплошные квадраты*. Для «пунктирных» наборов необходимо выполнить подсчет поддержки, в то время как для «сплошных» наборов подсчет поддержки закончен. «Квадрат» соответствует частому набору, «окружность» — редкому. В соответствии с этим определяются четыре непересекающихся множества наборов: *DashedCircle*, *DashedBox*, *SolidCircle* и *SolidBox*. Множества *DashedCircle* и *DashedBox* содержат неподтвержденные редкие и неподтвержденные частые наборы соответственно, а множества *SolidCircle* и *SolidBox* — подтвержденные редкие и подтвержденные частые наборы соответственно. При инициализации множества *DashedBox*, *SolidCircle* и *SolidBox* полагаются пустыми, а множество *Dashed Circle* заполняется 1-наборами из множества объектов $\mathcal{I}$.

При обработке транзакций блока *DIC* вычисляет поддержку «пунктирных» наборов из множеств *DashedCircle* и *DashedBox*. По завершении обработки блока наборы, поддержка которых сравнялась или превысила порог *minsup* перемещаются из множества *DashedCircle* в *DashedBox*. В множество *DashedCircle* добавляется каждый набор, который является надмножеством таких наборов из *DashedBox*, любое подмножество которых является «квадратом». После обработки последнего блока выполняется переход к первому блоку транзакций. Алгоритм завершается, когда множества *DashedCircle* и *DashedBox* становятся пустыми.

Параньяп-Водител (Paranjape-Voditel) и др. разработали алгоритм *DIC-OPT* [189], который является параллельной версией алгоритма *DIC* [50] для вычислительных систем с распределенной памятью. Основная идея алгоритма заключается в том, что каждый вычислительный узел выполняет рассылку сообщений, содержащих значения поддержки наборов, всем остальным узлам по завершении обработки очередного блока из M транзакций. Авторы провели эксперименты на вычислительной системе из 12 узлов, где *DIC-OPT* показал сублинейное ускорение.

Чеунг (Cheung) и др. в работе [62] представили алгоритм *APM*, который является модификацией алгоритма DIC для SMP-систем (вычислительных систем с общей памятью). Процессоры SMP-системы динамически генери-

руют наборы-кандидаты независимо друг от друга и не требуют синхронизации. Транзакции распределяются по узлам системы. Эксперименты, проведенные на вычислительной системе Sun Enterprise 4000 из 12 узлов, показали, что APM опережает параллельные реализации классического алгоритма *Apriori*. Однако, ускорение APM постепенно снижается до 4, когда количество задействованных узлов больше четырех.

Шлегел (Schlegel) и др. предложили алгоритм *mcEclat* [215], который является параллельной версией *Eclat* [255] для сопроцессора Intel Xeon Phi. *mcEclat* использует представление транзакций в виде вертикальных битовых карт: *tid* всех транзакций, в которых присутствует данный объект, преобразуются в битовую карту этого объекта, где в соответствующих позициях биты установлены в 1. Для вычисления поддержки набора над битовыми картами входящих в набор объектов выполняется логическая побитовая операция `AND` и затем подсчитывается количество битов результата, установленных в 1. Эксперименты показали ускорение алгоритма до 100 на 240 нитях сопроцессора, однако реализация не в полной мере использует возможности векторизации вычислений Xeon Phi и не опережает себя на платформе двухпроцессорной системы Intel Xeon.

В работах Кумара (Kumar) и др. [129], Бардика (Burdick) и др. [51] и Донга (Dong) и др. [75] предложены различные последовательные алгоритмы поиска частых наборов на основе использования битовых карт: *BitwiseDIC* (версия алгоритма DIC [50]), *MAFIA* и *BitTableFI* соответственно.

1.3.4. Алгоритмы анализа временных рядов

Одним из наиболее перспективных направлений ускорения ***поиска похожих подпоследовательностей*** временных рядов является распараллеливание наиболее быстрого из существующих последовательных алгоритмов *UCR-DTW* [200].

В работе [216] представлены две параллельные реализации алгоритма *UCR-DTW* для многоядерного процессора: на основе использования библиотеки ThreadPool [274] и на базе технологии OpenMP. В обеих реализациях чтение исходных данных и вычисление меры *DTW* выполняются посредством двух параллельных секций. Секция, выполняющая чтение, помещает прочитанные данные в рабочую очередь, доступную по чтению для секции, выполняющей вычисления. Вычисления выполняются несколькими нитями. Нить-вычислитель выполняет действия, предусмотренные оригинальным алгоритмом, последовательно. Результаты экспериментов, проведенные авторами для случаев с одной, четырьмя и восемью нитями-вычислителями, показывают, что удалось достигнуть следующих максимальных значений ускорения последовательного алгоритма: в 1.6 раза, в 3.6 раза и 3.4 раза соответственно.

В работе [233] рассмотрен 32-битный встраиваемый процессор, который обеспечивает специализированный набор инструкций для вычисления меры *DTW* в приложениях интеллектуального анализа временных рядов. Данная разработка способна обеспечить превосходство в быстродействии до 5 раз и сокращение энергопотребления до 80% по сравнению с последовательной реализацией, однако предназначена для обработки сенсорных данных в приложениях Интернета вещей. В работе [219] подпоследовательности, начинающиеся с разных позиций временного ряда, направляются для вычисления меры *DTW* на различные процессоры Intel Xeon.

Реализация на GPU [258] распараллеливает создание матрицы трансформации шкалы времени, однако путь трансформации вычисляется последовательно. В работе [213] предложена GPU-реализация, использующая те же идеи, что и в работе [219].

Реализация для FPGA, описанная в работе [213], предлагает поиск похожих подпоследовательностей, не использующий предварительную обработку данных. Приложение, осуществляющее поиск, генерируется с помощью инструмента C-to-VHDL и ввиду отсутствия знания внутреннего устройства FPGA не может быть применено к задачам большой размерности.

Для преодоления указанных проблем в работе [241] предложен потоково-ориентированный фреймворк, в котором реализован крупнозернистый параллелизм путем повторного использования данных различных вычислений меры DTW.

В работе [106] рассмотрена реализация алгоритма поиска похожих подпоследовательностей, использующая как центральный процессор, так и GPU. GPU выполняет каскад оценок для отбрасывания заведомо непохожих подпоследовательностей и вычисление меры DTW, процессор — z-нормализацию данных. Данная разработка показывает большую производительность, чем другие реализации, использующие GPU. Несмотря на то, что в данной работе в качестве центрального процессора использовался гибридный процессор AMD APU (объединение центрального процессора с графическим в одном кристалле), GPU в вычислениях фактически не использовался, хотя на него можно возложить нагрузку по выполнению каскада оценок.

В работе [216] описано распараллеливание алгоритма $UCR\text{-}DTW$ для вычислительного кластера на основе использования фреймворка Apache Spark [217]. В рамках Apache Spark приложение запускается как задача, координируемая мастер-узлом вычислительного кластера, на котором установлен соответствующий драйвер. Алгоритм предполагает фрагментацию временного ряда, однако количество фрагментов не совпадает с количеством вычислительных узлов кластерной системы. Фрагменты сохраняются в виде отдельных файлов, доступных всем узлам в силу использования распределенной файловой системы HDFS (Hadoop Distributed File System). Каждый фрагмент обрабатывается отдельной нитью, реализующей последовательный алгоритм $UCR\text{-}DTW$. Количество процессов, запускаемых на узле кластера, равно количеству ядер процессора на данном узле.

Алгоритмы поиска диссонансов во временном ряде представлены следующими работами. Янков, Кеог и др. в работе [252] представили алгоритм поиска диссонансов для временного ряда, хранящегося на диске, а не в оперативной памяти (далее для краткости данный алгоритм обозначается

как *DADD, Disk Aware Discord Discovery*). В алгоритме *DADD* вводится понятие *диапазонного диссонанса (range discord)*, означающее диссонанс, который имеет расстояние по меньшей мере r до своего ближайшего соседа, где r — наперед заданный параметр. Алгоритм *DADD* состоит из двух фаз: поиск и отсеивание кандидатов в диссонансы, — каждая из которых требует одну операцию полного сканирования данных на диске. Для нахождения значения параметра r применяется алгоритм *HOTSAX* [116,117] следующим образом. С помощью равномерной выборки получают подпоследовательность исходного временного ряда, допускающую размещение в оперативной памяти. Далее алгоритм HOTSAX находит диссонанс в указанной подпоследовательности, и значением r полагается расстояние от найденного диссонанса до его ближайшего соседа. Алгоритм способен обрабатывать временные ряды, которые не могут быть размещены в оперативной памяти, однако до половины общего времени выполнения занимают дисковые операции чтения и записи данных [253].

В работе [245] Вудбридж (Woodbridge) и др. представили подход для обнаружения диссонансов в данных ЭКГ, основанный на применении параллельной СУБД в составе программно-аппаратного комплекса IBM PureData for Analytics. Параллельная СУБД предполагает горизонтальную фрагментацию таблицы с данными ЭКГ пациентов по узлам кластерной вычислительной системы. На каждом узле выполняется извлечение данных ЭКГ и поиск аномалий. Поиск аномалий реализуется как хранимая процедура на языке pgPL/SQL, которая использует рассмотренный выше алгоритм *HOTSAX* [116,117].

Алгоритмы Хуанга (Huang) и др. *PDD (Parallel Discord Discovery)* [107] и *DDD (Distributed Discord Discovery)* Ву (Wu) и др. [248] используют платформу кластера Spark [254] для параллельного поиска диссонансов. В данных алгоритмах временной ряд разбивается на фрагменты, каждый из которых обрабатывается отдельным узлом Spark-кластера.

Алгоритм *PDD* использует парадигму мастер-рабочие. На первой фазе алгоритма выполняется вычисление оценки расстояния подпоследователь-

ностей ряда до ближайшего соседа (которая, как в алгоритме *HOTSAX*, используется для отбрасывания бесперспективных кандидатов). Узел-мастер получает от узлов-рабочих локальные оценки, выбирает из них наибольшую оценку и рассылает ее рабочим. На второй фазе алгоритм сканирует все подпоследовательности ряда для нахождения диссонанса. При этом подпоследовательности, которые отсутствуют в текущем фрагменте и пересекаются друг с другом, объединяются в пакет и пересылаются на текущий узел.

Алгоритм *DDD* реализуется как распараллеливание алгоритма *DADD*, выполняемое следующим образом. Каждый узел выполняет поиск кандидатов в диапазонные диссонансы в своем фрагменте. Далее полученные кандидаты объединяются и направляются на каждый узел, где проходят отсеивание. Объединение отброшенных кандидатов исключается из объединения кандидатов в диапазонные диссонансы, давая результирующее множество диссонансов.

В своей более поздней работе [253] Янков, Кеог и др. описали параллельную версию алгоритма *DADD* на базе парадигмы MapReduce [68] (далее для краткости этот алгоритм обозначается как *MR-DADD*). По сравнению с алгоритмом *DDD*, алгоритм *MR-DADD* более эффективно реализует фазу отсеивания, используя в ней технику подсчета расстояния с ранним окончанием вычислений, которая заключается в следующем. Пусть S_i и C_j — подпоследовательность ряда и кандидат в диссонансы соответственно. Тогда вычисление евклидова расстояния $ED(S_i, C_j) = \sum_{k=1}^{n} \sqrt{(s_{i_k} - c_{j_k})^2}$ следует прекратить на позиции $k = p$ $(1 \leqslant p \leqslant n)$, если $\sum_{k=1}^{p} (s_{i_k} - c_{j_k})^2 \geqslant dist_{C_j}^2$, где $dist_{C_j}$ — текущее расстояние до ближайшего соседа C_j.

1.4. Выводы по главе 1

Методы интеллектуального анализа данных направлены на обнаружение в данных скрытых закономерностей (трендов и аномалий), которые используются для принятия решений. В рамках данного исследования рас-

сматриваются задачи кластеризации, поиска шаблонов (ассоциативных правил) и анализ временных рядов.

Одной из актуальных задач в данной области является интеграция методов и алгоритмов интеллектуального анализа данных в реляционные СУБД. Это обусловлено следующими основными причинами. С одной стороны, на рынке СУБД доминирующими являются системы на основе реляционной модели данных, вследствие чего данные потребителей, подлежащие анализу, хранятся, как правило, в реляционных хранилищах данных. С другой стороны, большинство существующих классических алгоритмов интеллектуального анализа данных предполагают размещение анализируемых данных в оперативной памяти, и их использование требует значительных накладных расходов, связанных с предварительным экспортом анализируемых данных из хранилища данных во внешнюю аналитическую систему и импортом результатов работы этой системы обратно в хранилище.

Наиболее перспективным подходом к решению указанной проблемы является *сильносвязанная интеграция* реляционных СУБД и функций интеллектуального анализа данных, которая предполагает, что система интеллектуального анализа данных является одной из функциональных компонент СУБД. Это позволяет исполнять запросы интеллектуального анализа данных на основе имеющихся в СУБД механизмов индексирования данных, управления буферным пулом, оптимизации запросов и др.

На сегодня эффективная обработка и анализ сверхбольших хранилищ данных требуют использования параллельных СУБД, построенных на основе концепции фрагментного параллелизма и работающих на платформе высокопроизводительных вычислительных кластерных систем. Однако существующие сегодня коммерческие СУБД, использующие фрагментный параллелизм, имеют высокую стоимость и ориентированы, как правило, на специфические аппаратно-программные платформы. В то же время свободные СУБД надежны и предоставляют разработчикам открытый исходный код, что делает возможным построение параллельной СУБД на

основе свободной СУБД путем внедрения в код последней фрагментного параллелизма без масштабных изменений в исходных текстах. На сегодня, однако, отсутствуют свободные СУБД на базе фрагментного параллелизма, поскольку разработка такого сложного системного программного обеспечения, как параллельная СУБД, требует существенных финансовых и временных затрат.

В соответствии с этим *актуальной* является задача разработки методов внедрения фрагментного параллелизма в свободные реляционные СУБД с открытым кодом, позволяющие осуществить параллелизацию без масштабных изменений исходного кода.

Одной из современных тенденций развития процессорной техники являются многоядерные ускорители, которые обеспечивают от десятков до сотен процессорных ядер, поддерживающих векторную обработку данных. Для эффективной обработки и анализа данных соответствующие решения должны обеспечить полноценное и масштабируемое использование возможностей как многоядерных ускорителей, так и кластеров с узлами на базе таких вычислительных систем.

В соответствии с вышесказанным является *актуальной* задача разработки новых подходов и методов интеграции интеллектуального анализа в реляционные системы баз данных, а также разработка и реализация в рамках предлагаемых подходов новых параллельных алгоритмов интеллектуального анализа данных для кластерных вычислительных систем с узлами на базе современных многоядерных ускорителей.

Глава 2. Кластеризация и поиск шаблонов

В данной главе рассмотрены две задачи интеллектуального анализа данных: кластеризация и поиск шаблонов. В рамках первой задачи исследована проблематика использования реляционных СУБД для кластеризации больших объемов данных. Предложены алгоритмы кластеризации данных в параллельной СУБД на основе фрагментного параллелизма: алгоритм *dbParGraph* для кластеризации графа и алгоритм *pgFCM* для нечеткой кластеризации данных. В рамках второй задачи исследованы параллельные методы поиска шаблонов на многоядерных процессорах. Предложены следующие параллельные алгоритмы поиска частых наборов для многоядерных вычислительных систем: алгоритм *PDIC* для ускорителя Intel Xeon Phi и алгоритм *DDCapriori* для процессора IBM Cell BE. Представлены результаты вычислительных экспериментов, исследующих эффективность разработанных алгоритмов.

2.1. Алгоритм *dbParGraph* кластеризации графа в параллельной СУБД

Задача кластеризации вершин графа заключается в разбиении множества вершин графа на заданное количество непересекающихся частей примерно одинакового размера таким образом, чтобы суммарный вес ребер с концами в разных частях был минимальным. Предполагается, что количество вершин и количество ребер исходного графа таковы, что он не может быть целиком размещен в оперативной памяти.

Для решения данной задачи предлагается алгоритм *dbParGraph*, основанный на использовании параллельной СУБД. Граф представляется в виде реляционной таблицы (список ребер), которая подвергается горизон-

тальной фрагментации и распределяется по узлам кластерной вычислительной системы. Экземпляр параллельной СУБД запускается на каждом узле и обрабатывает собственный фрагмент таблицы.

2.1.1. Проектирование алгоритма

Вычислительная схема алгоритма

Предлагаемый алгоритм *dbParGraph* выполняет бисекцию графа (разбиение на две части) с помощью параллельной СУБД на основе фрагментного параллелизма (см. раздел 1.2.2). В алгоритме используется *реляционное представление графа* в виде списка ребер, где для каждого ребра графа хранятся номера (идентификаторы) концевых вершин и вес ребра. Реляционная таблица, представляющая граф, подвергается горизонтальной фрагментации. Каждый фрагмент обрабатывается независимо от других экземпляром параллельной СУБД, установленным на соответствующем вычислительном узле кластерной системы. Алгоритм *dbParGraph* состоит из двух этапов: поиск сообществ в графе и многоуровневое разбиение графа.

Целью **этапа поиска сообществ** является улучшение первоначальной фрагментации таблицы, представляющей список ребер графа. *Сообщество (community)* графа неформально определяют как подмножество вершин этого графа, в котором вершины плотно связаны между собой и редко связаны с другими частями графа [242] (в настоящее время отсутствует общепринятое формальное определение сообщества [251]). Данный этап реализуется в параллельной СУБД и состоит из трех стадий. На первой стадии выполняется *выявление сообществ*, количество которых в случае сверхбольшого графа, как правило, существенно превышает число вычислительных узлов кластерной системы. На второй стадии выполняется *консолидация (укрупнение) сообществ*, которая заключается в объединении некоторых пар сообществ в одно сообщество, выполняемом до тех пор, пока число сообществ не будет равно количеству вычислительных узлов

кластерной системы. На третьей стадии ребра графа, концевые вершины которых принадлежат одному и тому же сообществу, назначаются для обработки в один и тот же фрагмент базы данных. В силу того, что дальнейшее разбиение графа будет выполняться параллельной СУБД отдельно для каждого подграфа (фрагмента таблицы со списком ребер исходного графа) и вершины из одного сообщества с большей вероятностью попадут в один кластер, этап поиска сообществ позволит, таким образом, повысить качество разбиения.

На сегодня разработано большое количество алгоритмов выявления сообществ в графе (см., например, обзор [251]), в ряду которых *алгоритм распространения меток (Label Propagation Algorithm, LPA)* [199] является одним из первых. В указанном алгоритме вводятся меры сходства соседних вершин графа и вхождения вершины графа в сообщество, определяемые следующим образом. Пусть имеется вершина $v \in N$, обозначим множество соседних с ней вершин как $\mathcal{N}_v$. Тогда *сходство (affinity)* вершины v и соседней с ней вершины $u \in \mathcal{N}_v$ определяется функцией

$$affinity(v,u) = \frac{w(v,u)}{\sum_{i \in \mathcal{N}_v} w(v,i)}. \tag{2.1}$$

Вершина $v \in N$ имеет *метку* (уникальный числовой идентификатор) сообщества, которому она принадлежит, которую обозначим как $\mathcal{L}_v$. Тогда *степень вхождения вершины (degree of membership)* v в сообщество C определяется функцией

$$d(v,C) = \frac{\sum_{u \in \mathcal{N}_v \wedge \mathcal{L}_u = C} aff(v,u)}{\sum_{u \in \mathcal{N}_v} aff(v,u)}. \tag{2.2}$$

Метка каждой вершины графа инициализируется значением номера этой вершины графа. Затем итеративно производится следующая операция «распространения меток»: метка каждой вершины заменяется на метку, которую имеет соседняя вершина с наибольшей степенью вхождения в сообщество среди всех вершин, соседних с данной. Если таких меток несколько, выбирается метка с наименьшим идентификатором. Распространение

меток направлено на то, чтобы каждая вершина вошла в сообщество (подграф), в котором количество его внутренних ребер, инцидентных вершине, было бы минимальным. Процесс распространения меток завершается по достижении стабильного состояния вершин. Стабильное состояние считается достигнутым, если когда доля вершин графа, сохранивших свои метки, составляет не менее δ $(0 < \delta \leqslant 1)$, где число δ — наперед задаваемый параметр алгоритма.

При обработке сверхбольших графов наиболее вероятен сценарий, когда общее количество полученных сообществ превышает количество вычислительных узлов кластерной системы, на которых запускается алгоритм. Поэтому далее выполняется консолидация найденных сообществ посредством объединения двух сообществ в одно, если имеется хотя бы одно соединяющее их ребро (ребро с началом в одном сообществе и концом в другом сообществе). Для объединения выбираются два сообщества, которые имеют наименьшее среди остальных пар сообществ суммарное количество принадлежащих им и соединяющих их ребер. Процесс выполняется, пока количество сообществ не окажется равным количеству вычислительных узлов кластерной системы.

На финальной стадии этапа поиска сообществ выполняется распределение ребер графа по фрагментам. Ребро, концевые вершины которого принадлежат одному и тому же сообществу, полученному с помощью консолидации, назначается для обработки в один и тот же фрагмент базы данных (номер которого совпадает с номером сообщества). Ребро с концевыми вершинами из двух разных сообществ назначается для обработки в тот из двух фрагментов, соответствующих этим сообществам, в котором количество ребер меньше (для поддержания баланса фрагментов).

Этап **многоуровневого разбиения графа** *(multilevel graph partitioning)* [114] предполагает выполнение следующих трех стадий: огрубление, начальное разбиение и уточнение. *Стадия огрубления* выполняется в параллельной СУБД и обеспечивает редукцию исходного графа до размеров, позволяющих разместить граф и промежуточные данные в оперативной

памяти. На *стадии начального разбиения* огрубленный граф экспортируется из СУБД подается на вход сторонней системы, которая выполняет начальное разбиение графа в оперативной памяти (могут использоваться, например, свободно распространяемые утилиты Chaco [104] или METIS [113]). Результат начального разбиения сохраняется в базе данных в виде реляционной таблицы. На *стадии уточнения разбиения* параллельная СУБД с помощью запросов к базе данных формирует финальное разбиение графа в виде реляционной таблицы из двух столбцов: номер вершины графа и номер подграфа, которому принадлежит эта вершина.

Огрубление (coarsening) исходного графа G заключается в формировании последовательности графов $(G_0, G_1, \ldots, G_k)$, в которой граф G_{i+1} получается из графа G_i посредством стягивания ребер этого графа, образующих максимальное паросочетание. *Паросочетание (matching)* M графа G — это множество попарно несмежных ребер в G. *Максимальное паросочетание (maximal matching)* — это такое паросочетание M в графе G, которое не содержится ни в каком другом паросочетании этого графа.

В результате стягивания ребер вершина $v \in G_i$ преобразуется в вершину $v' \in G_{i+1}$, которая называется *образом* вершины v, что обозначается как $v' = Im(v)$. Каждая вершина в последовательности графов $G_0, G_1, \ldots, G_k$ имеет в точности один образ.

Посредством увеличения числа k количество вершин в графе G_k сокращается до сколь угодно малого значения, при котором граф G_k и промежуточные данные могут быть целиком размещены в оперативной памяти. После этого над графом G_k выполняется *начальное разбиение (initial partitioning)*. Для выполнения начального разбиения могут использоваться различные алгоритмы для разбиения графа в оперативной памяти (например, алгоритм Кернигана—Лина [118], спектральный алгоритм [198] и др.).

На стадии *уточнения (uncoarsening)* разбиения огрубленных графов преобразуются в разбиения этих графов до огрубления. Для выполнения разбиения графа G_i при наличии разбиения графа G_{i+1} достаточно разбить вершины в соответствии с разбиением их образов в G_{i+1}. Если множество

вершин N_i является разбиением графа G_{i+1}, то соответствующее разбиение графа G_i будет состоять из частей вида $N_i^{'} = \{v \mid Im(v) \in N_i\}$. После получения разбиения графа G_i оно улучшается с помощью используемого алгоритма локального улучшения разбиений. Эти действия повторяются до тех пор, пока не будет получено разбиение графа $G_0 \equiv G$.

Схема базы данных алгоритма

Табл. 2.1. Схема базы данных алгоритма *dbPargraph*

№ п/п	Таблица	Семантика	Поля таблицы
1	G	Исходный граф	A, B: номера концевых вершин ребра W: вес ребра
2	$VERTEX$	Сведения о принадлежности вершин сообществам	A: номер вершины C: номер сообщества данной вершины
3	$GRAPH$	Исходный граф с улучшенной фрагментацией ребер	A, B: номера концевых вершин ребра W: вес ребра F: номер фрагмента
4	$MATCH$	Максимальное паросочетание исходного графа	A, B: номера концевых вершин ребра F: номер фрагмента
5	$COARSE\_GRAPH$	Огрубленный граф	A, B: номера концевых вершин ребра W: вес ребра F: номер фрагмента
6	$COARSE\_PARTITIONS$	Начальное разбиение огрубленного графа	A: номер вершины, P: цвет вершины
7	$PARTITIONS$	Разбиение исходного графа	A: номер вершины, P: цвет вершины, G: значение функции выгоды

В табл. 2.1 представлены основные таблицы реляционной схема базы данных, спроектированной для решения задачи разбиения графа. Каждая таблица базы данных подвергается горизонтальной фрагментации. На

каждом узле кластера набор фрагментов таблиц обрабатывается отдельно соответствующим экземпляром параллельной СУБД.

На стадии поиска сообществ фрагментация таблиц G, $VERTEX$ осуществляется с помощью функции фрагментации $\varphi(t) = t.A \; mod \; F$, где F — общее количество вычислительных узлов кластерной системы. Результатом стадии является таблица $GRAPH$ с улучшенной фрагментацией исходного графа, где поле F указывает номер фрагмента для хранения данного ребра.

На стадиях огрубления и начального разбиения таблицы $MATCH$ и $COARSE\_GRAPH$, хранящие ребра графа, в качестве атрибута фрагментации используют поле F и функцию фрагментации $\varphi(t) \equiv t.F$. Остальные таблицы используют функцию фрагментации $\varphi(t) = t.A \; mod \; F$.

2.1.2. Реализация алгоритма

Выявление сообществ в графе

```
1  -- Полный список ребер исходного графа
2  INSERT INTO tmp_AFF_SUBG
3     (SELECT A, B, W FROM G) UNION (SELECT B, A, W FROM G);
4
5  -- Суммарный вес смежных ребер каждой концевой вершины исходного графа
6  INSERT INTO tmp_AFF_WNBR
7     SELECT A, sum(W) as WNBR FROM tmp_AFF_SUBG GROUP BY A;
8
9  -- Схожесть пар вершин исходного графа
10 INSERT INTO tmp_AFFINITY
11    SELECT x.A, B, W/WNBR as AFTY
12    FROM tmp_AFF_SUBG as x, tmp_AFF_WNBR as y
13    WHERE x.A = y.A;
14
15 -- Суммарная схожесть соседних вершин каждой вершины исходного графа
16 INSERT INTO tmp_COMM_AFNBRALL
17    SELECT A, sum(AFTY) as AFNBRALL FROM tmp_AFFINITY GROUP BY A;
```

Рис. 2.1. Реализация вспомогательных таблиц для выявления сообществ

Поиск сообществ в графе начинается с создания и заполнения таблиц, предназначенных для хранения промежуточных данных алгоритма. За-

просы SQL, реализующие заполнение вспомогательных таблиц данными, представлены на рис. 2.1.

```
1  -- Инициализация таблицы с данными о принадлежности вершин сообществам
2  INSERT INTO VERTEX
3    SELECT A, A as C
4    FROM (SELECT A FROM G UNION SELECT B FROM G) as nodes;
5
6  s ← 0;
7  -- Распространять метки до достижения стабильности
8  while s < ⌈δ · |N|⌉ do
9  -- Создание вспомогательной таблицы
10   INSERT INTO tmp_COMM_AFNBRCOM
11     SELECT tmp_AFFINITY.A, VERTEX.C, sum(AFTY) as AFNBRCOM
12     FROM tmp_AFFINITY, VERTEX
13     WHERE tmp_AFFINITY.B = VERTEX.A
14     GROUP BY tmp_AFFINITY.A, VERTEX.C;
15
16 -- Заполнение таблицы со сведениями о сообществах
17   INSERT INTO tmp_COMMUNITY
18     SELECT x.A, C, AFNBRCOM/AFNBRALL as D
19     FROM tmp_COMM_AFNBRALL as x, tmp_COMM_AFNBRCOM as y
20     WHERE X.A = Y.A;
21
22 -- Вычисление max D
23   INSERT INTO tmp_COMM_DMAX
24     SELECT A, max(D) AS D FROM tmp_COMMUNITY GROUP BY A;
25
26 -- Распространение меток
27   INSERT INTO tmp_VER_COMM
28     SELECT A, min(C)
29     FROM (SELECT x.A, x.C
30       FROM tmp_COMMUNITY as x, tmp_COMM_DMAX as y
31       WHERE x.A = y.A AND x.D = y.DMAX)
32     GROUP BY A;
33
34 -- Вычисление количества вершин, сохраняющих метки
35   SELECT count(*) INTO s
36   FROM VERTEX, tmp_VER_COMM
37   WHERE VERTEX.A = tmp_VER_COMM.A AND VERTEX.C = tmp_VER_COMM.C;
38
39 -- Обновление меток вершин
40   UPDATE VERTEX
41     SET C = (SELECT C
42       FROM tmp_VER_COMM WHERE tmp_VER_COMM.A = VERTEX.A);
43
44 -- Очистка вспомогательных таблиц
45   TRUNCATE tmp_COMMUNITY, tmp_COMM_AFNBRCOM, tmp_COMM_DMAX, tmp_VER_COMM;
46 end while
```

Рис. 2.2. Реализация стадии распространения меток сообществ

Запросы SQL, реализующие распространение меток сообществ в графе, представлены на рис. 2.2.

Консолидация сообществ в графе

Консолидация сообществ в графе начинается с создания и заполнения таблиц, предназначенных для хранения промежуточных данных алгоритма. Запросы SQL, реализующие заполнение вспомогательных таблиц данными, представлены на рис. 2.3.

```
-- Список соответствия исходных и укрупненных сообществ
INSERT INTO tmp_COMM_MATCH
  SELECT row_number() as C_new, C FROM VERTEX GROUP BY C;

-- Список ребер с указанием сообществ их концевых вершин
INSERT INTO tmp_COMM_EDGE
  SELECT A, B, c1 as C_A, VERTEX.C as C_B
  FROM (
    SELECT A, B, VERTEX.C as c1
    FROM G, VERTEX
    WHERE G.A = VERTEX.A) as comm, VERTEX
  WHERE VERTEX.A = comm.B;

-- Количество ребер в сообществах (для балансировки ребер по фрагментам)
INSERT INTO tmp_COMM_CAPACITY
  SELECT C_new, 0 as E FROM tmp_COMM_MATCH;
UPDATE tmp_COMM_CAPACITY
  SET E = (SELECT cnt
    FROM (
      SELECT C_A, count(*) as cnt
      FROM tmp_COMM_EDGE WHERE C_A = C_B) as comm
    WHERE comm.C_A = tmp_COMM_EDGE.C);

-- Пары связанных сообществ с указанием количества соединяющих ребер
-- (для балансировки ребер по фрагментам)
INSERT INTO tmp_COMM_CUT
  SELECT C_A as C1, C_B as C2, count(*) as CUT
  FROM tmp_COMM_EDGE
  WHERE C_A ≠ C_B
  GROUP BY C_A, C_B ORDER BY CUT;
```

Рис. 2.3. Реализация вспомогательных таблиц для консолидации сообществ

Запросы SQL, реализующие консолидацию сообществ графа, представлены на рис. 2.4.

```
-- Количество сообществ перед консолидацией
comm_count ← SELECT count(distinct C) FROM VERTEX;
last_comm ← comm_count; -- Идентификатор укрупняемого сообщества

-- Укрупнять сообщества, пока их больше, чем вычислительных узлов кластера
while comm_count > F do
   comm_count ← comm_count − 1; last_comm ← last_comm + 1;

-- Нахождение пары объединяемых сообществ (c1, c2)
   SELECT c1, c2, (CUT+E+edge) as total
   FROM (
      SELECT C1, C2, CUT, E as edge
      FROM tmp_COMM_CAPACITY, tmp_COMM_CUT
      WHERE C1 = C) as comm, tmp_COMM_CAPACITY
   WHERE C2 = C
   ORDER BY total, c1, c2
   LIMIT 1;

-- Изменение меток вершин, входящих в укрупненные сообщества
   UPDATE VERTEX
      SET C = last_comm WHERE C = c1 OR C = c2;
   UPDATE tmp_COMM_EDGE
      SET C_A = last_comm WHERE A in (SELECT A FROM VERTEX WHERE C = last_comm),
      SET C_B = last_comm WHERE B in (SELECT A FROM VERTEX WHERE C = last_comm);

-- Очистка и обновление вспомогательных таблиц
   TRUNCATE tmp_COMM_CAPACITY, tmp_COMM_CUT;
   INSERT INTO tmp_COMM_CAPACITY
      SELECT distinct C, 0 FROM VERTEX;
-- пересчет количества ребер в сообществах
   UPDATE tmp_COMM_CAPACITY
      SET E = (
         SELECT cnt
         FROM (
            SELECT C_A, count(*)
            FROM tmp_COMM_EDGE WHERE C_A = C_B GROUP BY C_A) as comm
         WHERE comm.C_A = tmp_COMM_EDGE.C);
-- актуализация сведений о парах сообществ
   comm ← SELECT C_A, C_B, count (*) as cnt
      FROM tmp_COMM_EDGE WHERE C_A ≠ C_B GROUP BY C_A, C_B;
   INSERT INTO tmp_COMM_CUT
      SELECT C_A, C_B, sum(cnt) FROM comm GROUP BY C_A, C_B;
end while
```

Рис. 2.4. Реализация стадии консолидации сообществ

Улучшение фрагментации графа

На данной стадии создается таблица $GRAPH$, в которой улучшено распределение ребер графа по фрагментам по сравнению с исходной таблицей G (см. табл. 2.1). Ребро, концевые вершины которого принадлежат одному

```
for edge in (SELECT * FROM tmp_COMM_EDGE) loop
  if (edge.C_A = edge.C_B) then
    INSERT INTO GRAPH
      VALUES (edge.A, edge.B,
        (SELECT W FROM G WHERE G.A = edge.A AND G.B = edge.B),
        edge.C_A as F);
  else
    e1 ← SELECT E FROM tmp_COMM_CAPACITY WHERE edge.C_A = C;
    e2 ← SELECT E FROM tmp_COMM_CAPACITY WHERE edge.C_B = C;
    if (e1 ⩽ e2) then
      INSERT INTO GRAPH
        VALUES (edge.A, edge.B,
          (SELECT W FROM G WHERE G.A = edge.A AND G.B = edge.B), edge.C_A as
            F);
      UPDATE tmp_COMM_CAPACITY SET E = E+1 WHERE C = edge.C_A;
    else
      INSERT INTO GRAPH
        VALUES (edge.A, edge.B,
          (SELECT W FROM G WHERE G.A = edge.A AND G.B = edge.B),
          edge.C_A as F);
        UPDATE tmp_COMM_CAPACITY SET E = E+1 WHERE C = edge.C_B;
    end if;
  end if;
end loop;
```

Рис. 2.5. Реализация стадии улучшения фрагментации

и тому же сообществу, назначается для обработки в один и тот же фрагмент базы данных. Для балансировки размеров фрагментов каждое ребро с концевыми вершинами из разных сообществ назначается для обработки в тот из соответствующих фрагментов, где количество ребер меньше. Реализация улучшения фрагментации графа представлена на рис. 2.5.

Огрубление графа

Огрубление графа выполняется на основе эвристики, предложенной в работе [114], и состоит из двух последовательно выполняемых шагов: поиск и стягивание. Пример работы алгоритма *dbParGraph* на стадии огрубления графа приведен на рис. 2.6.

На шаге *поиска* осуществляется нахождение максимального паросочетания исходного графа, которое имеет вес, близкий к максимальному. Запрос SQL, реализующие шаг поиска, представлен на рис. 2.7.

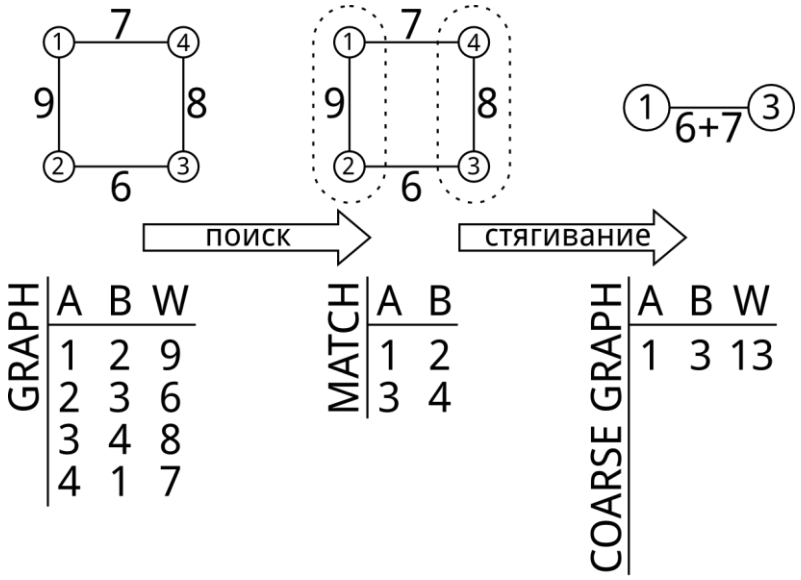


Рис. 2.6. Пример работы алгоритма на стадии огрубления графа

```
for edge in (
  SELECT A, B
  FROM GRAPH
  ORDER BY W DESC)
loop
  if not exists(
    SELECT *
    FROM visited
    WHERE A=edge.A OR A=edge.B) then
      INSERT INTO visited VALUES (edge.A);
      INSERT INTO visited VALUES (edge.B);
      INSERT INTO MATCH VALUES (edge.A, edge.B);
  end if;
end loop;
```

Рис. 2.7. Реализация поиска паросочетания

На шаге *стягивания* осуществляется удаление ребер, найденных на шаге поиска. Концы удаляемого ребра заменяются одной вершиной, петли удаляются. Кратные ребра преобразуются в одно ребро, вес которого равен сумме весов кратных ребер. Запрос SQL, реализующий шаг стягивания, представлен на рис. 2.8.

В оригинальном алгоритме [114] при огрублении *многократно* выполняется поиск и стягивание ребра с наибольшим весом. Заметим, что поскольку в оригинальном алгоритме при поиске игнорируются ребра, концевые вершины которых являются результатом ранее выполненных операций стя-

```
SELECT
   least(newA, newB) as A,
   greatest(newA, newB) as B,
   sum(W) as W
FROM (
   SELECT
      coalesce(match2.A, GRAPH.A) as newA,
      coalesce(MATCH.A, GRAPH.B) as newB,
      GRAPH.W
   FROM
      GRAPH left join MATCH on GRAPH.B = MATCH.B
      left join MATCH as match2 on GRAPH.A = match2.B)
WHERE newA <> newB
GROUP BY A, B;
```

Рис. 2.8. Реализация шага стягивания

гивания, то найденные данным алгоритмом ребра являются паросочетанием. Таким образом, предложенный алгоритм эквивалентен оригинальному. При этом использование СУБД позволяет выполнить шаг стягивания как один запрос на удаление записей из таблицы.

Огрубление повторяется многократно, чтобы редуцировать количество ребер исходного граф до приемлемого значения, при котором граф может быть целиком размещен в оперативной памяти. Огрубленный граф экспортируется из базы данных и является исходными данными следующей стадии начального разбиения.

Уточнение разбиения графа

Стадия уточнения разбиения графа состоит из трех последовательно выполняемых шагов [118]: окрашивание, оценка качества разбиения и улучшение разбиения. Пример работы алгоритма *dbParGraph* на стадии уточнения разбиения графа приведен на рис. 2.9.

На шаге *окрашивания* концам стянутых ребер исходного графа присваивается цвет соответствующей вершины огрубленного графа. Запрос SQL, реализующий шаг окрашивания, представлен на рис. 2.10.

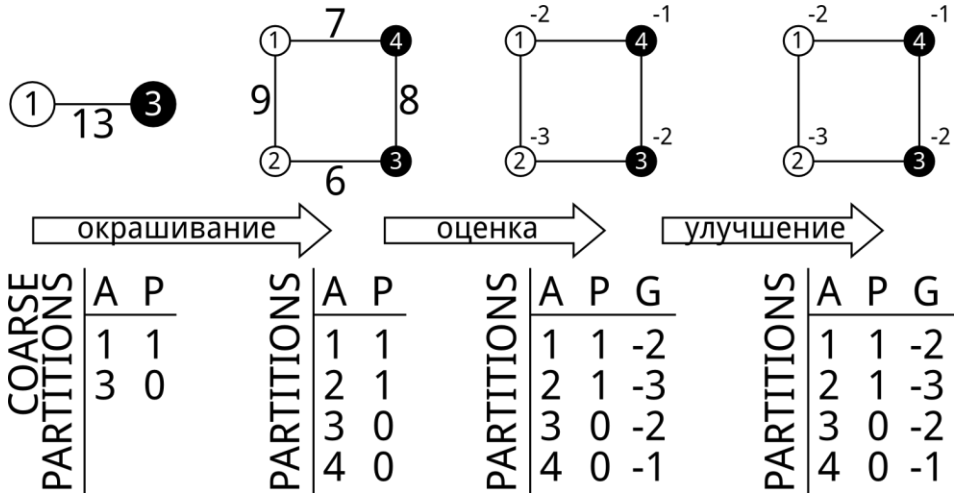


Рис. 2.9. Пример работы алгоритма на стадии уточнения разбиения графа

```
1  INSERT INTO PARTITIONS
2  SELECT A, P
3  FROM COARSE_PARTITIONS
4  UNION
5  SELECT MATCH.B, COARSE_PARTITIONS.P
6  FROM MATCH, COARSE_PARTITIONS
7  WHERE MATCH.A = COARSE_PARTITIONS.A;
```

Рис. 2.10. Реализация шага окрашивания

На шаге *оценки качества разбиения* для каждой вершины графа, полученного на предыдущем шаге, вычисляется *функция выгоды (gain)* [118]:

$$\begin{aligned} \mathrm{g}(v) &= \mathrm{ext}(v) - \mathrm{int}(v), \\ \mathrm{ext}(v) &= \sum_{(v,u)\in E \wedge P(v)\neq P(u)} w(v,u), \\ \mathrm{int}(v) &= \sum_{(v,u)\in E \wedge P(v)=P(u)} w(v,u) \end{aligned} \tag{2.3}$$

В формулах (2.3) функция $P : E \to \{1, \ldots, p\}$ возвращает номер части разбиения графа, к которой относится заданная вершина. Значение функции $\mathrm{g}(v)$ показывает выгоду от инверсии цвета вершины v: если $\mathrm{g}(v) > 0$, то цвет вершины необходимо изменить на противоположный. Оценка качества разбиения реализуется с помощью запроса, представленного на рис. 2.11.

На шаге *улучшения разбиения* осуществляются поиск вершин с макси-

```
SELECT
   PARTITIONS.A,
   PARTITIONS.P,
   SUM(subgains.Gain) as Gain
FROM
   PARTITIONS left join (
      SELECT
         GRAPH.A, GRAPH.B,
         case when ap.P = bp.P then
            -GRAPH.W
            else
            GRAPH.W
         end as Gain
      FROM
         GRAPH left join PARTITIONS as ap on GRAPH.a = ap.A
            left join PARTITIONS as bp on GRAPH.b = bp.A
   ) as subgains
   on PARTITIONS.A = subgains.A or PARTITIONS.A = subgains.B
GROUP BY PARTITIONS.A, PARTITIONS.P
```

Рис. 2.11. Подсчет функции выгоды

мальным значением функции выгоды и инвертирование ее цвета (до тех пор, пока такие вершины существуют). Реализация данных операций выполняется с помощью запросов, представленных на рис. 2.12.

Таким образом, результатом шага улучшения разбиения является таблица $PARTITIONS$, поля которой представляют номер и цвет соответствующей вершины исходного графа.

2.1.3. Вычислительные эксперименты

Для исследования эффективности разработанного алгоритма были проведены вычислительные эксперименты с параллельной СУБД PargreSQL [15] (рассматривается в главе 5) на платформе суперкомпьютера «Торнадо ЮУрГУ» [6]. Целью экспериментов является сравнение производительности и качества разбиения алгоритма *dbPargraph* со следующими параллельными алгоритмами разбиения графа, рассмотренными ранее в разделе 1.3.2.

Параллельный алгоритм *MSP* предложен Зенгом (Zeng) и др. в работе [256], в которой приведены результаты экспериментов, исследующих эффективность данного алгоритма на кластерной системе из одного мастер-

```
SELECT *
FROM PARTITIONS
WHERE
    P = current AND
    G = (
        SELECT max(G)
        FROM PARTITIONS
        WHERE P = current)
LIMIT 1
INTO V
```

(a) Поиск вершины

```
UPDATE PARTITIONS
   SET G = G + W *
      (case when P = V.P then
          2
         else
        -2
       end)
FROM (
   SELECT
      case when A = V.A then
        B
       else
        A
      end,
      W
   FROM GRAPH
   WHERE B = V.A OR A = V.A)
   as neighbors
WHERE neighbors.A = PARTITIONS.A;

UPDATE PARTITIONS
   SET G = -G, P = 1-P
WHERE A=V.A;
```

(b) Инвертирование цвета вершины

Рис. 2.12. Реализация шага улучшения разбиения

узла и 32 рабочих узлов. В качестве входных данных использовался синтетический граф Ваттса—Строгаца [243] (10 млн. вершин, 50 млн. ребер).

Система распределенного разбиения графов *ParMETIS* [113], разработанная Кариписом (Karypis) и др., и свободно распространяемая на уровне исходных кодов, исследована на синтетическом графе R-MAT [53] (7.7 млн. вершин, 133 млн. ребер).

Система распределенного разбиения графов *PT-Scotch*, разработанная Пеллегрини (Pellegrini) и др., и свободно распространяемая на уровне исходных кодов, в работе [63] исследована на кластерной системе из четырех SMP узлов. В качестве входных данных использовался граф с 23 млн. вершин, 175 млн. ребер.

В экспериментах мерой качества Q разбиения графа $G = (N, E)$ на подграфы $N_1, \ldots, N_p$, $N_i \subseteq N$ выступает доля ребер исходного графа, входящих в разрез (см. раздел 1.1.1; меньшим значениям соответствует

лучшее качество разбиения):

$$Q(N_1, \ldots, N_p) = \frac{|E_{cut}|}{|E|}.$$

При сравнении алгоритм *dbPargraph* запускался на тех же графах и конфигурациях кластера «Торнадо ЮУрГУ» с примерно равной пиковой производительностью, что и у алгоритмов-конкурентов. Данные о быстродействии и качестве конкурентов взяты из соответствующих статей.

Для алгоритмов-аналогов, предполагающих работу вне СУБД, измерялось время накладных расходов на экспорт исходных данных из СУБД и импорт результатов в СУБД. Экспорт исходных данных включает в себя сохранение списка ребер исходного графа из СУБД в текстовый формат CSV (Comma-Separated Values) и преобразование данных из формата CSV в формат CSR (Compressed Sparse Row), представляющий собой один из наиболее популярных форматов для обработки сверхбольших графов [209]. Импорт результатов включает в себя время на преобразование полученного результата в формат CSV и сохранение его в базе данных.

Табл. 2.2. Сравнение производительности алгоритма *dbPargraph* с аналогами

Граф		Аналог						***dbPargraph***	
		Алгоритм	Платформа	Время, с					
$\|N\|$	$\|E\|$			Разб-е	Экспорт	Импорт	Всего	Время, с	Платформа
10^7	$5 \cdot 10^7$	*MSP* [256]	1 master, 32 slave 2×CPU 3.2 ГГц	962	307	36	1 305	**1 189**	32 CPU 3.33 ГГц
$7.7 \cdot 10^6$	$1.33 \cdot 10^8$	*ParMETIS* [113]	64 2×CPU 3.33 ГГц	500	474	30	1 004	**886**	64 CPU 3.33 ГГц
$2.3 \cdot 10^7$	$1.75 \cdot 10^8$	*PT-Scotch* [63]	4 8×CPU 1.5 ГГц	417	652	79	1 148	**897**	2 CPU 3.33 ГГц

Результаты сравнения производительности и качества разбиения алгоритмов приведены в табл. 2.2 и табл. 2.3 соответственно.

Табл. 2.3. Сравнение качества разбиения алгоритма *dbPargraph* с аналогами

Алгоритм	Граф		Качество разбиения, %	
	$\|N\|$	$\|E\|$	Аналог	***dbPargraph***
MSP [256]	10^7	$5 \cdot 10^7$	7	9.8
ParMETIS [113]	$7.7 \cdot 10^6$	$1.33 \cdot 10^8$	4	6.3

Можно видеть, что разработанный алгоритм показывает худшее, чем у конкурентов, быстродействие, и сравнимое качество кластеризации при

разбиении графа. Однако, в отличии от аналогов, *dbPargraph* работает в предположении, что исходный граф непосредственно хранится в базе данных и выполняет разбиение графа, не выходя за рамки СУБД. При использовании алгоритмов-аналогов, напротив, перед их запуском необходимо выполнять экспорт анализируемых данных из базы данных, а после запуска — импорт результатов кластеризации в базу данных. С учетом указанных накладных расходов на экспорт-импорт разработанный алгоритм *dbPargraph* опережает аналоги по производительности.

2.2. Алгоритм *pgFCM* нечеткой кластеризации данных в параллельной СУБД

Нечеткая кластеризация данных в СУБД выполняется посредством интеграции алгоритма *Fuzzy c-Means (FCM)* [43] в параллельную СУБД в соответствии со следующим сценарием. Исходные данные, результаты промежуточных вычислений и выходные данные алгоритма представляются в виде реляционных таблиц, входящих в базу данных со специально разработанной схемой. Вычисления реализуются как запросы SQL к указанным таблицам. Таблицы базы данных подвергаются горизонтальной фрагментации и распределяются по узлам кластерной вычислительной системы. Экземпляр параллельной СУБД запускается на каждом узле и обрабатывает собственный фрагмент таблицы.

2.2.1. Проектирование алгоритма

Нотация

В дальнейшем описании используются следующие обозначения [44]:

— $d \in \mathbb{N}$ — размерность пространства кластеризуемых объектов;

— $\ell \in \mathbb{N} : 1 \leqslant \ell \leqslant d$ — номер координаты кластеризуемого объекта;

— $n \in \mathbb{N}$ — количество кластеризуемых объектов;

— $X \subset \mathbb{R}^d$ — множество кластеризуемых объектов;

— $i \in \mathbb{N} : 1 \leqslant i \leqslant n$ — номер объекта;

— $x_i \in X$ — i-й объект;

— $k \in \mathbb{N}$ — количество кластеров;

— $j \in \mathbb{N} : 1 \leqslant j \leqslant k$ — номер кластера;

— $C \subset \mathbb{R}^{k \times d}$ — матрица, содержащая центры кластеров (*матрица центроидов*);

— $c_j \in \mathbb{R}^d$ — центр кластера j;

— $x_{i\ell}, c_{j\ell} \in \mathbb{R}$ — ℓ-е координаты объектов x_i и c_j соответственно;

— $U \subset \mathbb{R}^{n \times k}$ — матрица степеней принадлежности, где число $u_{ij} \in \mathbb{R}$ показывает степень принадлежности объекта x_i кластеру j и выполняются следующие свойства:

$$\forall\, i, j \quad u_{ij} \in [0; 1], \quad \forall\, i \quad \sum_{j=1}^{k} u_{ij} = 1 \tag{2.4}$$

— $\rho : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}_+ \cup 0$ — функция расстояния, используемая для определения близости объекта x_i и центроида c_j (параметр алгоритма);

— $m \in \mathbb{R} : m > 1$ — степень нечеткости целевой функции (параметр алгоритма);

— J_{FCM} — целевая функция.

Алгоритм *FCM* выполняет минимизацию *целевой функции* J_{FCM}, которая имеет следующий вид:

$$J_{FCM}(X, k, m) = \sum_{i=1}^{n} \sum_{j=1}^{k} u_{ij}^{m} \rho^2(x_i, c_j). \tag{2.5}$$

Нечеткое разбиение исходного множества объектов достигается посредством минимизации целевой функции (2.5). На каждой итерации происходит обновление матрицы центроидов C и матрицы степеней принадлежности U по следующим формулам:

$$\forall j, \ell \quad c_{j\ell} = \frac{\sum\limits_{i=1}^{n} u_{ij}^{m} \cdot x_{i\ell}}{\sum\limits_{i=1}^{n} u_{ij}^{m}} \tag{2.6}$$

$$u_{ij} = \sum_{t=1}^{k} \left(\frac{\rho(x_i, c_j)}{\rho(x_i, c_t)} \right)^{\frac{2}{1-m}} \tag{2.7}$$

Пусть s — номер итерации, $u_{ij}^{(s)}$ и $u_{ij}^{(s+1)}$ — элементы матрицы U на s-м и $(s{+}1)$-м шагах соответственно, $\varepsilon \in (0, 1) \subset \mathbb{R}$ — пороговое значение останова алгоритма. Тогда критерий останова вычислений имеет следующий вид:

$$\max_{ij} \{ |u_{ij}^{(s+1)} - u_{ij}^{(s)}| \} \leqslant \varepsilon \tag{2.8}$$

Входными данными алгоритма *FCM* (см. алг. 2.1) являются множество объектов $X = (x_1, x_2, \ldots, x_n)$, количество кластеров k, степень нечеткости m и пороговое значение останова ε. Алгоритм возвращает матрицу степеней принадлежности U.

Алг. 2.1. FUZZY c-MEANS(IN X, m, ε, k; OUT U)

1: Инициализировать матрицу $U^{(0)}$ случайными числами
2: $s \leftarrow 0$
3: **repeat**
4: Вычислить матрицу $C^{(s)}$, используя формулу 2.6
5: Вычислить матрицы $U^{(s)}$ и $U^{(s+1)}$, используя формулу 2.7
6: $s \leftarrow s + 1$
7: **until** условие 2.8 не выполняется
8: **return** U

Схема базы данных алгоритма

Для идентификации записей в реляционных таблицах используются уникальные номера, введенные в табл. 2.4 (где n, k и d определены ранее.

Табл. 2.4. Нумерация элементов данных в реляционных таблицах

Номер	Область значений	Семантика
i	$1 \leqslant i \leqslant n$	Номер вектора данных
j	$1 \leqslant j \leqslant k$	Номер кластера
ℓ	$1 \leqslant \ell \leqslant d$	Номер координаты вектора
s	$0 \leqslant s < +\infty$	Номер итерации алгоритма

Для вычисления критерия останова алгоритма (2.8) определим функцию $\delta(s, s+1)$ следующим образом:

$$\delta(s, s+1) = \max_{ij} \{|u_{ij}^{(s+1)} - u_{ij}^{(s)}|\} \tag{2.9}$$

В табл. 2.5 представлена схема базы данных, используемой для интеграции алгоритма нечеткой кластеризации данных в СУБД. Атрибуты, входящие в первичные ключи таблиц, подчеркнуты; первичные ключи представляют собой натуральные числа (определенные выше в табл. 2.4). Неключевые атрибуты таблиц имеют вещественный тип.

Для хранения объектов множества X в базе данных определяется таблица SH, каждая запись которой хранит объект данных размерности d

Табл. 2.5. Схема базы данных алгоритма *pgFCM*

№ п/п	Таблица	Поля таблицы	Кол-во записей	Семантика
1	SH	$\underline{i}, x1, x2, \ldots, xd$	n	Множество объектов
2	SV	$\underline{i, \ell}, val$	$n \cdot d$	Множество объектов «вертикальное»
3	C	$\underline{j, \ell}, val$	$k{\cdot}d$	Координаты центроидов
4	SD	$\underline{i, j}, dist$	$n{\cdot}k$	Расстояния между объектами x_i и центроидами c_j
5	U	$\underline{i, j}, val$	$n{\cdot}k$	Степени принадлежности объектов x_i кластерам j на шаге s
6	UT	$\underline{i, j}, val$	$n{\cdot}k$	Степени принадлежности объектов x_i кластерам j на шаге $s+1$
7	P	$d, k, n, \underline{s}, delta$	s	Значение функции (2.9) на текущей итерации

с номером i. Таблица $SH(\underline{i}, x1, x2, \ldots, xd)$ имеет n записей и первичный ключ i.

В ходе выполнения вычислений, предусмотренных алгоритмом *FCM*, требуется выполнять агрегирование координат векторов множества X (подсчет суммы, максимума и др.). Однако, к таблице SH не могут быть применены соответствующие агрегатные вычислительные функции SQL, поскольку указанные функции предназначены для агрегации значений по строкам, а не по столбцам реляционной таблицы.

В соответствии с этим в базу данных вводится таблица $SV(\underline{i, \ell}, val)$, состоящая из $n{\cdot}d$ записей и имеющая составной первичный ключ (i, ℓ), где i указывает номер исходного объекта, ℓ — номер координаты этого объекта. Таблица SV представляет собой выборку данных из таблицы SH и обеспечивает хранение координат исходных объектов в «вертикальном» виде (в одном столбце), что позволяет применить строчные агрегатные вычислительные функции SQL ($max()$, $count()$, $sum()$ и др.).

Для хранения данных о координатах центроидов кластеров в базе дан-

ных создается таблица $C(\underline{j, \ell}, val)$, которая имеет $k{\cdot}d$ записей и составной первичный ключ (j, ℓ). Как и в случае таблицы SV, структура таблицы C позволяет применить в вычислениях агрегатные функции SQL.

Алгоритм *FCM* предполагает определение степени принадлежности объекта с номером i кластеру с номером j, что включает в себя вычисление расстояний $\rho(x_i, c_j)$. Для хранения расстояний в базе данных создается таблица $SD(\underline{i, j}, dist)$ с количеством записей $n{\cdot}k$ и составным первичным ключом (i, j).

Таблица $U(\underline{i, j}, val)$ предназначена для хранения степеней принадлежности, полученных на шаге s. Для хранения степеней принадлежности на шаге $s{+}1$ дополнительно требуется таблица $UT(\underline{i, j}, val)$ с аналогичной структурой. Каждая из данных таблицы содержит $n{\cdot}k$ записей и имеет составной первичный ключ (i, j).

Таблица $P(d, k, n, \underline{s}, delta)$ хранит номер итерации s и значение формулы (2.9) для этого номера. Количество записей в таблице зависит от числа итераций, которые понадобились для завершения алгоритма.

Каждая таблица базы данных подвергается горизонтальной фрагментации. На каждом узле кластера набор фрагментов таблиц обрабатывается отдельно соответствующим экземпляром параллельной СУБД. Фрагментация осуществляется с помощью следующей функции, которая обеспечивает примерно равные размеры фрагментов таблиц на всех узлах кластерной вычислительной системы:

$$\varphi(t) = t.i \ mod \ F, \tag{2.10}$$

где F — количество вычислительных узлов в кластерной системе.

Интерфейс алгоритма

Алгоритм нечеткой кластеризации данных, интегрируемый в параллельную СУБД, получил название *pgFCM* (см. алг. 2.2). *pgFCM* оформляется в виде хранимой процедуры на языке *PL/pgSQL*. Исходное множество

Алг. 2.2. pgFCM(IN таблица SH, m, ε, k; OUT таблица U)

1: Создать и инициализировать таблицы U, P, SV и др.
2: **repeat**
3: Вычислить координаты центроидов c_j. Обновить таблицу C
4: Вычислить расстояния $\rho(x_i, c_j)$. Обновить таблицу SD
5: Вычислить степени принадлежности ut_{ij}. Обновить таблицу UT
6: Обновить таблицы U, P
7: **until** $P.delta > \varepsilon$
8: **return** U

объектов данных X хранится в таблице SH. Степень нечеткости m, пороговое значение останова ε и количество кластеров k являются входными параметрами. Результат работы алгоритма сохраняется в таблице U.

2.2.2. Реализация алгоритма

Создание и инициализация таблиц

```
1  CREATE TEMP TABLE U (
2    i int, j int, val numeric,
3    PRIMARY KEY (i,j));
4
5  CREATE TEMP TABLE P (
6    d int, k int, n int, s int, delta numeric,
7    PRIMARY KEY (s));
8
9  CREATE TEMP TABLE SV (
10   i int, l int, val numeric,
11   PRIMARY KEY (i,l));
```

Рис. 2.13. Создание таблиц SV, U и P алгоритма *pgFCM*

Команды SQL, обеспечивающие создание таблиц SV, U и P, приведены на рис. 2.13. В данном случае для создания таблиц используется ключевое слово `TEMP`, которое указывает СУБД, что создается специальная временная таблица. Временные таблицы создаются в отдельном табличном пространстве и уничтожаются по завершении работы алгоритма.

Перед выполнением вычислительной части алгоритма необходимо инициализировать таблицы SV, U и P. Команды SQL, обеспечивающие иници-

```
-- Инициализация таблицы SV
INSERT INTO SV
  SELECT SH.i, 1, x1
  FROM SH;
...
INSERT INTO SV
  SELECT SH.i, d, xd
  FROM SH;

-- Инициализация таблицы P
INSERT INTO P(d, k, n, s, delta)
  VALUES (d, k, n, 0, 0.0);

-- Инициализация таблицы U случайными числами
INSERT INTO U (i, j, val)
  VALUES (1, 1, random());
...
INSERT INTO U (i, j, val)
  VALUES (i, j, random());
...
INSERT INTO U (i, j, val)
  VALUES (n, k, random());

-- Нормирование степеней принадлежности
UPDATE U
  SET val = val / U1.tmp
  FROM (
    SELECT i, sum(val) AS tmp
    FROM U
    GROUP BY i) AS U1
  WHERE U1.i = U.i;
```

Рис. 2.14. Инициализация реляционных таблиц алгоритма *pgFCM*

ализацию таблиц SV, U и P, представлены на рис. 2.14. Таблица SV формируется путем выборки записей из таблицы SH. Для таблицы U за степень принадлежности вектора x_i кластеру j принимается случайное число, которое затем нормируется.

При инициализации таблицы P количество кластеров k задается процедурой $pgFCM$ и является ее параметром. Размерность пространства векторов d и мощность обучающей выборки n задаются на этапе подготовки. Номер итерации s и $delta$ инициализируются нулевыми значениями.

Табл. 2.6. Индексы таблиц базы данных алгоритма *pgFCM*

№ п/п	Таблица	Индексируемое поле (-я)
1	SV	i
2		ℓ
3		(i, ℓ)
4	C	ℓ
5		(j, ℓ)
6	SD	i
7		(i, j)
8	U	i
9		(i, j)
10	UT	(i, j)

Создание индексов таблиц

Перед началом вычислений выполняется создание индексов таблиц базы данных. *Индекс* представляет собой системный объект базы данных, используемый ядром СУБД для повышения производительности поиска данных. Индексы, используемые алгоритмом *pgFCM*, приведены в табл. 2.6. Индексы соответствуют комбинациям полей таблиц, задействованных в запросах, реализующих вычисления (см. рис. 2.15) и обновление таблиц базы данных (см. рис. 2.16).

Реализация вычислений

На рис. 2.15 приведены команды SQL, обеспечивающие вычисления алгоритма *pgFCM*.

В качестве функции расстояния ρ используется евклидово расстояние:

$$\rho(x_i, c_j) = \sqrt{\sum_{\ell=1}^{d} (x_{i\ell} - c_{j\ell})^2} \tag{2.11}$$

На шаге вычислений алгоритма $pgFCM$ производятся вычисления степеней принадлежности, центров кластеров и расстояний по формулам (2.7),

```
-- Вычисление центров кластеров
INSERT INTO C
  SELECT R.j, SV.l, sum(R.s * SV.val) / sum(R.s) AS val
  FROM (
    SELECT i, j, U.val^m AS s
    FROM U) AS R, SV
  WHERE R.i = SV.i
  GROUP BY j, l;

-- Вычисление расстояний
INSERT INTO SD
  SELECT i, j, sqrt(sum((SV.val - C.val)^2)) as dist
  FROM SV, C
  WHERE SV.l = C.l;
  GROUP BY i, j;

-- Вычисление степеней принадлежности
INSERT INTO UT
  SELECT i, j, SD.dist^(2.0^(1.0 - m)) * SD1.den AS val
  FROM (
    SELECT i, 1.0 / sum(dist^(2.0^(m - 1.0))) AS den
    FROM SD
    GROUP BY i) AS SD1, SD
  WHERE SD.i = SD1.i;
```

Рис. 2.15. Реализация вычислений

(2.6) и (2.11) соответственно.

Поскольку числитель дроби в формуле (2.7) вычисления степеней принадлежности не зависит от t, то для удобства вычислений эта формула переписана в следующем виде:

$$u_{ij} = \frac{\rho^{\frac{2}{1-m}}(x_i, c_j)}{\sum\limits_{t=1}^{k} \rho^{\frac{2}{m-1}}(x_i, c_t)} \tag{2.12}$$

Реализация обновления таблиц

Команды SQL, реализующие обновление таблиц P и U, представлены на рис. 2.16. В таблице P обновляются значения номера итерации s и значение функции *delta* (см. (2.9)). Таблица UT хранит временные значения степеней принадлежности, которые затем вносятся в таблицу U. Для быст-

```
1  -- Обновление служебной таблицы UT
2  SELECT max(abs(UT.val - U.val)) INTO tmp
3  FROM U, UT
4  WHERE U.i = UT.i AND U.j = UT.j;
5
6  INSERT INTO P
7  VALUES (d, k, n, steps, tmp);
8
9  -- Обновление таблицы степеней принадлежности
10 TRUNCATE U;
11 INSERT INTO U
12   SELECT * FROM UT;
```

Рис. 2.16. Обновление таблиц P и U

рого удаления всех записей таблицы U, полученных на предыдущем шаге, используется оператор `truncate`.

2.2.3. Вычислительные эксперименты

Для исследования эффективности разработанного алгоритма были проведены вычислительные эксперименты с параллельной СУБД PargreSQL [15] (рассматривается в главе 5) на платформе суперкомпьютера «Торнадо ЮУрГУ» [6]. Целью экспериментов является сравнение производительности алгоритма *pgFCM* со следующими параллельными алгоритмами нечеткой кластеризации данных, которые являются наиболее производительными из алгоритмов, рассмотренных ранее в разделе 1.3.2.

Параллельный алгоритм *BigFCM* предложен Гадири (Ghadiri) и др. в работе [93], в которой приведены результаты экспериментов, исследующих эффективность данного алгоритма на кластерной системе из одного мастер-узла и восьми рабочих узлов. В качестве входных данных использовался набор HIGGS (11 млн. 28-атрибутных элементов) [38].

Параллельный алгоритм *WCFC* предложен Хидри (Hidri) и др. в работе [105], в которой приведены результаты экспериментов, исследующих эффективность данного алгоритма на кластерной системе из 20 узлов. В качестве входных данных использовался набор KDD99 (4.9 млн. 41-атрибутных элементов) [271].

Табл. 2.7. Сравнение алгоритма *pgFCM* с аналогами

Набор данных			Аналог						***pgFCM***	
					Время, с					
n	d	k	Алгоритм	Платформа	Класт-я	Экспорт	Импорт	Всего	Время, с	Платформа
$1.1 \cdot 10^7$	28	2	*BigFCM* [93]	1 master CPU 3.4 ГГц, 8 slave CPU 3.2 ГГц	189	397	25	611	**531**	8 CPU 3.33 ГГц
$4.9 \cdot 10^6$	41	2	*WCFC* [105]	20 CPU 2.5 ГГц	5 100	203	10	5 313	**5 182**	16 CPU 3.33 ГГц

При сравнении алгоритм *pgFCM* запускался на конфигурациях кластера «Торнадо ЮУрГУ», которые обеспечивают примерно равную пиковую производительность, что и у алгоритма-конкурента. В экспериментах использовались те же наборы данных, на которых исследовалась эффективность алгоритмов-конкурентов в вышеуказанных работах. Данные о быстродействии конкурентов взяты из соответствующих статей. Результаты сравнения производительности алгоритмов приведены в табл. 2.7.

Можно видеть, что разработанный алгоритм показывает худшее, чем у конкурентов, быстродействие при выполнении нечеткой кластеризации. Однако, в отличии от аналогов, *pgFCM* работает в предположении, что данные, подлежащие кластеризации, непосредственно хранятся в базе данных и выполняет кластеризацию этих данных, не выходя за рамки СУБД. При использовании алгоритмов-аналогов, напротив, перед их запуском необходимо выполнять экспорт анализируемых данных из базы данных, а после запуска — импорт результатов кластеризации в базу данных. С учетом указанных накладных расходов на экспорт-импорт разработанный алгоритм *pgFCM* опережает аналоги по производительности.

2.3. Параллельный алгоритм *PDIC* поиска частых наборов

Алгоритм *PDIC (Parallel Dynamic Itemset Counting)* представляет собой параллельную версию алгоритма *DIC* [50], описанного в разделе 1.3.3, для многоядерных ускорителей архитектуры Intel MIC [78].

2.3.1. Проектирование алгоритма

Параллельный алгоритм *PDIC* использует *битовое представление* наборов и транзакций. Транзакция $T \subseteq \mathcal{D}$ (набор $I \subseteq \mathcal{I}$, соответственно) представляется в виде слова, в котором каждый $(p-1)$-й бит установлен в 1, если элемент $i_p \in T$ ($i_p \in I$, соответственно), и остальные биты установлены в 0. Количество битов в слове W зависит от используемой системы программирования и определяется как $W = \lceil \frac{m}{sizeof(byte)} \rceil$. В предлагаемой реализации используется система программирования C++, а набор и транзакция представлены значением типа данных `unsigned long long int`, таким образом, в данной реализации $W = 8$ и $m = 64$.

Введем *функцию битовой маски* $BitMask : \mathcal{I} \to \mathbb{N}$, которая возвращает натуральное число, являющееся битовым представлением указанной транзакции либо набора. Например, для набора $\{i_1, i_4, i_5\}$ функция $BitMask$ возвратит целое число 25, имеющее такое битовое представление, в котором 0-й, 3-й и 4-й биты установлены в 1, а остальные биты равны 0. Тогда *битовой картой* базы транзакций $\mathcal{D}$ назовем n-элементный одномерный массив $\mathcal{B}$, где $\mathcal{B}_j = BitMask(T_j)$, $1 \leqslant j \leqslant n$.

Использование битовых масок наборов и битовой карты базы транзакций упрощает подсчет поддержки наборов и обеспечивает векторизацию этой операции. Действительно, факт вхождения набора I в транзакцию T ($I \subseteq T$), установление которого необходимо при подсчете поддержки данного набора, может быть определен с помощью одной логической побитовой операции $BitMask(I)$ `AND` $BitMask(T)$=$BitMask(I)$, а циклическое выполнение данной операции может быть автоматически векторизовано компилятором.

Набор элементов, таким образом, реализуется как структура со следующими основными полями:

— $mask$ — битовая маска набора;

— k — количество элементов в наборе;

— *stop* — счетчик части транзакций, в которых проверено наличие данного набора;

— *supp* — поддержка данного набора;

— *shape* — вид данного набора (`BOX` или `CIRCLE`, либо `NULL`).

Множество наборов реализуется с помощью класса `vector` из стандартной библиотеки классов C++ (Standard Template Library) [141]. Данный класс реализует массив элементов, принадлежащих одному типу данных и обеспечивает операции добавления, удаления элементов, доступ к элементу по его индексу и итерацию элементов массива. Каждый из векторов `DASHED` и `SOLID` обеспечивает хранение двух множеств наборов алгоритма: *DashedCircle* и *DashedBox* и *SolidCircle* и *SolidBox* соответственно. Хранение двух множеств в рамках одного вектора обеспечивает меньшие накладные расходы, чем выделение одного вектора на каждое множество элементов: для перемещения элемента из одного множества в другое достаточно изменить значение поля *shape*.

2.3.2. Реализация алгоритма

Разработанная параллельная реализация поиска частых наборов представлена в алг. 2.3. Распараллеливанию подвергаются следующие стадии алгоритма: подсчет поддержки (см. алг. 2.4), нахождение и отбрасывание наборов-кандидатов, заведомо не являющихся частыми (см. алг. 2.5) и проверка завершения просмотра наборов-кандидатов по всей базе данных транзакций (см. алг. 2.6).

Реализация подсчета поддержки показана в алг. 2.4. Подсчет выполняется посредством двух вложенных циклов: внешний распараллеливается и выполняется по наборам-кандидатам, а внутренний — по транзакциям. Это позволяет избежать гонок данных при обновлении поддержки одного и того же набора разными нитями. Алгоритм различает два случая в зависимости от того, больше ли мощность множества наборов-кандидатов,

Алг. 2.3. PDIC(IN $\mathcal{B}$, $minsup$, M; OUT $\mathcal{L}$)

```
1:  SOLID.init(); DASHED.init()
2:  for all i ∈ 0..m − 1 do
3:      I.shape ← NIL
4:      I.mask ← SetBit(I.mask, i)
5:      I.stop ← 0; I.supp ← 0; I.k ← 1
6:      SOLID.push_back(I)
7:  end for
8:  k ← 1
9:  stop_max ← ⌈n/M⌉; stop ← 0
10: FirstPass(SOLID, DASHED)
11: while not DASHED.empty() do
12:     stop ← stop + 1
13:     if stop > stop_max then
14:         stop ← 1
15:     end if
16:     first ← (stop − 1) · M; last ← stop · M − 1
17:     k ← k + 1
18:     CountSupport(DASHED)
19:     Prune(DASHED)
20:     MakeCandidates(DASHED)
21:     CheckFullPass(DASHED)
22: end while
23: L ← {I | I ∈ SOLID ∧ I.shape = BOX}
24: return L
```

чем количество нитей, или нет. В первом случае внешний цикл распараллеливается на все доступные нити. Во втором случае алгоритм активирует режим вложенного параллелизма, и внешний цикл распараллеливается на количество нитей, равное количеству наборов-кандидатов.

Внутренний цикл по транзакциям распараллеливается таким образом, что каждая нить внешнего цикла инициирует (*fork*) одинаковое количество подчиненных нитей, выполняющих подсчет поддержки. Указанная техника позволяет сбалансировать нагрузку нитей на финальной стадии алгоритма, когда общее количество наборов-кандидатов последовательно уменьшается, что позволяет повысить общую производительность алгоритма.

Подпрограмма `FirstPass` выполняет подсчет поддержки для кандида-

Алг. 2.4. CountSupport(in out $DASHED$)

```
1: if DASHED.size() ⩾ num_of_threads then
2:     #pragma omp parallel for
3:     for all I ∈ DASHED do
4:         I.stop ← I.stop + 1
5:         for all T ∈ B_first .. B_last do
6:             if I.mask AND T = I.mask then
7:                 I.supp ← I.supp + 1
8:             end if
9:         end for
10:    end for
11: else
12:    omp_set_nested(true)
13:    #pragma omp parallel for num_threads(DASHED.size())
14:    for all I ∈ DASHED do
15:        I.stop ← I.stop + 1
16:        #pragma omp parallel for reduction(+:I.supp)
17:         num_threads(⌈num_of_threads / DASHED.size()⌉)
18:        for all T ∈ B_first .. B_last do
19:            if I.mask AND T = I.mask then
20:                I.supp ← I.supp + 1
21:            end if
22:        end for
23:    end for
24: end if
25: return DASHED
```

тов-синглтонов, которыми инициализируется множество наборов-кандидатов, по полной базе транзакций. Это позволяет сократить количество кандидатов-пар, обрабатываемых в алгоритмом дальнейшем.

Реализация отбрасывания наборов, заведомо не являющихся частыми, представлена в алг. 2.5. Максимально возможная поддержка набора вычисляется путем сложения текущего значения поддержки данного набора с количеством транзакций, которые еще не были обработаны. Если вычисленная максимально возможная поддержка меньше, чем предопределенное пороговое значение поддержки, переданное алгоритму, то данный набор заведомо не является частым, и может быть исключен из дальнейшего рас-

Алг. 2.5. PRUNE(IN OUT $DASHED$)

```
1:  #pragma omp parallel for
2:  for all I ∈ DASHED and I.shape = CIRCLE do
3:      if I.supp ⩾ minsup then
4:          I.shape ← BOX
5:      else
6:          supp_max ← I.supp + M · (stop_max − I.stop)
7:          if supp_max < minsup then
8:              I.shape ← NIL
9:              for all J ∈ DASHED and J.shape = CIRCLE do
10:                 if I.mask AND J.mask = I.mask then
11:                     J.shape ← NIL
12:                 end if
13:             end for
14:         end if
15:     end if
16: end for
17: DASHED.erase({I | I.shape = NIL})
18: return DASHED
```

смотрения. В соответствии с принципом *a priori* (любое подмножество частого набора должно быть частым набором) далее отбрасывается каждый набор-кандидат, который является надмножеством отброшенного набора.

По завершении отбрасывания подпрограмма `MakeCandidates` выполняет генерацию наборов-кандидатов для обработки на следующей итерации алгоритма. Новые кандидаты получаются посредством применения логической побитовой операции `OR` к каждой паре наборов из множества `DASHED`, помеченных как `BOX` (частый).

Финальным шагом обработки наборов-кандидатов является проверка завершения просмотра этих наборов по всей базе данных транзакций (см. алг. 2.6). Если просмотр завершен, набор перемещается в множество `SOLID`. Если при этом набор имеет поддержку не ниже порогового значения, он помечается как `BOX` (частый). Цикл обработки наборов-кандидатов распараллеливается с помощью директивы компилятора `#pragma omp parallel for`. По завершении обработки все искомые частые наборы будут переме-

Алг. 2.6. CheckFullPass(in out $DASHED$)

```
1: #pragma omp parallel for
2: for all I ∈ DASHED do
3:     if I.stop = stop_max then
4:         if I.supp ⩾ minsup then
5:             I.shape ← BOX
6:         end if
7:         SOLID.push_back(I)
8:         I.shape ← NIL
9:     end if
10: end for
11: DASHED.erase({I | I.shape = NIL})
12: return DASHED
```

щены в множество SOLID и помечены как BOX.

2.3.3. Вычислительные эксперименты

Цели, аппаратная платформа и наборы данных экспериментов

Табл. 2.8. Аппаратная платформа экспериментов

Характеристика	Хост	Сопроцессор
Модель, Intel Xeon	X5680	Phi (KNC), SE10X
Количество физических ядер	2×6	61
Гиперпоточность	2	4
Количество логических ядер	24	244
Частота, ГГц	3.33	1.1
Размер VPU, бит	128	512
Пиковая производительность, TFLOPS	0.371	1.076

Для исследования эффективности разработанного алгоритма были проведены вычислительные эксперименты. В качестве аппаратной платформы экспериментов использованы вычислительные узлы кластерной системы «Торнадо ЮУрГУ» [6], характеристики которой приведены в табл. 2.8.

В качестве тестовых данных использовались наборы, представленные в табл. 2.9. Синтетический набор данных 20M подготовлен с помощью гене-

Табл. 2.9. Наборы данных для экспериментов с алгоритмом *PDIC*

Набор данных	Вид	Транзакции			Частые наборы (при $minsup = 0.1$)	
		n	m	Ср. длина	Количество	k_{max}
20M	Синтетический	$2 \cdot 10^7$	64	40	4 606	6
Tornado20M	Реальный	$2 \cdot 10^7$	64	15	346	4

ратора IBM Quest Data Generator [269], использованного для исследования эффективности оригинального алгоритма DIC в работе [50]. В итоге набор данных 20M содержит 4 606 частых набора, самый длинный из которых состоит из 6 элементов.

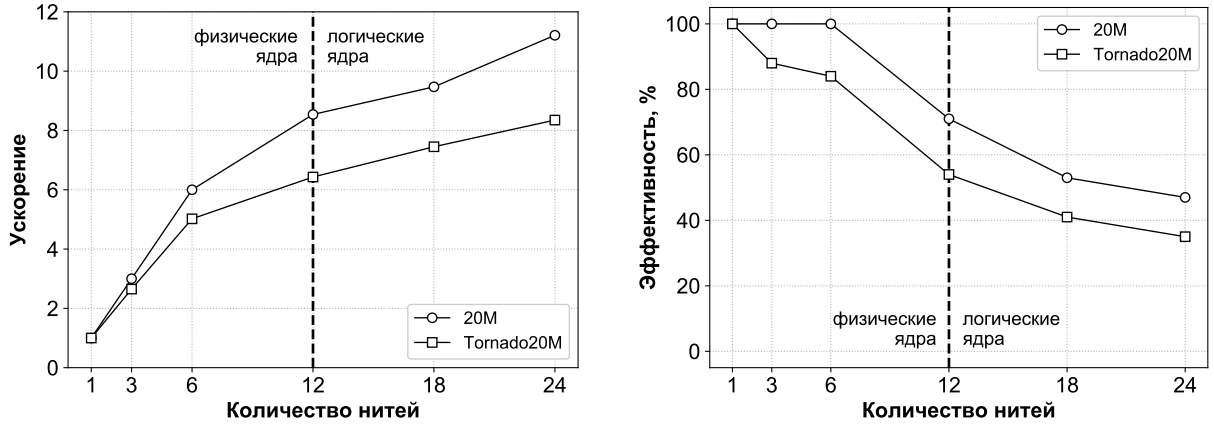
Набор Tornado20M представляет собой журнал с показаниями датчиков напряжения в вычислительных узлах суперкомпьютера «Торнадо ЮУрГУ» [6], снятых в течение одного месяца. Данный журнал используется для нахождения устойчивых ассоциативных правил, связующих вычислительные шкафы, полки, узлы суперкомпьютера и опасные значения напряжения. «Торнадо ЮУрГУ» состоит из 8 шкафов, каждый шкаф состоит из 8 полок, каждая полка состоит из 6 узлов. Рассматривается 4 возможных значения измеряемого напряжения, для каждого из которых различают 4 статуса («меньше нормы», «норма», «больше нормы», «ошибка измерения»). В связи с этим транзакция журнала может быть закодирована с помощью 64 бит (8 бит на номер шкафа, 8 бит на номер полки, 48 бит на 6 узлов, где каждая пара битов отражает статус измеренного напряжения). В итоге набор данных Tornado20M содержит 340 частых наборов, самый длинный из которых состоит из 4 элементов.

В экспериментах исследовались производительность и масштабируемость алгоритма *PDIC*. Под производительностью понимается время работы алгоритма без учета времени загрузки данных в память и выдачи результата. Масштабируемость параллельного алгоритма означает его способность адекватно адаптироваться к увеличению параллельно работающих вычислительных элементов (процессов, процессоров, нитей и др.) и характеризуется ускорением и параллельной эффективностью, которые

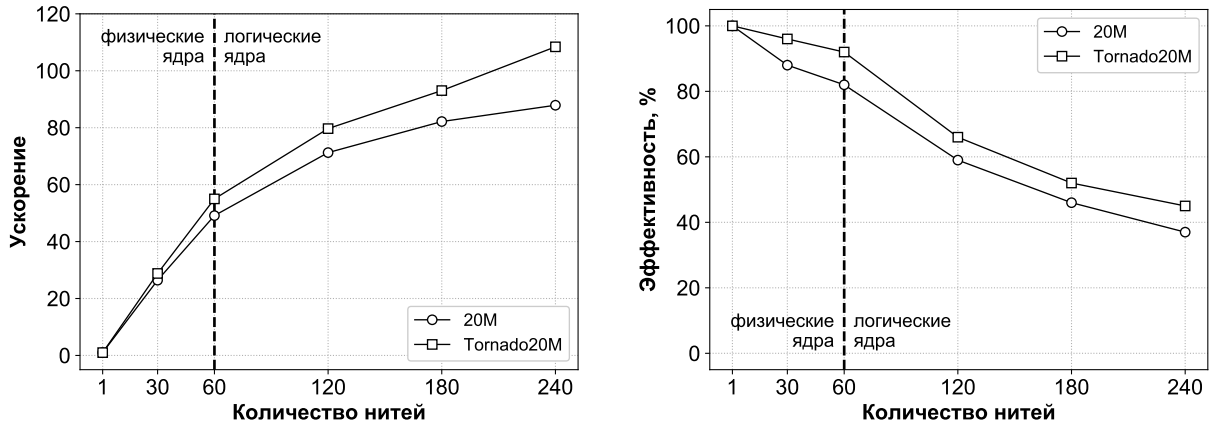
определяются следующим образом [2].

Ускорение и *параллельная эффективность* параллельного алгоритма, запускаемого на k нитях, вычисляются как $s(k) = \frac{t_1}{t_k}$ и $e(k) = \frac{s(k)}{k}$ соответственно, где t_1 и t_k — время работы алгоритма на одной и k нитях соответственно.

Результаты экспериментов



(a) Платформа 2×Intel Xeon



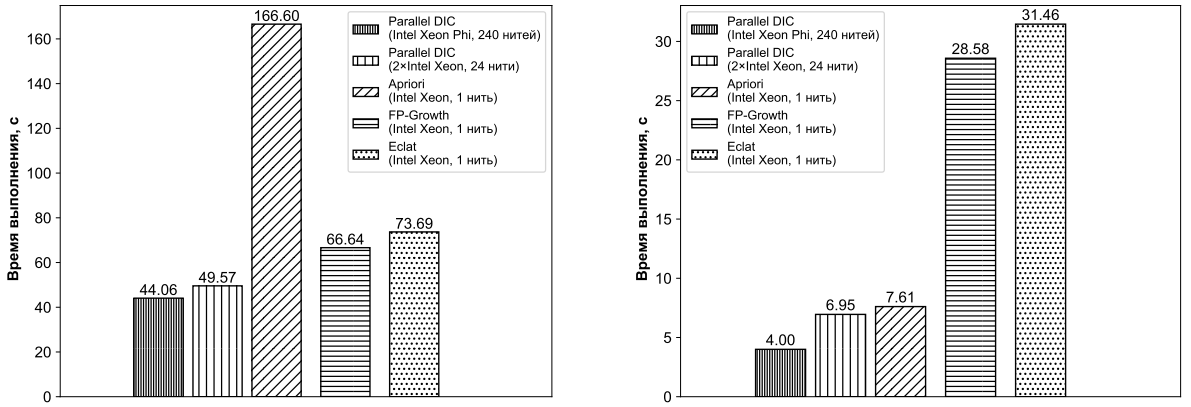
(b) Платформа Intel Xeon Phi

Рис. 2.17. Масштабируемость алгоритма *PDIC*

Результаты экспериментов по исследованию ускорения и параллельной эффективности алгоритма *PDIC* представлены на рис. 2.17. На платформе Intel Xeon Phi алгоритм *PDIC* показывает ускорение, близкое к линейному

и параллельную эффективность, близкую к 100%, когда количество нитей, на которых запущен алгоритм, совпадает с количеством физических ядер процессора. Если при запуске алгоритм использует более одной нити на физическое процессорное ядро, то ускорение становится сублинейным (его значение уменьшается до 88 и 108 соответственно для наборов данных 20M и Tornado20M), а параллельная эффективность снижается соответствующим образом (до 37% и 45% для соответствующих наборов данных).

На платформе вычислительного узла с двумя процессорами Intel Xeon наблюдается схожая картина, хотя и с несколько более скромными результатами для набора данных Tornado20M. Ускорение и параллельная эффективность алгоритма в экспериментах на наборе данных Tornado20M уменьшаются до значений 8 и 35% соответственно, когда алгоритм запускается на максимальном количестве физических ядер.



(a) Набор 20M (b) Набор Tornado20M

Рис. 2.18. Производительность алгоритма *PDIC*

Результаты экспериментов по исследованию производительности алгоритма *PDIC* представлены на рис. 2.18. Можно видеть, что алгоритм *PDIC* работает до полутора раз быстрее на платформе многоядерного процессора Intel Xeon Phi, чем на платформе вычислительного узла, состоящего из двух обычных процессоров Intel Xeon. Алгоритм *PDIC* на платформе многоядерного процессора Intel Xeon Phi опережает в два раза лучшие резуль-

таты последовательных алгоритмов-конкурентов на платформе процессора Intel Xeon.

Табл. 2.10. Влияние векторизации на производительность алгоритма *PDIC*

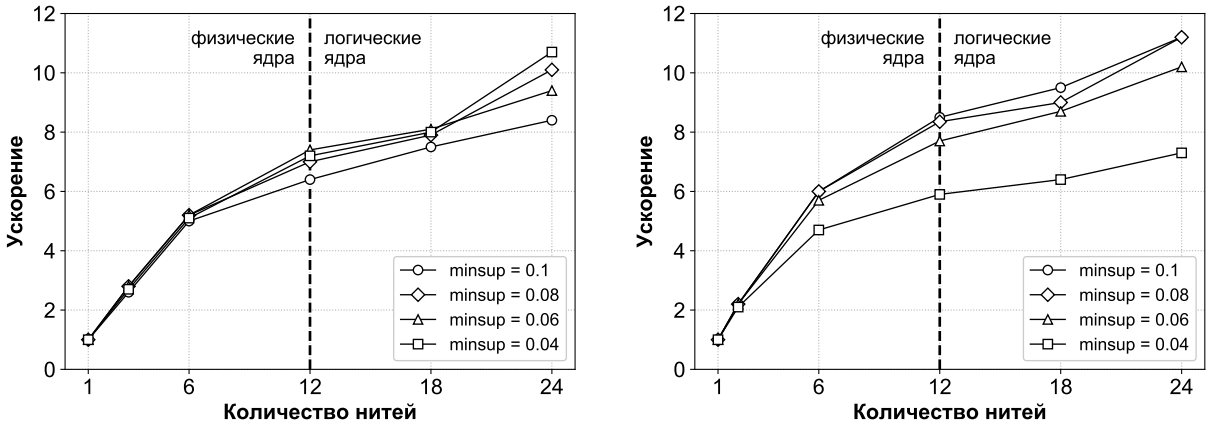
Платформа	Время выполнения алгоритма (с), параметр компиляции	
	векторизация включена	векторизация отключена
Intel Xeon Phi	4.00	10.36
Intel Xeon	6.95	8.55

В табл. 2.10 представлены результаты экспериментов с набором данных Tornado20M, показывающих выгоду векторизации вычислений. Можно видеть, что производительность алгоритма *PDIC* на платформе Intel Xeon Phi увеличивается в два раза, если обеспечивается векторизация вычислений.

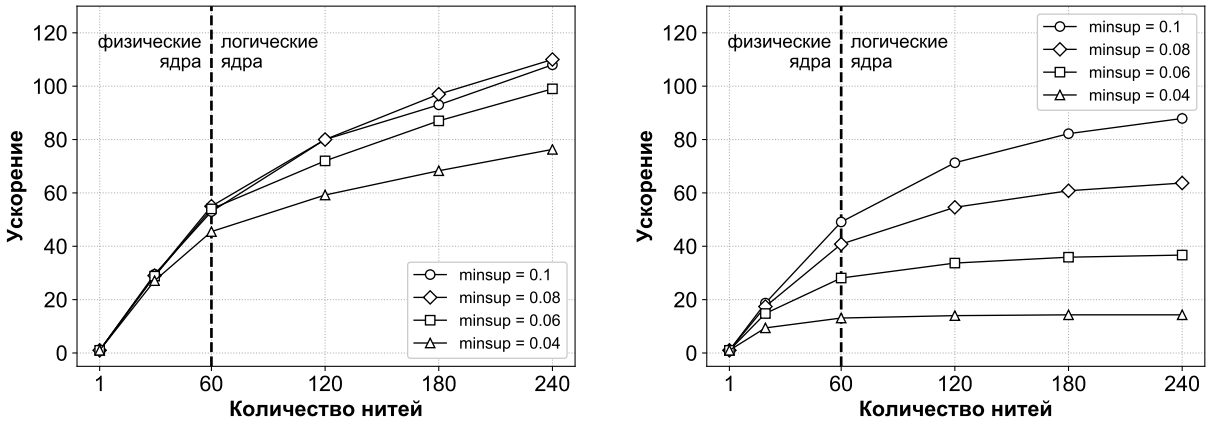
На рис. 2.19 показано влияние порога *minsup* на ускорение алгоритма. На обеих платформах и для обоих наборов данных ускорение алгоритма ожидаемо страдает от уменьшения значения *minsup*, поскольку это существенно увеличивает количество наборов-кандидатов для вычисления поддержки. Алгоритм *PDIC* все еще показывает лучшее ускорение, когда задействованы только физические ядра, и лучшее ускорение на платформе Intel Xeon Phi, чем на двухпроцессорной системе Intel Xeon.

2.4. Параллельный алгоритм *DDCapriori* поиска частых наборов

Алгоритм *DDCapriori (Data Distribution Counting apriori)* представляет собой параллельную версию алгоритма *Apriori* [34], описанного в разделе 1.3.3, для многоядерных процессоров архитектуры Cell BE [112].



(a) Платформа 2×Intel Xeon



(b) Платформа Intel Xeon Phi

Рис. 2.19. Влияние параметра *minsup* на ускорение алгоритма (слева — набор Tornado20M, справа — набор 20M)

2.4.1. Проектирование алгоритма

Разработанный алгоритм *DDCapriori* (см. алг. 2.7) использует вычислительную модель «мастер-рабочие». *Нить-мастер* запускается на управляющем ядре PPE (POWER Processing Element) и распределяет задания для рабочих. *Нити-рабочие* запускаются на вычислительных ядрах SPE (Synergistic Processing Element) и выполняют обработку данных, получаемых от мастера. Алгоритм использует операции *Send*, *Recv* и *MakeTask*, которые имеют следующую семантику.

Операция $Send(dst, msg)$ выполняет асинхронную отправку сообщения

Алг. 2.7. DDCapriori

Master

(in $\mathcal{D}, minsup$; out $\mathcal{L}$)

▷ Отправка заданий рабочим
1: **for all** S_j **do**
2: $task_j \leftarrow \mathsf{MakeTask}(D_j)$
3: $\mathsf{Send}(S_j, task_j)$
4: $\mathsf{Send}(S_j, \mathcal{D})$
5: $\mathsf{Send}(S_j, I)$
6: **end for**
▷ Вычисление $\mathcal{L}_1$
7: **for all** S_j **do**
8: $\mathsf{Recv}(S_j, supp_j)$
9: **for all** $i \in \mathcal{I}$ **do**
10: $supp(i) \leftarrow \sum_{j=1}^{n} supp_j(i)$
11: **end for**
12: **end for**
13: $\mathcal{L}_1 \leftarrow \mathcal{I} \setminus \{i \mid supp(i) < minsup\}$
▷ Обработка k-наборов
14: $k \leftarrow 2$
15: **while not** $Stop$ **do**
▷ Генерация кандидатов
16: $C_k \leftarrow \textsc{MakeCand}(\mathcal{L}_{k-1})$
▷ Вычисление результата
17: **if** $C_k = \varnothing$ **then**
18: $\mathcal{L} \leftarrow \cup_{j=1}^{k-1} \mathcal{L}_j$
19: $Stop \leftarrow \texttt{TRUE}$
20: **break**
21: **end if**
▷ Отправка заданий рабочим
22: **for all** S_j **do**
23: $msg_j \leftarrow \textsc{MakeTask}(C_k^j)$
24: $\mathsf{Send}(S_j, msg_j)$
▷ Прием и агрегация результатов
25: $\mathsf{Recv}(S_j, \mathcal{L}_k^j)$
26: **end for**
27: $\mathcal{L}_k \leftarrow \cup_{j=1}^{num_{slaves}} \mathcal{L}_k^j$
28: $k \leftarrow k + 1$
29: **end while**
30: **return** $\mathcal{L}$

Slave

(in $task$, $minsup$; out $\mathcal{L}_k$)

▷ Прием задания от мастера
1: $k \leftarrow 1$
2: $\mathsf{Recv}(M, task)$
3: $\mathsf{Recv}(M, \mathcal{D})$
4: $\mathsf{Recv}(M, C)$
▷ Обработка 1-наборов
5: **for all** $c \in C$ **do**
6: $supp(c) \leftarrow \textsc{CountSupp}(c, D)$
7: **end for**
8: $\mathsf{Send}(M, supp)$
▷ Обработка k-наборов
9: **while not** $Stop$ **do**
10: $k \leftarrow k + 1$
11: **if** $C = \varnothing$ **then**
12: $Stop \leftarrow \texttt{TRUE}$
13: **break**
14: **end if**
▷ Вычисление поддержки
15: **for all** $c \in C$ **do**
16: $supp(c) \leftarrow \textsc{CountSupp}(c, \mathcal{D})$
17: **end for**
▷ Отбрасывание
18: $\mathcal{L}_k \leftarrow C \setminus \{c \mid supp(c) < minsup\}$
▷ Отправка результата мастеру
19: $\mathsf{Send}(M, \mathcal{L}_k)$
20: **end while**

msg получателю dst. Операция $Recv(src, msg)$ выполняет синхронное получение сообщения msg от отправителя src.

Операция $MakeTask(d)$ создает и возвращает задание на обработку данных d. *Задание* представляет собой совокупность адреса, по которому располагаются данные в оперативной памяти, и размера этих данных. В качестве данных может выступать множество транзакций либо множество наборов-кандидатов.

Идея предлагаемого параллельного алгоритма заключается в том, чтобы возложить на мастера задачу формирования множеств C_k и $\mathcal{L}_k$, а на рабочих — вычисление поддержки для наборов-кандидатов из C_k.

В отличие от подхода *Data Distribution*, при вычислении множества $\mathcal{L}_1$ множество транзакций B разбивается на подмножества, которые затем назначаются для обработки разным рабочим. Рабочий рассматривает каждый элемент в своем подмножестве транзакций как одноэлементный кандидат и увеличивает его поддержку всякий раз, когда этот кандидат встречается в транзакциях. При вычислении множества $\mathcal{L}_k$ $(k > 1)$ между рабочими распределяются кандидаты, а не транзакции.

Деятельность мастера кратко может быть описана следующим образом. После создания рабочих мастер отправляет каждому из них первое задание и все множество транзакций, после чего переходит в состояние ожидания. По получении результатов мастер выполняет их агрегацию, формируя таким образом множество кандидатов единичной длины, и отбрасывание редких кандидатов, формируя множество частых наборов единичной длины. Далее мастер полагает счетчик k равным 1 и циклически выполняет следующую последовательность действий. Из множества частых k-наборов формируется множество кандидатов длины $k + 1$. Затем мастер формирует задания на обработку полученного множества кандидатов, отправляет их рабочим и ожидает от них результаты вычислений (значения поддержки). Если мастеру не удается сформировать кандидаты длины $k + 1$, то мастер прерывает цикл, уничтожает рабочих и вычисляет результирующее множество частых наборов.

Деятельность рабочего состоит в следующем. Получив задание от мастера, рабочий формирует множество кандидатов единичной длины из своего подмножества транзакций, после чего отправляет результаты мастеру. Далее рабочий циклически выполняет следующую последовательность действий: ожидание от мастера подмножества кандидатов, вычисление поддержки и отправка результата вычислений мастеру. Цикл прерывается, если получено задание на обработку пустого множества кандидатов.

2.4.2. Реализация алгоритма

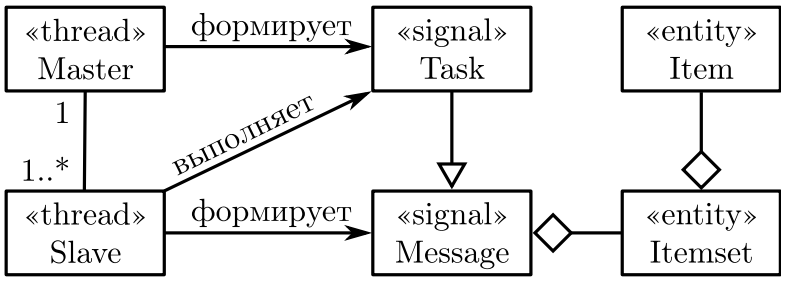


Рис. 2.20. Диаграмма классов, реализующих алгоритм *DDCapriori*

Диаграмма классов, реализующих предложенный алгоритм, представлена на рис. 2.20. Класс *Master* реализует нить-мастер и выполняет следующие основные функции: управление рабочими и формирование множеств C_k и $\mathcal{L}_k$. Экземпляр класса *Master* исполняется на управляющем ядре PPE.

Класс *Slave* реализует нить-рабочего и выполняет расчет опорных чисел для кандидатов из множества C_k. Экземпляры класса *Slave* создаются экземпляром класса *Master* на вычислительных ядрах SPE (по одному на каждом вычислительном ядре).

Класс *Task* служит для управления рабочими, выполняет роль сигнала и хранит входные данные для рабочего. Класс *Message* выполняет роль сигнала, отправляемого мастеру рабочим, и хранит результаты вычислений рабочего.

Реализация данных классов выполнена на языке C с использованием библиотеки IBM SDK for Multicore Acceleration [35]. Процессор IBM Cell BE

оперирует векторами длиной 128 битов. В зависимости от длины идентификатора объекта, в одном векторе могут быть размещены от двух до 16 целочисленных идентификаторов. В представляемой реализации используются 32-битные идентификаторы, то есть в одном векторе размещаются 4 идентификатора.

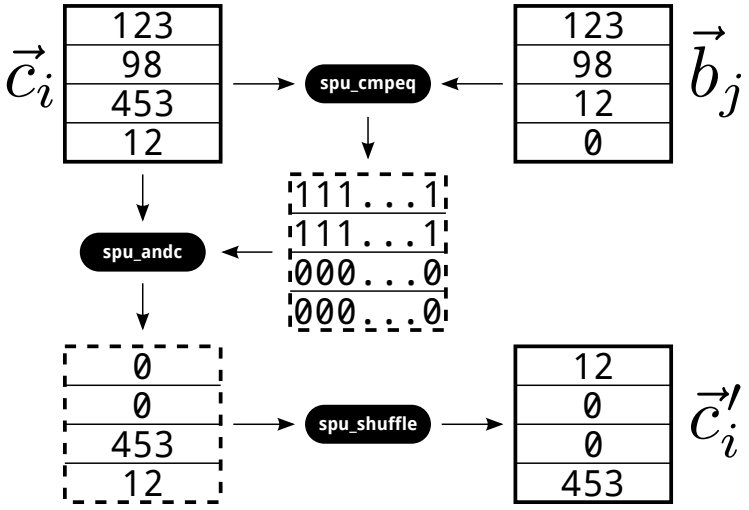


Рис. 2.21. Реализация проверки вхождения кандидата в транзакцию

При проверке вхождения кандидата в транзакцию $c \subset b$ кандидат c и транзакция b разбиваются на вектора $\vec{c}_i$ и $\vec{b}_j$. Каждый из векторов кандидата сравнивается с каждым вектором транзакции с помощью последовательности векторных операций, которая показана на рис. 2.21.

С помощью функции сравнения векторов `spu_cmpeq` формируется вектор-маска, в котором биты элемента установлены, если соответствующие элементы векторов $\vec{c}_i$ и $\vec{b}_j$ равны, и сброшены в противном случае. Далее к исходному вектору кандидата $\vec{c}_i$ и полученной маске применяется векторная функция `spu_andc`, которая выполняет логическую побитовую операцию $A \wedge \neg B$, обращая в нуль все элементы исходного вектора кандидата $\vec{c}_i$, совпадающие с соответствующим элементом вектора транзакции $\vec{b}_j$. Затем с помощью векторной функции `spu_shuffle` выполняется циклический сдвиг полученного вектора на один элемент.

Данная процедура, примененная N раз, где N — длина вектора, позволяет обнулить все элементы вектора $\vec{c}_i$, содержащиеся в векторе $\vec{b}_j$.

Таким образом, после применения этой процедуры ко всем парам векторов из кандидата и транзакции, в кандидате останутся только те элементы, которые не входят в транзакцию. Если таких элементов не осталось, то кандидат входит в данную транзакцию.

Поскольку каждый вектор кандидата проверяется на вхождение в каждый вектор транзакции, то для вычисления поддержки кандидатов из множества C_k потребуется следующее количество векторных операций:

$$O\left(\sum_{i=1}^{|C_k|}\sum_{j=1}^{|\mathcal{D}|}|c_i|\cdot|b_j|\right). \tag{2.13}$$

Заметим, что в случае, когда векторные функции не используются, для выполнения проверки требуется такое же количество скалярных операций, скорость выполнения которых на вычислительных ядрах SPE существенно ниже [60].

2.4.3. Вычислительные эксперименты

Табл. 2.11. Аппаратная платформа экспериментов

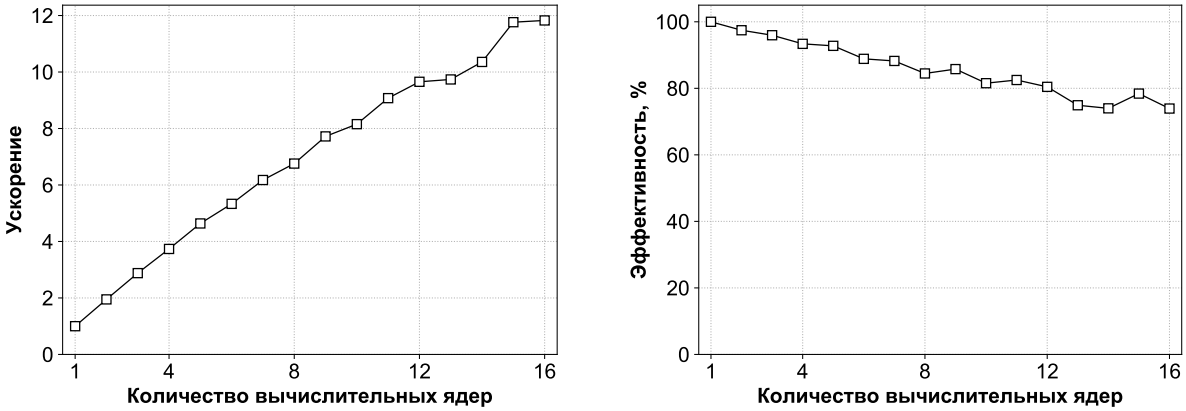
Процессор	PowerXCell8i
Количество вычислительных ядер	2×8
Тактовая частота, ГГц	3.2
Пропускная способность шины EIB, ГБ/с	200

Для оценки эффективности разработанного алгоритма были проведены вычислительные эксперименты. Характеристики вычислительной системы, использованной в экспериментах, приведены в табл. 2.11.

Табл. 2.12. Параметры экспериментов

Количество транзакций $\lvert\mathcal{D}\rvert$	10^6
Минимальная поддержка $minsup$	20 000
Количество рабочих	от 1 до 16
Реализация проверки $I \subset T$	векторная, скалярная

Параметры экспериментов представлены в табл. 2.12. В качестве исходных данных экспериментов был взят стандартный тестовый набор данных о посещении страниц web-сайта `msnbc.com` [52]. База транзакций $\mathcal{D}$ в тестовой задаче представляет собой записи о посещениях страниц сайта. Каждая запись содержит отметку о том, к какой семантической категории принадлежат посещенные за один сеанс страницы. В экспериментах осуществляется поиск наборов категорий страниц, часто посещаемых совместно (в течение одной сессии пользователя).



(a) Ускорение

(b) Параллельная эффективность

Рис. 2.22. Масштабируемость алгоритма *DDCapriori*

В экспериментах исследовались ускорение и параллельная эффективность алгоритма в зависимости от количества задействованных вычислительных ядер. Результаты экспериментов представлены на рис. 2.22. Можно видеть, что алгоритм *DDCapriori* демонстрирует ускорение, близкое к линейному, и параллельную эффективность не ниже 73%.

В экспериментах также было проведено сравнение ускорения разработанного алгоритма и алгоритма *Count Distribution* для многоядерного процессора IBM Cell BE, на основе результатов, опубликованных авторами данного алгоритма в работе [76]. Сравнение показывает (см. рис. 2.23) лучшее ускорение разработанного алгоритма.

В рамках экспериментов проведено исследование выгоды применения векторных операций вместо скалярных при проверке вхождения кандидата

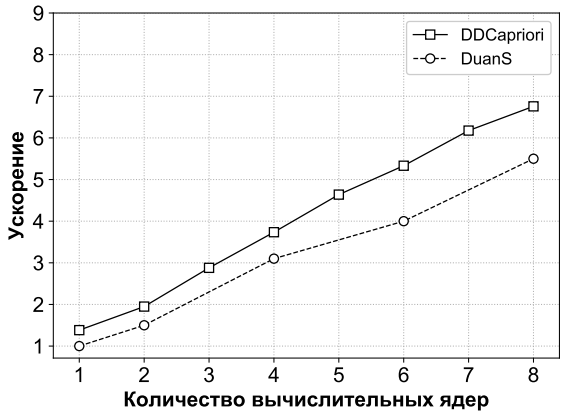
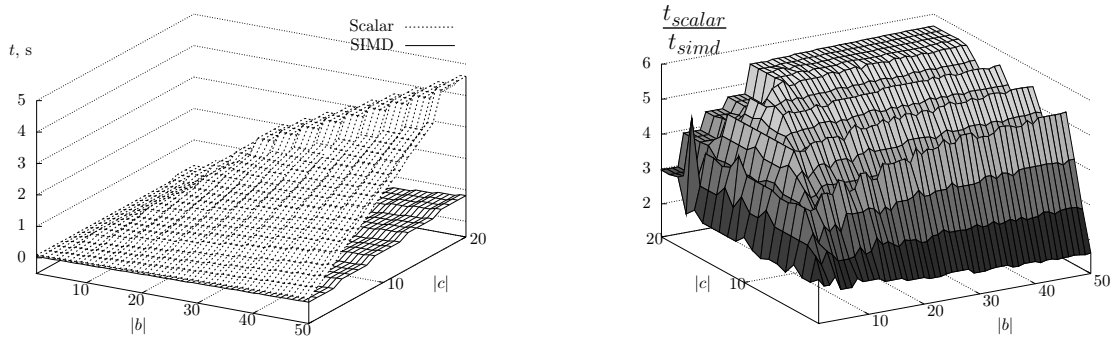


Рис. 2.23. Сравнение алгоритмов *DDCapriori* и *CountDistribution*



(a) Время (b) Выгода от использования векторных функций

Рис. 2.24. Использование векторных функций при проверке вхождения кандидата в транзакцию

в транзакцию в зависимости от длин кандидата и транзакции. Результаты данной серии экспериментов представлены на рис. 2.24 и показывают, что выигрыш от использования векторных операций прямо пропорционален длинам кандидата и транзакции.

2.5. Выводы по главе 2

В данной главе рассмотрен подход к выполнению кластеризации данных с использованием параллельной СУБД применительно к двум следую-

щим задачам: кластеризация вершин графа (разбиение графа) и нечеткая кластеризация данных.

Предложен оригинальный алгоритм *dbParGraph* кластеризации графа, количество вершин и количество ребер которого таковы, что он не может быть целиком размещен в оперативной памяти. Разработанный алгоритм предполагает представление графа в виде реляционной таблицы (списка ребер), распределяемой по узлам кластерной вычислительной системы. Разбиение состоит из двух этапов: поиск сообществ и многоуровневое разбиение.

Поиск сообществ выполняется в параллельной СУБД и предполагает нахождение таких подмножеств вершин данного графа, в каждом из которых вершины плотно связаны и редко связаны с другими частями графа. Далее выполняется укрупнение найденных сообществ посредством объединения двух сообществ в одно, если имеется хотя бы одно соединяющее их ребро. Укрупнение выполняется, пока число сообществ не будет равно числу узлов вычислительного кластера, на котором запускается параллельная СУБД. Ребра вершин, которые принадлежат одному и тому же сообществу, назначаются для обработки в один и тот же фрагмент базы данных.

Многоуровневое разбиение состоит из трех последовательно выполняемых стадий: огрубление, начальное разбиение и уточнение. Стадия огрубления выполняется с помощью параллельной СУБД и обеспечивает редукцию исходного графа до размеров, позволяющих разместить граф и промежуточные данные в оперативной памяти. На стадии начального разбиения огрубленный граф экспортируется из базы данных и подается на вход сторонней утилиты, которая выполняет начальное разбиение графа в оперативной памяти с помощью одного из известных алгоритмов. Результат начального разбиения импортируется в базу данных в виде реляционной таблицы. На стадии уточнения разбиения параллельная СУБД с помощью запросов к таблице, полученной на предыдущей стадии, формирует финальное разбиение графа в виде реляционной таблицы из двух столбцов: номер вершины графа и номер подграфа, которому принадлежит эта вер-

шина.

Результаты проведенных вычислительных экспериментов показывают, что данное решение может обеспечить сверхлинейное ускорение алгоритма при приемлемом качестве разбиения. Предложенное решение может быть применено в ряде теоретических и практических задач: раскраска графа, определение числа и состава компонент связности графа, проектирование больших интегральных схем и топологии локальной сети и др.

Предложен оригинальный алгоритм *pgFCM* нечеткой кластеризации данных с помощью параллельной СУБД. Выполнено проектирование схемы базы данных, позволяющей хранить исходные и промежуточные данные в реляционных таблицах и выполнять вычисления с помощью агрегатных функций SQL.

Результаты проведенных вычислительных экспериментов показывают, что предложенный алгоритм для СУБД на больших объемах данных более эффективен по сравнению с традиционной реализацией нечеткой кластеризации, предполагающей использование оперативной памяти. Предложенное решение может быть применено в обработке медицинских изображений [218], геоинформационных данных [149] и др.

Предложен оригинальный параллельный алгоритм решения задачи поиска частых наборов *PDIC* для многоядерного процессора Intel Xeon Phi. Алгоритм используется битовое представление транзакций и наборов. Использование битовых масок наборов и битовой карты базы транзакций упрощает подсчет поддержки наборов и обеспечивает векторизацию этой операции. Результаты проведенных вычислительных экспериментов показывают, что разработанный алгоритм демонстрирует ускорение, близкое к линейному и параллельную эффективность, близкую к 100%, когда количество нитей, на которых запущен алгоритм, совпадает с количеством физических ядер процессора.

Битовое представление наборов и транзакций, используемое в алгоритме *PDIC*, означает, что каждые набор и транзакция, представляемые целым положительным числом, не могут состоять из более чем 64 объектов

(количество размещений с повторениями из двух элементов 0 и 1 по 8). Данное ограничение, очевидно, неприемлемо для поиска частых наборов в задаче анализа корзины покупок в супермаркете [50] или в задаче анализа данных ДНК-микрочипов [55] и проч. Однако алгоритм *PDIC* потенциально применим для поиска шаблонов в медицинских данных, что может быть подтверждено следующими прецедентами в научных публикациях. В работе [135] описан поиск шаблонов риска заболевания у пациентов клиники, где количество атрибутов не превышает 30. В работах [177,183] описан механизм трансформации данных медицинских карт в формат транзакций для последующего поиска частых наборов. В экспериментах авторы взяли не более 25 атрибутов из более чем 100 атрибутов медицинской карты пациента, поскольку выбранные атрибуты могут дать полную картину течения болезни. Кроме того, опыт авторов показал, что шаблоны, в которые входит более чем пять медицинских атрибутов, трудно поддаются интерпретации врачами. В работе [193] описан поиск шаблонов в базе данных больницы с более чем 2.5 млн. транзакций с данными пациентов, включая атрибуты, касающиеся демографии, диагностики и употребления лекарств.

Предложен оригинальный параллельный алгоритм решения задачи поиска частых наборов *DDCapriori* для многоядерного процессора IBM Cell BE. Схема распараллеливания выглядит следующим образом. База транзакций реплицируется на каждом вычислительном ядре. Кандидаты в частые наборы разделяются на непересекающиеся подмножества, которые распределяются по вычислительным ядрам. В реализации использована модель «мастер-рабочие». Нить-мастер запускается на управляющем ядре процессора и выполняет следующие функции: управление нитями-рабочими и формирование наборов-кандидатов на каждом шаге алгоритма. Нити-рабочие запускаются на вычислительных ядрах процессора и выполняют вычисление поддержки наборов-кандидатов. Реализация выполнена на языке программирования C с использованием векторных функций библиотеки IBM Cell Broadband Engine SDK, которые позволяют эффективно реализовать операции вычисления поддержки наборов, генерации и отбрасы-

вания кандидатов. Результаты проведенных вычислительных экспериментов показывают, что разработанный алгоритм демонстрирует ускорение, близкое к линейному, и параллельную эффективность не ниже 75%.

Результаты, описанные в этой главе, опубликованы в работах [9, 13, 14, 16, 27, 187, 265, 266].

Глава 3. Анализ временных рядов

Данная глава посвящена параллельным методам интеллектуального анализа временных рядов на платформе современных многопроцессорных многоядерных вычислительных систем. Рассмотрены следующие задачи: поиск похожих подпоследовательностей во временном ряде и поиск диссонансов во временном ряде. Описаны оригинальные алгоритмы решения указанных задач: алгоритм *PBM* для кластерных систем с узлами на базе многоядерных ускорителей и алгоритм *MDD* для многоядерных ускорителей. Представлены результаты вычислительных экспериментов, исследующих эффективность разработанных алгоритмов.

3.1. Параллельный алгоритм поиска похожих подпоследовательностей *PBM*

3.1.1. Проектирование алгоритма

Базовые принципы распараллеливания

Распараллеливание поиска похожих подпоследовательностей временного ряда на вычислительных кластерах с узлами на базе многоядерных процессорных систем архитектуры Intel MIC основано на следующих принципах: параллелизм по данным, выравнивание данных в памяти, векторизация вычислений.

Параллелизм по данным на уровне вычислительных узлов кластерной системы реализуется с помощью фрагментации временного ряда. Временной ряд разбивается на фрагменты (подпоследовательности) примерно равной длины, каждый из которых размещается на диске отдельного вычислительного узла, выполняющего обработку данного фрагмента. Для обработки фрагмента на каждом узле запускается вычислительный процесс, и все процессы используют один и тот же алгоритм. В процессе обработки про-

цессы обмениваются данными для сокращения объема вычислений. Обмены данными между процессами реализуются на базе технологии MPI [96]. Параллелизм по данным внутри узла кластерной системы реализуется следующим образом. В рамках вычислительного процесса на ядрах процессора узла запускаются (*fork*) нити, разделяющие оперативную память узла. Нити осуществляют параллельную обработку фрагмента на базе технологии OpenMP [150].

Векторизация циклов заключается в способности компилятора при построении исполняемого кода программы преобразовать последовательность скалярных операций в теле цикла в одну векторную инструкцию. Векторизация является одним из ключевых условий достижения высокой производительности параллельных программ [37]. В то же время такие факторы как зависимость операций в теле цикла по данным (например, обращение к одному и тому же элементу массива на разных итерациях цикла), наличие условных операторов или вызовов пользовательских функций, невыровненный доступ к данным и др. препятствуют векторизации [37]. Таким образом, при поиске похожих подпоследовательностей вычисления организуются таким образом, чтобы увеличить, насколько это возможно, количество векторизуемых циклов.

Выравнивание данных в оперативной памяти существенно влияет на эффективность исполнения векторизованных циклов [37]. Если начальный адрес обрабатываемого в цикле массива данных не выровнен на ширину векторного регистра (т.е. на количество элементов, которые могут быть загружены в векторный регистр), то имеет место эффект разделения цикла (*loop peeling*). Компилятор разбивает цикл на три части, где первая часть итераций, которые обращаются к памяти с начального адреса массива до первого выровненного адреса, и третья часть итераций с последнего выровненного адреса до конечного адреса массива, векторизуются отдельно. В соответствии с этим в алгоритме используется компоновка данных в оперативной памяти, которая обеспечивает выровненный доступ к подпоследовательностям временного ряда и соответствующая этой компоновке

вычислительная схема, в которой вычисления реализованы в виде векторизуемых циклов.

Фрагментация временного ряда

Фрагментация временного ряда обеспечивает параллелизм по данным на уровне вычислительных узлов кластерной системы и осуществляется следующим образом. Для предотвращения потери результирующих подпоследовательностей, находящихся на стыке фрагментов, предлагается техника *разбиения с перекрытием*, которая заключается в следующем. В конец каждого фрагмента временного ряда, за исключением последнего по порядку, добавляется $n-1$ элементов ряда, взятых с начала следующего фрагмента, где n — длина поискового запроса. Формальное определение разбиения с перекрытием выглядит следующим образом.

Пусть $N = |T| - n + 1 = m - n + 1$ — количество подпоследовательностей, которые необходимо обработать, F — количество фрагментов, $T^{(k)}$ — k-й фрагмент временного ряда $T = (t_1, t_2, \ldots, t_m)$, где $0 \leqslant k \leqslant F-1$. Тогда *фрагмент* $T^{(k)}$ определяется как подпоследовательность $T_{start,\,len}$, где

$$start = k \cdot \lfloor \tfrac{N}{F} \rfloor + 1$$

$$len = \begin{cases} \lfloor \frac{N}{F} \rfloor + (N \bmod F) + n - 1, & k = F - 1 \\ \lfloor \frac{N}{F} \rfloor + n - 1, & otherwise. \end{cases} \tag{3.1}$$

Количество фрагментов (вычислительных узлов кластерной системы) F выбирается таким образом, чтобы фрагмент и соответствующие вспомогательные данные алгоритма могли быть размещены в оперативной памяти вычислительного узла.

Компоновка данных в памяти

Предлагаемая компоновка данных в оперативной памяти вычислительного узла кластерной системы и обеспечивает представление фрагмента временного ряда и вспомогательных данных алгоритма в виде выровнен-

ных в памяти матриц, циклы обработки которых векторизуются компилятором.

Выравнивание данных выполняется следующим образом. Пусть обработка некой подпоследовательности $T_{i,n}$ ряда T осуществляется с использованием векторного регистра, вмещающего w вещественных чисел. Если длина подпоследовательности не кратна w, то подпоследовательность дополняется фиктивными нулевыми элементами. Обозначим количество фиктивных элементов за $pad = w - (n \ mod \ w)$, тогда *выровненная подпоследовательность* $\tilde{T}_{i,n}$ определяется следующим образом:

$$\tilde{T}_{i,n} = \begin{cases} t_i, t_{i+1}, \ldots, t_{i+n-1}, \underbrace{0, 0, \ldots, 0}_{pad}, & if \ n \ mod \ w > 0 \\ t_i, t_{i+1}, \ldots, t_{i+n-1}, & otherwise. \end{cases} \tag{3.2}$$

Для обеспечения векторизации вычислений выровненные подпоследовательности временного ряда сохраняются в виде матрицы. *Матрица подпоследовательностей* $S_T^n \in \mathbb{R}^{N \times (n+pad)}$ определяется следующим образом:

$$S_T^n(i, j) := \tilde{t}_{i+j-1}. \tag{3.3}$$

Предлагаемый алгоритм для ускорения вычислений использует предварительное вычисление всех нижних границ схожести для каждой подпоследовательности ряда, в отличие от алгоритмов-аналогов, где нижние границы схожести вычисляются каскадом: следующая нижняя граница вычисляется только в том случае, если с помощью предыдущей границы схожести не выявлено, что текущая подпоследовательность не является заведомо непохожей на поисковый запрос. Строго говоря, предлагаемое предварительное вычисление представляет собой избыточные накладные расходы. Тем не менее, эти предвычисления обоснованы тем, что выполняются однократно, и могут быть реализованы с помощью циклов с фиксированным количеством повторений, которые относительно просто распараллеливаются и могут быть векторизованы компилятором.

Матрица предвычисленных нижних границ схожести подпоследователь-

ностей временного ряда определяется следующим образом. Обозначим количество нижних границ схожести, используемых в алгоритме поиска самой похожей подпоследовательности, за lb_{max} ($lb_{max} \geqslant 1$), и за LB_1, LB_2, ..., $LB_{lb_{max}}$ обозначим эти границы, перечисленные в порядке их вычисления в каскаде. Тогда $L_T^n \in \mathbb{R}^{N \times lb_{max}}$, *матрица нижних границ схожести* всех подпоследовательностей длины n временного ряда T с поисковым запросом Q, имеет следующий вид:

$$L_T^n(i,j) := LB_j(T_{i,n}, Q). \tag{3.4}$$

В предлагаемом алгоритме используется карта схожести, представляющая собой вектор-столбец, который для каждой подпоследовательности ряда хранит результат конъюнкции операций сравнения каждой из нижних границ схожести этой подпоследовательности с текущим значением локального минимума схожести (*best-so-far*). *Карта схожести* подпоследовательностей длины n временного ряда T с поисковым запросом Q, $B_T^n \in \mathbb{B}^N$, определяется следующим образом:

$$B_T^n(i) := \wedge_{j=1}^{lb_{max}} L_T^n(i,j) < bsf. \tag{3.5}$$

Вычисление карты схожести выполняется периодически для отбрасывания подпоследовательностей, заведомо непохожих на образец поиска. Указанные вычисления реализуются с помощью циклов с фиксированным количеством повторений, которые относительно просто распараллеливаются и могут быть векторизованы компилятором.

В описываемом алгоритме все подпоследовательности, которые не были отброшены как заведомо непохожие на поисковый запрос в результате применения нижних оценок схожести, помещаются в так называемую матрицу кандидатов. Алгоритм предполагает параллельную обработку матрицы кандидатов и вычисление значений меры схожести DTW между подпоследовательностями-кандидатами и поисковым запросом. Далее минимальное из вычисленных значение меры схожести DTW используется

в качестве текущего значения локального минимума схожести (*best-so-far*).

Параллельная обработка матрицы кандидатов основана на посегментном распределении строк этой матрицы между нитями. Пусть *число нитей*, используемых параллельным алгоритмом в рамках одного вычислительного узла, равно p ($p \geqslant 1$). Пусть *размер сегмента* (количество строк матрицы, обрабатываемых одной нитью), равен s ($s \ll \lceil \frac{N}{p} \rceil$). Тогда *матрица кандидатов* $C_T^n \in \mathbb{R}^{(s \cdot p) \times (n+pad)}$ определяется следующим образом:

$$C_T^n(i, \cdot) := S_T^n(k, \cdot) : B_T^n(i) = \texttt{TRUE}. \tag{3.6}$$

Несмотря на то, что вычисление матрицы кандидатов является распараллеливаемой операцией, векторизация операторов соответствующего цикла затруднена, поскольку в соответствии с определением (1.6) при вычислении меры схожести DTW имеет место зависимость по данным.

3.1.2. Реализация алгоритма

Вычислительная схема алгоритма

Предложенная реализация поиска самой похожей подпоследовательности временного ряда представлена в алг. 3.1. Алгоритм выполняется следующим образом.

В рамках подготовки к поиску алгоритм осуществляет следующие действия. С помощью функций библиотеки MPI определяется номер текущего вычислительного процесса $myrank$. В дальнейшем в рамках алгоритма каждый вычислительный процесс с номером $myrank$ обрабатывает матрицу подпоследовательностей $S^n_{T^{(myrank)}}$ фрагмента $T^{(myrank)}$ исходного временного ряда T. Далее выполняется z-нормализация поискового запроса и вычисление нижней и верхней границ его оболочки по формулам (1.8) и (1.13) соответственно. Переменная bsf инициализируется значением меры схожести DTW между поисковым запросом и случайной подпоследо-

Алг. 3.1. PBM(IN T, Q, r; OUT $bsf, bestmatch$)

```
1:  myrank ← MPI_Comm_rank()
2:  N ← |T^(myrank)| − n + 1
3:  subseq_rnd ← T^(myrank)_(random(1..N), n)
4:  CalcEnvelope(Q, r, U, L)
5:  bsf ← DTW(subseq_rnd, Q, r, ∞)
6:  L^n_T(myrank) ← CalcLowerBounds(S^n_T(myrank), Q, r)
7:  num_cand ← N
8:  myFragDone ← FALSE
9:  repeat
10:     B^n_T(myrank) ← LowerBounding(L^n_T(myrank), bsf)
11:     {C^n_T(myrank), num_cand} ← FillCandMatr(S^n_T(myrank), B^n_T(myrank))
12:     if num_cand > 0 then
13:         {bsf, bestmatch} ← CalcCandMatr(C^n_T(myrank), num_cand, r)
14:     else
15:         myFragDone ← TRUE
16:     end if
17:     {bsf, bestmatch} ← MPI_Allreduce({bsf, bestmatch},
        MPI_FLOAT_LONG, MPI_MIN)
18:     Stop ← MPI_Allreduce(myFragDone, MPI_BOOL, MPI_AND)
19: until Stop
20: return {bsf, bestmatch}
```

вательностью из данного фрагмента временного ряда. Затем выполняется предвычисление матрицы нижних границ схожести $L^n_{T^{(myrank)}}$ (см. алг. 3.2).

После этого алгоритм выполняет следующий цикл действий до тех пор, пока каждый вычислительный узел на завершит обработку своего фрагмента. Сначала выполняется вычисление матрицы карты схожести на основе предвычисленной матрицы нижних границ схожести (см. алг. 3.3). После этого заполняется матрица кандидатов, в которую попадают подпоследовательности, которые в силу значений в матрице карты схожести не являются заведомо непохожими на образец поиска (см. алг. 3.4). Отсутствие кандидатов означает, что обработка фрагмента закончена. В противном случае выполняется вычисление меры схожести DTW для каждой строки матрицы кандидатов (см. алг. 3.5). В качестве обновленного значения нижней оценки схожести bsf выбирается наименьшее значение из вычисленных,

а соответствующая подпоследовательность является наиболее похожей на образец поиска на данном шаге в текущем фрагменте ряда.

Выбор наилучшей среди всех фрагментов ряда нижней оценки схожести и соответствующей подпоследовательности осуществляется с помощью операции глобальной редукции `MPI_Allreduce` стандарта MPI, которая возвращает минимальное значение нижней оценки схожести среди всех вычислительных процессов и соответствующую подпоследовательность, копируя их в память каждого процесса.

Факт завершения обработки ряда также определяется с помощью операции глобальной редукции, в которой выполняется конъюнкция флагов завершения обработки каждым процессом своего фрагмента ряда.

Вычисление матрицы нижних границ схожести

Алг. 3.2. CalcLowerBounds(in S_T^n, Q, r, p; out L_T^n)

1: #pragma omp parallel for num_threads(p)
2: **for** i **from** 1 **to** N **do**
3: ZNormalize($S_T^n(i, \cdot)$)
4: $L_T^n(i, 1) \leftarrow LB_{Kim}FL(Q, S_T^n(i, \cdot))$
5: $L_T^n(i, 2) \leftarrow LB_{Keogh}EC(Q, S_T^n(i, \cdot))$
6: CalcEnvelope($S_T^n(i, \cdot), r, U, L$)
7: $L_T^n(i, 3) \leftarrow LB_{Keogh}EQ(S_T^n(i, \cdot), Q, U, L)$
8: **end for**
9: **return** L_T^n

Реализация вычисления матрицы нижних границ схожести представлена в алг. 3.2. Вычисление представляет собой цикл, выполняемый для каждой строки матрицы подпоследовательностей. В теле цикла выполняются операции z-нормализации и вычисления нижних границ схожести $LB_{Kim}FL$ [120], $LB_{Keogh}EC$ [90] и $LB_{Keogh}EQ$ [200] с использованием формул (1.8), (1.10), (1.12) и (1.14) соответственно. Распараллеливание цикла осуществляется с помощью директивы `#pragma` OpenMP, обеспечивающей статическое разбиение итераций цикла между нитями. Выровненные данные в матрице подпоследовательностей и отсутствие зависимостей по дан-

ным в используемых вычислительных формулах обеспечивают векторизацию соответствующих вычислений.

Вычисление карты схожести

Алг. 3.3. LowerBounding(in L_T^n, bsf, p; out B_T^n)

```
1: #pragma omp parallel for num_threads(p)
2: whoami ← omp_get_thread_num()
3: for i from pos_whoami to ⌈N / (whoami·p)⌉ do
4:     B_T^n(i) ← TRUE
5:     for j from 1 to lb_max do
6:         B_T^n(i) ← B_T^n(i) AND (L_T^n(i, j) < bsf)
7:     end for
8: end for
9: return B_T^n
```

Реализация вычисления карты схожести представлена в алг. 3.3. Алгоритм осуществляет сканирование матрицы нижних границ схожести и вычисление соответствующих значений вектора-столбца карты схожести. Полученное значение элемента карты схожести `FALSE` означает, что соответствующий элемент матрицы подпоследовательностей является заведомо непохожим на образец поиска и может быть отброшен без вычисления меры схожести DTW. В противном случае этот элемент добавляется в матрицу кандидатов для последующего вычисления меры схожести DTW.

Для распараллеливания описанных выше операций матрица нижних границ схожести разбивается на сегменты (по числу нитей, которое является параметром алгоритма поиска). В силу (3.6) матрица нижних границ схожести имеет существенно большее количество строк, чем матрица кандидатов. Соответственно, после заполнения матрицы кандидатов и обновления нижней границы схожести bsf (с помощью алг. 3.5) каждая нить должна продолжить сканирование своего сегмента вплоть до его исчерпа-

ния. Для хранения номера последнего обработанного кандидата в сегменте вводится индексный массив $Pos \in \mathbb{N}^p$, где

$$\begin{gathered} pos_i := k : p \cdot (i-1) + 1 \leqslant k \leqslant \lceil \tfrac{N}{i \cdot p} \rceil \wedge \\ \forall j, 1 \leqslant j \leqslant lb_{max}, LB_T^n(k,j) < bsf. \end{gathered} \tag{3.7}$$

Заполнение матрицы кандидатов

Алг. 3.4. FillCandMatr(in S_T^n, B_T^n, p, s; out C_T^n, num_{cand})

```
1:  num_cand ← 0
2:  for i from 1 to p do
3:      for k from 1 to s do
4:          if B_T^n(pos_i + k) =TRUE then
5:              if num_cand < s · p then
6:                  num_cand ← num_cand + 1
7:                  C_T^n(num_cand, ·) ← S_T^n(pos_i, ·)
8:                  idx_{num_cand} ← (pos_i − 1) · n + 1
9:              else
10:                 break
11:             end if
12:         end if
13:     end for
14:     if num_cand = s · p then
15:         break
16:     end if
17: end for
18: return {C_T^n, num_cand}
```

Реализация заполнения матрицы кандидатов представлена в алг. 3.4. Карта схожести разбивается на сегменты равного размера по числу используемых в поиске нитей. Алгоритм последовательно сканирует каждый сегмент карты схожести. Если элемент карты схожести имеет значение `TRUE`, то соответствующая подпоследовательность не является заведомо непохожей на образец поиска и добавляется в матрицу кандидатов для последующего вычисления меры схожести DTW. Сканирование сегмента карты

схожести начинается с соответствующего данному сегменту элемента индексного массива Pos, определенного в (3.7).

Алгоритм оперирует индексным массивом $Idx \in \mathbb{N}^{s \cdot p}$, который предназначен для хранения позиции подпоследовательности во временном ряде и определяется следующим образом:

$$idx_i := k : 1 \leqslant k \leqslant N \wedge \exists S_T^n(i, \cdot) \Leftrightarrow \exists T_{i,n} \Leftrightarrow k = (i-1) \cdot n + 1. \qquad (3.8)$$

Вычисление матрицы кандидатов

Алг. 3.5. CalcCandMatr(in $C_T^n, num_{cand}, Q, r, p$; out bsf, $bestmatch$)

```
1: #pragma omp parallel for num_threads(p) shared (bsf, Idx) private (distance)
2: for i from 1 to num_cand do
3:     distance ← DTW(C_T^n(i, ·), Q, r, bsf)
4:     #pragma omp critical
5:     if bsf > distance then
6:         bsf ← distance
7:         bestmatch ← idx_i
8:     end if
9: end for
10: return {bsf, bestmatch}
```

Реализация вычисления матрицы кандидатов представлена в алг. 3.5. Для каждой строки матрицы вычисляется значение меры схожести DTW между подпоследовательностью-кандидатом и образцом поиска. Распараллеливание цикла осуществляется с помощью директивы `#pragma` OpenMP, обеспечивающей статическое разбиение итераций цикла между нитями. Если вычисленное значение схожести меньше, чем текущее значение локального минимума bsf, то bsf заменяется на вычисленное значение. Корректное обновление переменной bsf, разделяемой нитями, реализуется с помощью критической секции.

Вычисление меры схожести *DTW*

Алг. 3.6. DTW(IN $X = (x_1, \ldots, x_n)$, $Y = (x_1, \ldots, x_n)$, r, bsf)

```
1:  cost(1, ..., n) ← bsf
2:  cost_prev(1, ..., n) ← bsf
3:  cost_prev(1) ← d(x_1, y_1)
4:  for all j ∈ max(2, i − r)..min(n, i + r) do
5:      cost_prev(j) ← cost_prev(j) + d(x_1, y_j)
6:  end for
7:  for all i ∈ 2..n do
                          ▷ Цикл без зависимостей по данным (векторизуемый)
8:      for all j ∈ max(1, i − r)..min(n, i + r) do
9:          cost(j) ← min(cost_prev(j), cost_prev(j − 1)
10:     end for
                          ▷ Цикл с зависимостями по данным (не векторизуемый)
11:     for all j ∈ max(1, i − r)..min(n, i + r) do
12:         cost(j) ← d(x_i, y_j) + min(cost_prev(j), cost_prev(j − 1))
13:     end for
14:     SWAP(cost,cost_prev)
15: end for
16: return cost_prev(n)
```

Подсчет меры схожести DTW является наиболее затратной вычислительной частью поиска похожих подпоследовательностей [91, 230], занимая до 80% общего времени поиска [258]. В силу этого модификация алгоритма вычисления меры DTW для векторизации вычислений может повысить производительность параллельного поиска похожих подпоследовательностей.

Модифицированная реализация вычисления меры DTW, допускающая векторизацию, представлена в алг. 3.6. В этой реализации цикл, имеющий зависимости по данным, разбит на два цикла, один из которых избавлен от зависимостей по данным и может быть векторизован компилятором.

3.1.3. Вычислительные эксперименты

Цели и аппаратная платформа экспериментов

Табл. 3.1. Аппаратная платформа экспериментов

Характеристика	Процессор Intel Xeon		Ускоритель Intel Xeon Phi	
Модель	X5680	E5-2630v4	SE10X (KNC)	7290 (KNL)
К-во физ. ядер	2×6	2×10	60	72
Гиперпоточность	2×	2×	4×	4×
К-во лог. ядер	24	40	240	288
Частота, ГГц	3.3	2.2	1.1	1.5
Размер VPU, бит	128	256	512	512
Пик. пр-ть, TFLOPS	0.371	0.390	1.076	3.456

Для исследования эффективности разработанного алгоритма были проведены вычислительные эксперименты. В качестве аппаратной платформы экспериментов использованы вычислительные узлы суперкомпьютеров «Торнадо ЮУрГУ» [6] и Сибирского Суперкомпьютерного Центра ИВМиМГ СО РАН [267], характеристики которых приведены в табл. 3.1.

Проведенные эксперименты включают в себя две серии: исследование эффективности алгоритма *PBM* на одном вычислительном узле кластерной системы и исследование эффективности разработанного алгоритма на кластерной системе в целом.

Исследование на одном вычислительном узле кластерной системы позволяет определить, насколько эффективно реализованы параллельные вычисления, выполняемые многоядерной процессорной системой Intel Xeon Phi. Для такого исследования использовалась упрощенная версия алг. 3.1, в которой ряд отождествлен с одним фрагментом, вызовы коммуникационной библиотеки MPI не задействуются, и цикл обработки подпоследовательностей ряда продолжается до исчерпания подпоследовательностей-кандидатов.

Во всех экспериментах размер сегмента матрицы кандидатов — количество строк матрицы кандидатов, обрабатываемых одной нитью в рам-

ках одного вычислительного узла кластерной системы (см. формулу 3.6) — имеет значение $s = 100$.

Эффективность алгоритма на одном узле кластера

Табл. 3.2. Наборы данных для экспериментов на одном узле кластера

Набор данных	Вид	$\|T\| = m$	$\|Q\| = n$
Random Walk	Синтетический	10^6	128
EPG	Реальный	$2.5 \cdot 10^5$	360

Исследование эффективности алгоритма на одном вычислительном узле кластера производилось на наборах данных, которые представлены в табл. 3.2.

Временной ряд Random Walk получен искусственно на основе модели случайных блужданий [195]. Проведение экспериментов по исследованию свойств алгоритмов поиска подпоследовательностей на базе меры схожести DTW с использованием подобных временных рядов является общепринятой практикой (см., например, работы [213, 216, 241]). Ряд Random Walk обеспечивает в среднем несколько бóльшую вычислительную нагрузку на алгоритм, чем реальный ряд, но меньшую — чем ряд, состоящий полностью из случайных чисел, за счет соответствующего количества подпоследовательностей, отбрасываемых в ходе поиска как заведомо непохожие на образец.

Временной ряд EPG (Electrical Penetration Graph, график электрического проникновения) использован в экспериментах в работе [213] и представляет собой набор сигналов, используемых энтомологами для изучения поведения цикадок (*macrosteles quadrilineatus*), которые ежегодно наносят ущерб сельскому хозяйству США более 2 млн. долларов [213].

В экспериментах исследовались производительность и масштабируемость алгоритма *PhiBestMatch*. Под производительностью понимается время работы алгоритма без учета времени загрузки данных в память и выдачи результата. Масштабируемость параллельного алгоритма означает его

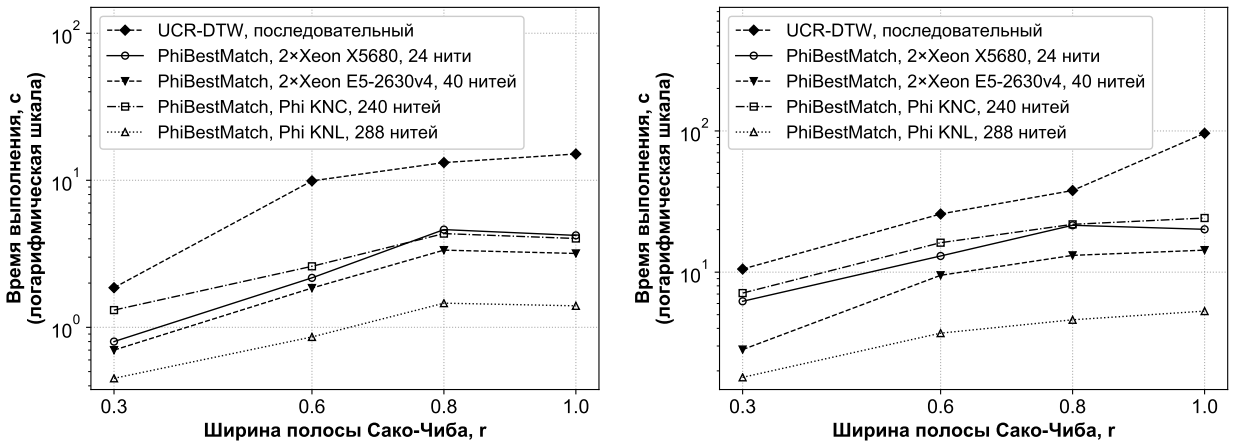
способность адекватно адаптироваться к увеличению параллельно работающих вычислительных элементов (процессов, процессоров, нитей и др.) и характеризуется ускорением и параллельной эффективностью, которые определяются следующим образом [2].

Ускорение и *параллельная эффективность* параллельного алгоритма, запускаемого на k нитях, вычисляются как $s(k) = \frac{t_1}{t_k}$ и $e(k) = \frac{s(k)}{k}$ соответственно, где t_1 и t_k — время работы алгоритма на одной и k нитях соответственно.

В экспериментах рассматривались вышеприведенные показатели в зависимости от изменения параметра r (ширина полосы Сако—Чиба), значения которого брались в долях длины поискового запроса n.

Время работы алгоритма *PBM* сравнивалось с временем работы алгоритма *UCR-DTW* [200], который является на сегодня, вероятно, самым быстрым последовательным алгоритмом поиска похожих подпоследовательностей [213].

Результаты экспериментов по исследованию производительности алгоритма представлены на рис. 3.1 и 3.2.

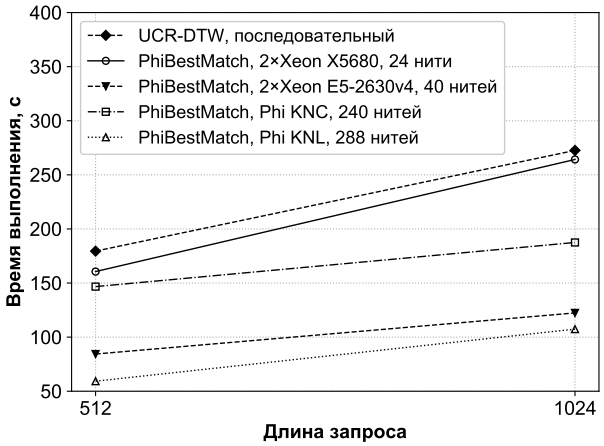


(a) Ряд Random Walk ($m = 10^6, n = 128$)

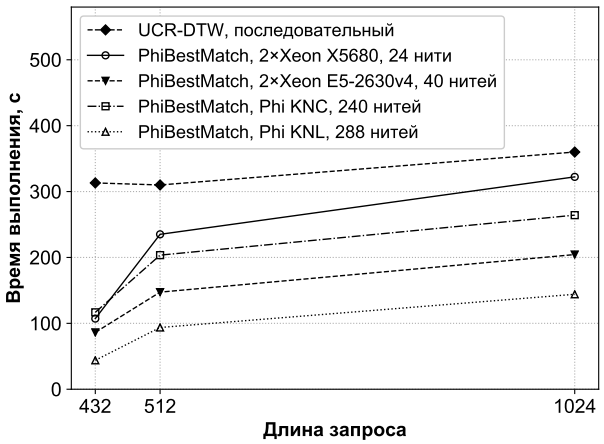
(b) Ряд EPG ($m = 2.5 \cdot 10^5, n = 360$)

Рис. 3.1. Влияние параметра r на производительность алгоритма *PhiBestMatch* на одном узле кластера

Результаты экспериментов показывают, что *PBM* работает до 5 раз быстрее, чем алгоритм *UCR-DTW*. Вместе с тем видно, на производитель-



(a) Ряд Random Walk ($m = 10^6, r = 0.8n$)



(b) Ряд ECG ($m = 10^6, r = 0.8n$)

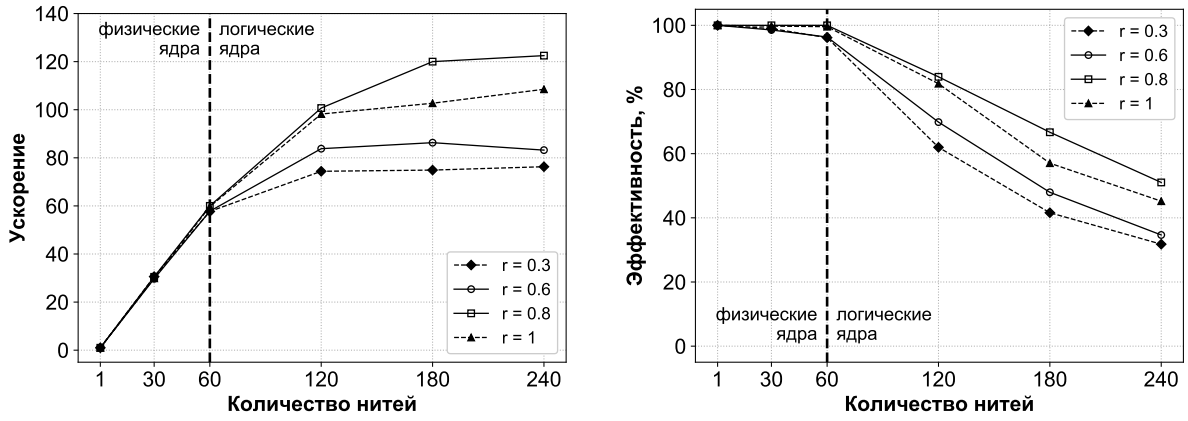
Рис. 3.2. Влияние параметра n на производительность алгоритма *PhiBestMatch* на одном узле кластера

ность алгоритма *PBM* на платформах двухпроцессорного узла Intel Xeon и многоядерной системы Intel Xeon Phi влияют следующие два параметра: ширина полосы Сако—Чиба r и длина поискового запроса n.

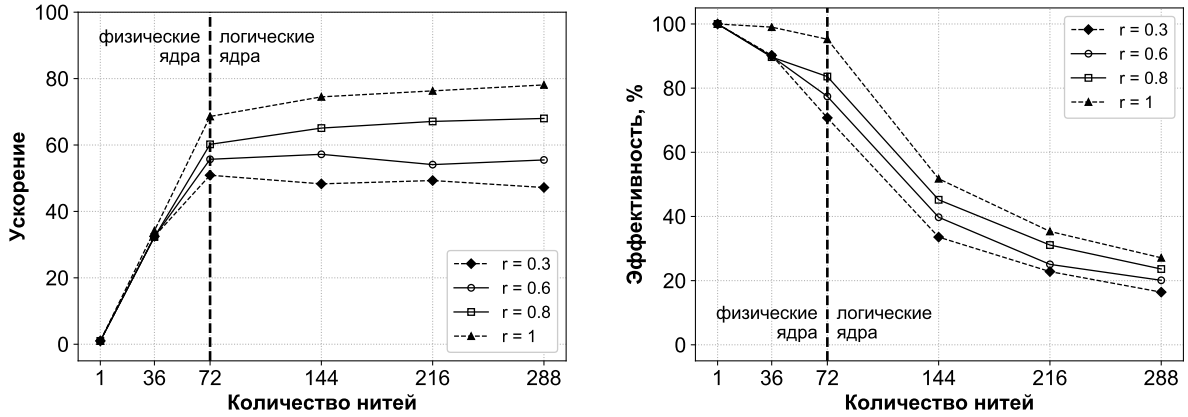
При малых значениях этих параметров (примерно $0 < r \leqslant 0.5n$ и $n < 512$) алгоритм *PBM* на платформе двухпроцессорного узла Intel Xeon работает несколько быстрее или примерно с тем же быстродействием, что и на платформе многоядерной системы Intel Xeon Phi. При бóльших значениях данных параметров ($0.5n < r \leqslant n$ и $n \geqslant 512$) алгоритм работает быстрее на платформе Intel Xeon Phi. Это значит, что заложенные в алгоритм при проектировании возможности векторизации наилучшим образом проявляются при увеличении общего объема вычислений.

Поисковые запросы с длиной $n \geqslant 512$, равно как и значение параметра $r = 1$ требуются на практике в ряде приложений, требующих при определении схожести подпоследовательностей как можно более высокую точность, например, в медицине при исследовании ЭКГ [152,236], в энтомологии [36], в астрономии [203] и др.

Результаты экспериментов по исследованию масштабируемости алгоритма представлены на рис. 3.3 и 3.4. Результаты экспериментов показывают, что *PBM* демонстрирует близкое к линейному ускорение и парал-



(a) Платформа Intel Xeon Phi Knights Corner

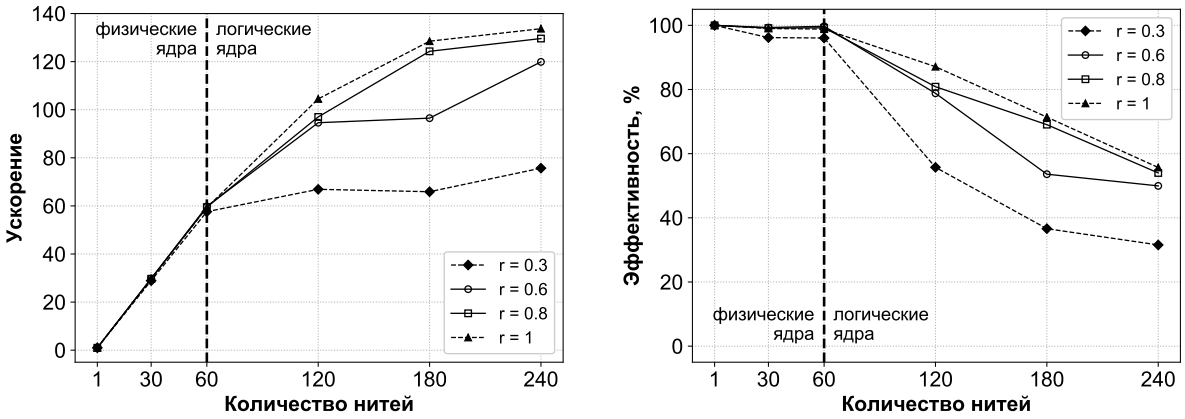


(b) Платформа Intel Xeon Phi Knights Landing

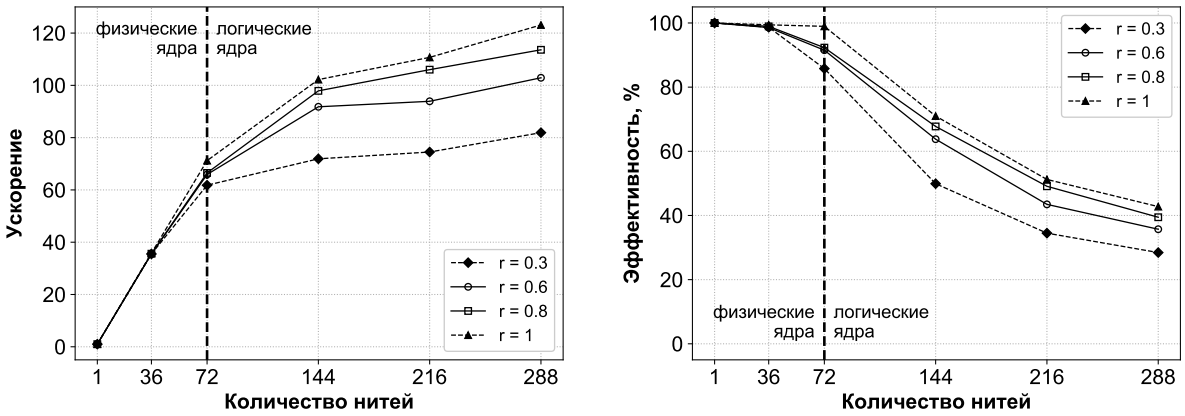
Рис. 3.3. Масштабируемость алгоритма *PBM* на одном узле кластера при обработке синтетических данных (ряд Random Walk, $m = 10^6, n = 128$)

лельную эффективность, близкую к 100%, если количество нитей, на которых запущен алгоритм, совпадает с количеством физических ядер системы Intel Xeon Phi.

При увеличении количества нитей, запускаемых на одном физическом ядре системы, ускорение становится сублинейным, равно как наблюдается и падение параллельной эффективности. При этом наилучшие показатели ускорения и параллельной эффективности ожидаемо наблюдаются при значениях параметра r 0.8 и 1 от длины поискового запроса n, обеспечивающих алгоритму наибольшую вычислительную нагрузку. Например, при задействовании 240 нитей при $r = 0.8n$ обработка ряда Random Walk



(a) Платформа Intel Xeon Phi Knights Corner



(b) Платформа Intel Xeon Phi Knights Landing

Рис. 3.4. Масштабируемость алгоритма *PhiBestMatch* на одном узле кластера при обработке реальных данных (ряд EPG, $m = 2.5 \cdot 10^5, n = 360$)

выполняется с ускорением 120 и параллельной эффективностью 50%; при $r = n$ обработка ряда EPG — с ускорением 130 и параллельной эффективностью 52%.

Полученные результаты позволяют сделать заключение о хорошей масштабируемости разработанного алгоритма и эффективном использовании им возможностей векторизации вычислений на многоядерном ускорителе Intel Xeon Phi. Указанные свойства проявляются при значениях параметров r (ширина полосы Сако—Чиба) и n (длина поискового запроса), обеспечивающих алгоритму наибольшую вычислительную нагрузку: $0.8n \leqslant r \leqslant n$ и $n \geqslant 512$ соответственно.

Эффективность алгоритма на кластерной системе

В исследовании эффективности алгоритма на кластерной системе в целом было задействовано от 16 до 128 вычислительных узлов суперкомпьютера «Торнадо ЮУрГУ» (см. табл. 3.1). Исследование производилось на наборах данных, которые представлены в табл. 3.3.

Табл. 3.3. Наборы данных для экспериментов на кластерной системе

Набор данных	Вид	$\|T\| = m$	$\|Q\| = n$
Random Walk	Синтетический	$12.8 \cdot 10^7$	128, 512, 1024
ECG	Реальный	$12.8 \cdot 10^7$	432, 512, 1024

В экспериментах исследовалось *ускорение масштабируемости (scaled speedup)* параллельного алгоритма, которое определяется как ускорение, демонстрируемое алгоритмом при линейном увеличении объема данных и количества используемых вычислительных элементов [130] и вычисляется следующим образом: $s_{scaled} = \frac{p \cdot m}{t_{p(p \cdot m)}}$, где p — количество задействованных вычислительных узлов, m — объем исходных данных, $t_{p(p \cdot m)}$ — время выполнения алгоритма на p узлах при обработке исходных данных, имеющих объем $p \cdot m$.

При этом значение параметра r было зафиксировано как $r = n$ и варьировалась длина поискового запроса.

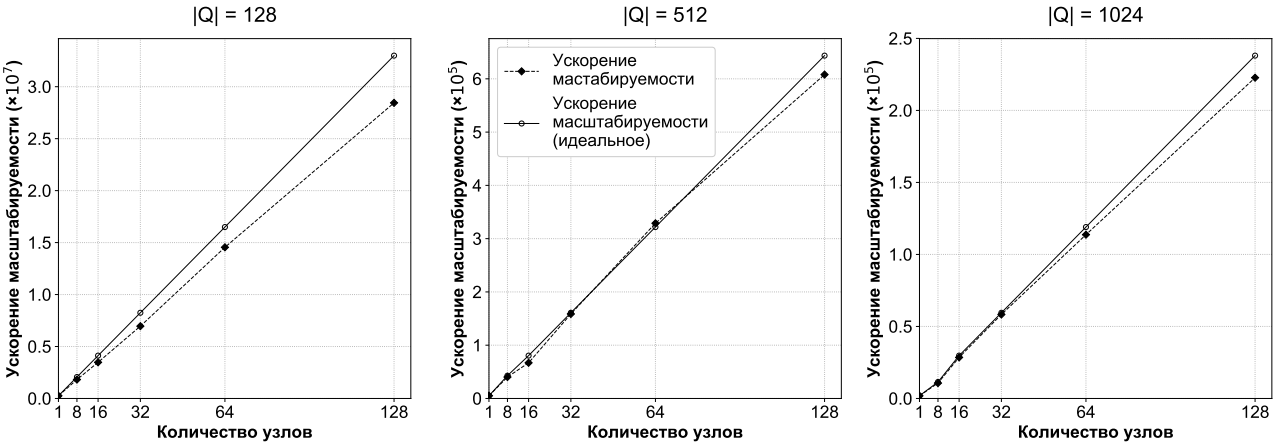


Рис. 3.5. Ускорение масштабируемости алгоритма *PBM* на кластерной системе при обработке синтетических данных

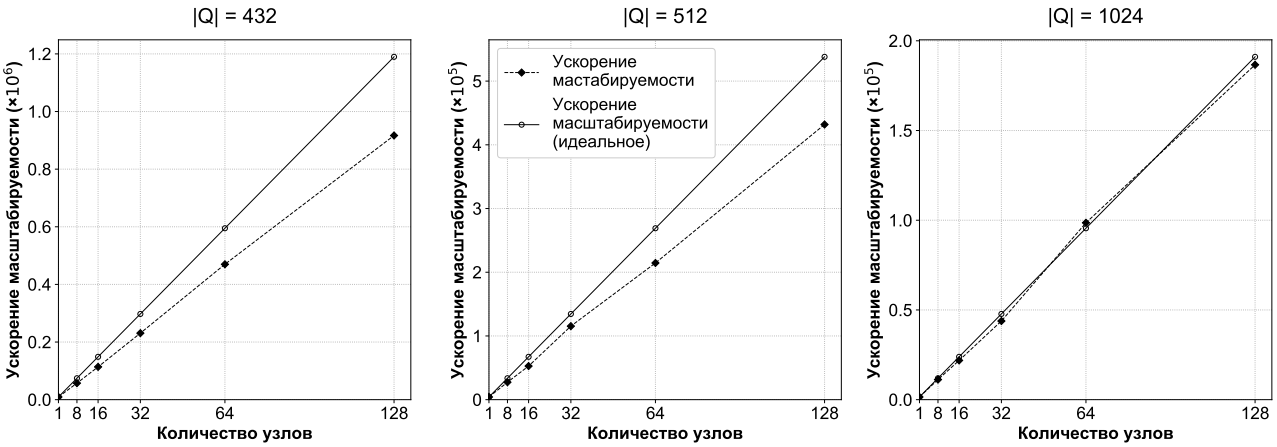


Рис. 3.6. Ускорение масштабируемости алгоритма *PhiBestMatch* на кластерной системе при обработке реальных данных

Результаты экспериментов по исследованию ускорения масштабируемости алгоритма представлены на рис. 3.5 и 3.6. Результаты экспериментов показывают, что *PBM* демонстрирует ускорение масштабируемости, близкое к линейному, как для синтетических, так и для реальных данных. При этом поиск подпоследовательности большей длины показывает более высокое ускорение масштабируемости, поскольку это обеспечивает больший объем вычислений в рамках одного вычислительного узла кластерной системы (см. результаты экспериментов на рис. 3.3 и 3.4).

Полученные результаты позволяют сделать заключение о хорошей масштабируемости разработанного алгоритма при работе на кластерной системе с вычислительными узлами на базе многоядерных процессоров Intel Xeon Phi. Указанные свойства проявляются при значениях параметров r (ширина полосы Сако—Чиба) и n (длина поискового запроса), обеспечивающих алгоритму наибольшую вычислительную нагрузку: $0.8n \leqslant r \leqslant n$ и $n \geqslant 512$ соответственно.

Сравнение с аналогами

Производительность алгоритма *PBM* сравнивалась с быстродействием следующих параллельных алгоритмов, рассмотренных выше в разделе 1.3.4.

Параллельный алгоритм поиска похожих подпоследовательностей для GPU предложен Сартом (Sart) и др. в работе [213], в которой приведены результаты экспериментов, исследующих эффективность данного алгоритма на вычислительной системе NVIDIA Tesla C1060, содержащей 240 процессорных ядер с тактовой частотой 1.3 ГГц. В качестве входных данных использовался реальный ряд EPG с длиной $m = 1.5 \cdot 10^6$ и запрос длиной $n = 360$.

Параллельный алгоритм поиска похожих подпоследовательностей для кластерной вычислительной системы с многоядерными процессорами предложен Шабибом (Shabib) и др. в работе [216], в которой приведены результаты экспериментов, исследующих эффективность данного алгоритма на кластерной системе с вычислительными узлами на базе 4-ядерных процессоров Intel Xeon E3-1200 с тактовой частотой 3.1 ГГц. В экспериментах использовался синтетический ряд Random Walk ($m = 2.2 \cdot 10^8$), длина поискового запроса $n = 128$.

Параллельный алгоритм поиска похожих подпоследовательностей для гибридной вычислительной системы, состоящей из GPU и AMD APU (объединение центрального процессора с графическим на одном кристалле) предложен Хуангом (Huang) и др. в работе [106], в которой приведены результаты экспериментов, исследующих эффективность данного алгоритма на вычислительной системе, содержащей AMD Radeon HD 7970 и AMD APU A10-5800K с тактовой частотой 3.8 ГГц. В качестве входных данных использовался синтетический ряд Random Walk с длиной $m = 10^6$ и запрос длиной $n = 128$.

Табл. 3.4. Сравнение алгоритма *PBM* с аналогами

Временной ряд		Аналог			***PhiBestMatch***	
m	n	Алгоритм	Платформа	Время, с	Время, с	Платформа
$2.2 \cdot 10^8$	128	*Shabib* [216]	6 CPU (4 ядра 3.1 ГГц)	32.0	**24.1**	6 Phi KNC (6× 60 ядер 1.1 ГГц)
$1.5 \cdot 10^6$	360	*Sart* [213]	Tesla C1060 (240 ядер 1.3 ГГц)	80.4	**36.9**	Phi KNC (20 ядер 1.1 ГГц)
10^6	128	*Huang* [106]	AMD Radeon HD 7970+AMD APU A10-5800K (3.8 ГГц)	11.2	**4.3**	Phi KNC (61 ядро 1.1 ГГц)

При сравнении алгоритм *PBM* запускался на конфигурациях кластера «Торнадо ЮУрГУ» (см. табл. 3.1), которые обеспечивают примерно равную пиковую производительность, что и у аппаратной платформы алгоритма-

конкурента. В экспериментах использовались те же наборы данных, что и в упомянутых выше работах. Результаты сравнения производительности алгоритмов приведены в табл. 3.4. Можно видеть, что алгоритм *PBM* опережает конкурентов.

3.2. Параллельный алгоритм поиска диссонансов *MDD*

3.2.1. Проектирование алгоритма

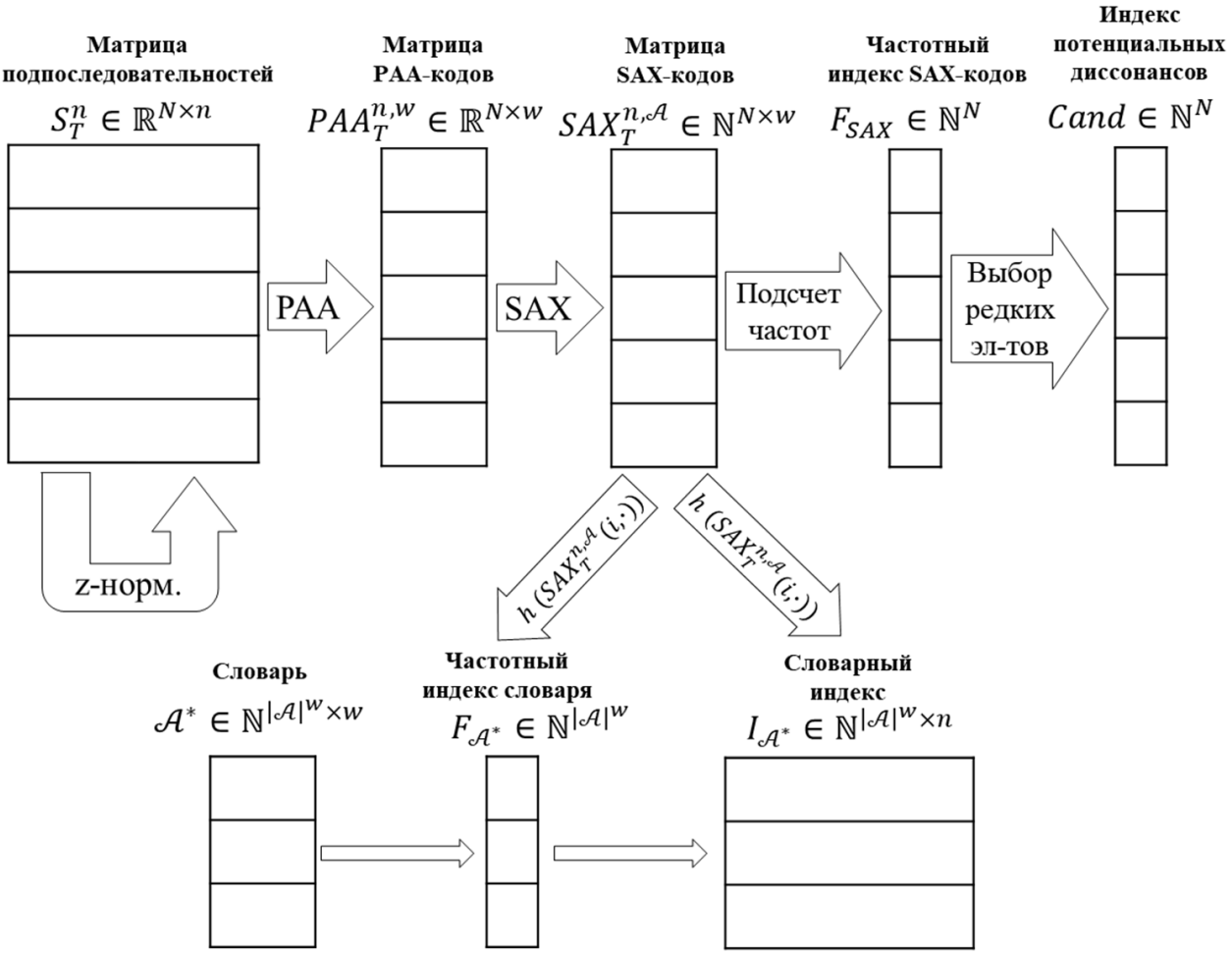


Рис. 3.7. Структуры данных алгоритма *MDD*

Предлагаемый алгоритм использует следующие основные структуры для хранения данных в оперативной памяти (см. рис. 3.7).

Матрица подпоследовательностей $S_T^n \in \mathbb{R}^{N\times n}$ определяется в соответствии с 3.3 и хранит все подпоследовательности исходного временного ряда, выровненные в соответствии с 3.2.

Матрица $PAA_T^{n,w} \in \mathbb{R}^{N\times w}$ предназначена для хранения *PAA-кодов* подпоследовательностей из S_T^n, полученных в соответствии с формулой (1.15).

Матрица SAX-кодов, $SAX_T^{n,\mathcal{A}} \in \mathbb{N}^{N\times w}$, хранит символьные подпоследовательности в алфавите $\mathcal{A}$, полученные из PAA-кодов в соответствии с формулой (1.16).

Индекс потенциальных диссонансов представляет собой упорядоченный по возрастанию массив $Cand \in \mathbb{N}^N$ с номерами тех подпоследовательностей в матрице подпоследовательностей S_T^n, чьи SAX-коды наиболее редко встречаются в матрице $SAX_T^{n,\mathcal{A}}$:

$$\begin{gathered} Cand(i) = k \Leftrightarrow F_{SAX}(k) = \min_{1\leqslant j\leqslant N} F_{SAX}(j) \wedge \\ \forall i < j \;\; Cand(i) < Cand(j). \end{gathered} \tag{3.9}$$

Здесь $F_{SAX} \in \mathbb{N}^N$ представляет собой *частотный индекс SAX-кодов* — массив, используемый для хранения частот слов из матрицы SAX-кодов:

$$F_{SAX}(i) = k \Leftrightarrow |\{j \mid SAX_T^{n,\mathcal{A}}(j,\cdot) = SAX_T^{n,\mathcal{A}}(i,\cdot)\}| = k. \tag{3.10}$$

Индекс потенциальных диссонансов создается на стадии подготовки данных и далее на стадии поиска задает порядок перебора подпоследовательностей, которые могут являться диссонансами.

Словарь — матрица $W_{\mathcal{A}} \in \mathbb{N}^{dict_{size}\times w}$, предназначенная для хранения всех возможных слов длины w, составленных из символов алфавита $\mathcal{A}$. Словарь заполняется в соответствии с известным алгоритмом, описанным у Д. Кнута [122]. Словарь организуется таким образом, чтобы все символы каждого слова (элементы в одной строке матрицы) и все слова (строки матрицы) были упорядочены по возрастанию. Мощность словаря $dict_{size}$

вычисляется как число размещений символов алфавита $\mathcal{A}$ по w символов с повторениями:

$$dict_{size} := \bar{A}^{w}_{|\mathcal{A}|} = |\mathcal{A}|^{w}. \tag{3.11}$$

В качестве символов алфавита $\mathcal{A}$ будем рассматривать упорядоченный набор натуральных чисел $1, 2, \ldots, |\mathcal{A}|$. Для доступа к элементам словаря вводится хэш-функция $h : \mathbb{N}^{w} \to \{1, 2, \ldots, dict_{size}\}$, определяемая следующим образом:

$$h(a_1, a_2, \ldots, a_w) := \sum_{j=1}^{w+1} a_j \cdot w^{w-j-1}. \tag{3.12}$$

Значения длины слова и мощности алфавита, $w = 4$ и $|\mathcal{A}| = 4$, соответственно, как показывают эксперименты [116,117], хорошо подходят при поиске диссонансов во временных рядах из различных предметных областей. В силу этого словарь ($4^4 \times 4 = 256$ элементов) может быть размещен в оперативной памяти.

Словарный индекс предназначен для хранения индексов слов алфавита $\mathcal{A}$ в матрице SAX-кодов и представляет собой матрицу $I_W \in \mathbb{N}^{dict_{size} \times N}$:

$$I_W(i, j) = k \Leftrightarrow W_{\mathcal{A}}(i, \cdot) = SAX_T^{n,\mathcal{A}}(k, \cdot). \tag{3.13}$$

Словарный индекс создается на стадии подготовки данных и далее на стадии поиска используется для задания порядка перебора тех подпоследовательностей, которые не пересекаются с данной.

Частотный индекс словаря $F_W \in \mathbb{N}^{|\mathcal{A}|^w}$ для каждого слова в словаре хранит частоту появления этого слова в матрице SAX-кодов:

$$F_W(i) = k \Leftrightarrow k = |\{j \mid W_{\mathcal{A}}(i, \cdot) = SAX_T^{n,\mathcal{A}}(j, \cdot)\}|. \tag{3.14}$$

Алг. 3.7. MDD(IN T, n, w; OUT $\{pos_{bsf}, dist_{bsf}\}$)

▷ Стадия подготовки

1: $S_T^n \leftarrow$ ZNORMALIZE(S_T^n)
2: $W_{\mathcal{A}} \leftarrow$ MAKEWORDMATRIX($\mathcal{A}, w$)
3: $PAA_T^{n,w} \leftarrow$ PAA(S_T^n, w)
4: $SAX_T^{n,\mathcal{A}} \leftarrow$ SAX($PAA_T^{n,w}, \mathcal{A}$)
5: $Cand \leftarrow$ MAKECANDIDATES($SAX_T^{n,\mathcal{A}}$)
6: $I_W \leftarrow$ MAKEINDEXWORD($SAX_T^{n,\mathcal{A}}$)

▷ Стадия поиска

7: $pos_{bsf} \leftarrow 0$; $dist_{bsf} \leftarrow 0$
8: $\{pos_{bsf}, dist_{bsf}\} \leftarrow$ POTENTIALDISCORD($I_W, Cand, pos_{bsf}, dist_{bsf}$)
9: $\{pos_{bsf}, dist_{bsf}\} \leftarrow$ REFINEDISCORD($I_W, Cand, pos_{bsf}, dist_{bsf}$)
10: **return** $\{pos_{bsf}, dist_{bsf}\}$

3.2.2. Реализация алгоритма

Реализация параллельного поиска диссонансов временного ряда представлена в алг. 3.7. На стадии подготовки алгоритм выполняет построение структур данных, рассмотренных выше в разделе 3.2.1. Далее на стадии поиска алгоритм осуществляет нахождение диссонанса с помощью указанных структур.

На *стадии подготовки* алгоритм действует следующим образом. Сначала осуществляется построение матрицы подпоследовательностей исходного временного ряда. Далее выполняется генерация матрицы слов, строками которой будут все возможные слова длины w, составленные из символов алфавита $\mathcal{A}$.

После этого происходит формирование PAA-кода каждой подпоследовательности в матрице подпоследовательностей в соответствии с формулой (1.15). Соответствующий цикл обработки строк матрицы подпоследовательностей распараллеливается с помощью стандартной директивы компилятора `#pragma omp parallel for` OpenMP, обеспечивающей статическое разбиение итераций цикла между нитями.

Затем выполняется формирование SAX-кода подпоследовательности для каждого PAA-кода, полученного на предыдущем шаге. Параллельная обра-

ботка строк матрицы PAA-представлений, так же как и на предыдущем шаге, обеспечивается использованием директивы компилятора `#pragma omp parallel for`.

Далее на основе полученной матрицы SAX-кодов подпоследовательностей осуществляется формирование индекса потенциальных диссонансов $Cand$ (см. формулу 3.9). Для этого вычисляется частотный индекс F_{SAX} (см. формулу 3.10) и в $Cand$ добавляются номера строк матрицы SAX-кодов, которые имеют минимальную частоту (встречаются в матрице SAX-кодов наиболее редко). В данном шаге нахождение минимального элемента массива F_{SAX} также распараллеливается с помощью директивы компилятора `#pragma omp parallel for` с использованием параметра `reduction`, который обеспечивает свертку операции поиска минимума.

Следующим шагом алгоритма является построение словарного индекса I_W, выполняемое следующим образом. Осуществляется сканирование матрицы SAX-кодов. Для каждой строки этой матрицы вычисляется хэш-функция, дающая номер соответствующего элемента в матрице слов (см. формулу 3.12). Далее в строку словарного индекса с номером, совпадающим со значением хэш-функции, записывается номер слова в матрице SAX-кодов. В данном шаге цикл обработки строк матрицы SAX-кодов распараллеливается с помощью стандартной директивы компилятора `#pragma omp parallel for` OpenMP, обеспечивающей статическое разбиение итераций цикла между нитями.

Стадия поиска состоит из двух следующих шагов. На первом шаге (см. алг. 3.8) выполняется поиск диссонансов среди подпоследовательностей, входящих в индекс потенциальных диссонансов, построенный на предыдущей стадии, в порядке, задаваемом указанным индексом. Результатом данного шага позиция предполагаемого диссонанса в исходном временном ряде и расстояние (степень схожести) предполагаемого диссонанса с его ближайшим соседом. На втором шаге (см. алг. 3.9) выполняется уточнение найденной ранее позиции предполагаемого диссонанса среди тех подпоследовательностей, которые не входят в индекс потенциальных диссонансов.

Алг. 3.8. PotentialDiscord(in T, n; out $\{pos_{bsf}, dist_{bsf}\}$)

```
1: for all C_i ∈ Cand do
2:     dist_min ← ∞
3:     #pragma omp parallel for schedule(dynamic)
4:     for all C_j ∈ I_W(SAX_T^{n,A}(C_i)) and |i − j| ⩾ n do
5:         dist ← ED^2(C_i, C_j)
6:         if dist < dist_bsf then
7:             break
8:         end if
9:         if dist < dist_min then
10:            dist_min ← dist
11:        end if
12:    end for
13:    #pragma omp parallel for schedule(dynamic)
14:    for all C_j ∉ I_W(SAX_T^{n,A}(C_i)) and |i − j| ⩾ n do
15:        dist ← ED^2(C_i, C_j)
16:        if dist < dist_bsf then
17:            break
18:        end if
19:    end for
20:    if dist_min > dist_bsf then
21:        dist_bsf ← dist_min
22:        pos_bsf ← i
23:    end if
24: end for
25: return {pos_bsf, dist_bsf}
```

Шаг поиска диссонансов (см. алг. 3.8) заключается в следующем. Для каждой подпоследовательности индекса потенциальных диссонансов $Cand$ выполняется нахождение ближайших соседей среди всех подпоследовательностей временного ряда, в порядке, задаваемом этим индексом, по формуле (3.9). Из найденных ближайших соседей выбирается сосед, имеющий наименьшую степень схожести с другими подпоследовательностями (в смысле Евклидовой метрики). При этом в вычислениях используется квадрат Евклидова расстояния (вместо собственно значения расстояния) для ускорения вычислений.

Поиск ближайшего соседа подпоследовательности осуществляется сле-

дующим образом. Сначала в качестве соседей рассматриваются подпоследовательности строки словарного индекса I_W, соответствующей данной обрабатываемой подпоследовательности, которые имеют одинаковый с ней SAX-код. Перебор соседей осуществляется в порядке, задаваемом словарным индексом.

Затем рассматриваются подпоследовательности, не вошедшие в словарный индекс, в порядке, задаваемом матрицей подпоследовательностей. При этом расстояние вычисляется только для подпоследовательностей, которые не являются пересекающимися. Если найденное расстояние между подпоследовательностями меньше найденного на предыдущих итерациях цикла расстояния до ближайшего соседа $dist_{bsf}$, то текущая подпоследовательность исключается из рассмотрения. Такое отбрасывание позволяет сократить объем вычислений и возможно, поскольку найдена подпоследовательность, степень схожести которой с текущей подпоследовательностью больше, чем между двумя другими подпоследовательностями, рассмотренными на предыдущей итерации цикла.

После того, как найден ближайший сосед для данной подпоследовательности, производится сравнение его степени схожести со степенью схожести найденного на предыдущих итерациях самого необычного из ближайших соседей. Полученный самый необычный ближайший сосед является предположительным диссонансом, и алгоритм выдает положение диссонанса во временном ряде.

На втором шаге (см. алг. 3.9) выполняется уточнение найденной ранее позиции предполагаемого диссонанса среди тех подпоследовательностей, которые не входят в индекс потенциальных диссонансов. Поиск ближайших соседей каждой из перебираемых подпоследовательностей выполняется аналогично поиску на первом шаге (нахождение предположительного диссонанса, см. алг. 3.8). Среди всех найденных ближайших соседей осуществляется поиск такого соседа, который имеет наименьшую степень схожести со всеми подпоследовательностями. Результатом данного шага является уточненное положение диссонанса временного ряда и расстояние

Алг. 3.9. RefineDiscord(in T, n; out $\{pos_{bsf}, dist_{bsf}\}$)

```
1: #pragma omp parallel for schedule(dynamic)
2: for all C_i ∉ Cand do
3:     dist_min ← ∞
4:     for all C_j ∈ I_W(SAX_T^{n,A}(C_i)) and |i − j| ⩾ n do
5:         dist ← ED^2(C_i, C_j)
6:         if dist < dist_bsf then
7:             break
8:         end if
9:         if dist < dist_min then
10:            dist_min ← dist
11:        end if
12:    end for
13:    for all C_j ∉ I_W(SAX_T^{n,A}(C_i)) and |i − j| ⩾ n do
14:        dist ← ED^2(C_i, C_j)
15:        if dist < dist_bsf then
16:            break
17:        end if
18:        if dist < dist_min then
19:            dist_min ← dist
20:        end if
21:    end for
22:    if dist_min > dist_bsf then
23:        dist_bsf ← dist_min
24:        pos_bsf ← i
25:    end if
26: end for
27: return {pos_bsf, √dist_bsf}
```

(степень схожести) от данного диссонанса до его ближайшим соседом.

В обоих описанных выше шагах стадии поиска перебор ближайших соседей распараллеливается с помощью стандартной директивы компилятора OpenMP `#pragma omp parallel for`. Поскольку отбрасывание неперспективных подпоследовательностей приводит неравномерной вычислительной загрузке нитей, в указанной директиве используется параметр `schedule (dynamic)`, обеспечивающий динамическое распределение итераций цикла между нитями. Операторы тела цикла, выполняющего вычисление квад-

ратов Евклидовых расстояний, векторизуются компилятором.

3.2.3. Вычислительные эксперименты

Цели, аппаратная платформа и наборы данных экспериментов

Табл. 3.5. Аппаратная платформа экспериментов

Характеристика	Процессор Intel Xeon		Ускоритель Intel Xeon Phi	
Модель	E5-2630v4	E5-2697v4	SE10X (KNC)	7290 (KNL)
К-во физ. ядер	2×10	2×16	60	72
Гиперпоточность	2×	2×	4×	4×
К-во лог. ядер	40	64	240	288
Частота, ГГц	2.2	2.6	1.1	1.5
Размер VPU, бит	256	256	512	512
Пик. пр-ть, TFLOPS	0.390	0.600	1.076	3.456

Для исследования эффективности разработанного алгоритма были проведены вычислительные эксперименты. В качестве аппаратной платформы экспериментов использованы вычислительные узлы суперкомпьютеров «Торнадо ЮУрГУ» [6] и Сибирского Суперкомпьютерного Центра ИВМиМГ СО РАН [267], характеристики которых приведены в табл. 3.5.

Табл. 3.6. Наборы данных для экспериментов

Набор данных	Вид	$\|T\| = m$	n
SCD-1M	Синтетический	10^6	64, 128, 512, 1024, 2048
SCD-10M	Синтетический	10^7	64, 128, 512, 1024, 2048

Исследование производилось на наборах данных, которые представлены в табл. 3.6. Указанные наборы использовались в экспериментах по исследованию эффективности работы алгоритма поиска диссонансов, предложенного в работе [107].

В экспериментах исследовались производительность и масштабируемость алгоритма *MDD*. Под производительностью понимается время работы алгоритма без учета времени загрузки данных в память и выдачи результата. Масштабируемость параллельного алгоритма означает его

способность адекватно адаптироваться к увеличению параллельно работающих вычислительных элементов (процессов, процессоров, нитей и др.) и характеризуется ускорением и параллельной эффективностью, которые определяются следующим образом [2].

Ускорение и *параллельная эффективность* параллельного алгоритма, запускаемого на k нитях, вычисляются как $s(k) = \frac{t_1}{t_k}$ и $e(k) = \frac{s(k)}{k}$ соответственно, где t_1 и t_k — время работы алгоритма на одной и k нитях соответственно.

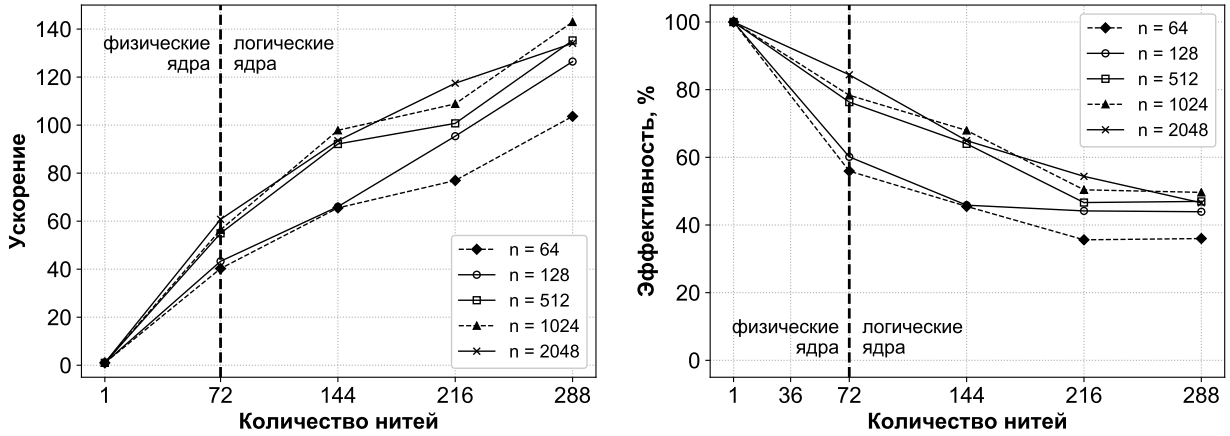
В экспериментах рассматривались вышеприведенные показатели в зависимости от изменения параметра n (длина искомого диссонанса).

Быстродействие алгоритма *MDD* сравнивалось с быстродействием параллельных алгоритмов *PDD* [107], *DDD* [248] и *MR-DADD* [253], рассмотренных ранее в разделе 1.3.4. В экспериментах использовались те же наборы данных, на которых исследовалась эффективность алгоритмов-конкурентов в вышеуказанных работах. Данные о быстродействии конкурентов взяты из соответствующих статей.

Запуск алгоритма *MDD* осуществлялся на ускорителе Intel Xeon Phi KNL (см. табл. 3.5). Для обеспечения справедливых условий сравнения количество вычислительных ядер, на котором запускался алгоритм *MDD*, уменьшалось таким образом, чтобы обеспечить примерное равенство пиковой производительности текущей аппаратной платформы с аппаратной платформой алгоритма-конкурента, упомянутой в соответствующей статье.

Результаты экспериментов

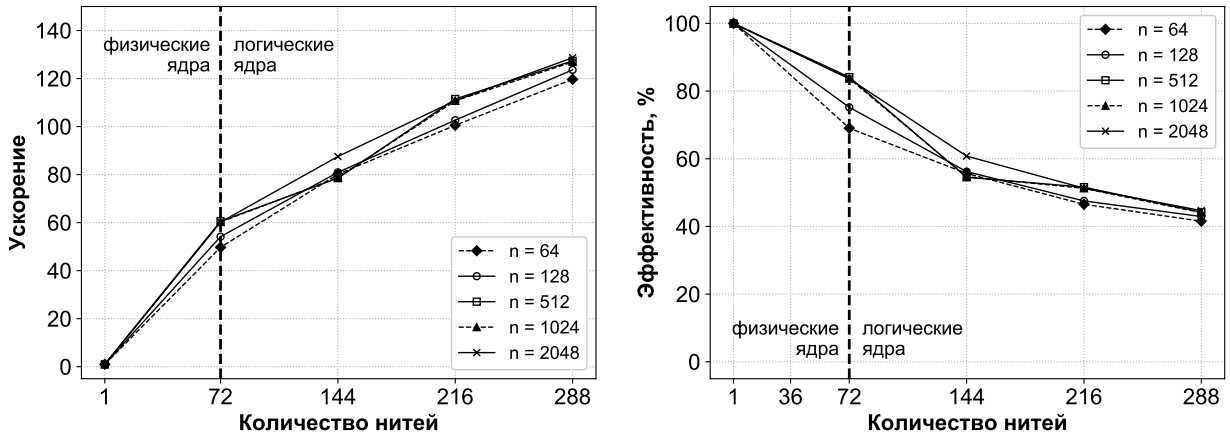
Результаты экспериментов ***по исследованию масштабируемости*** алгоритма на системе Intel Xeon Phi Knights Landing представлены на рис. 3.8 и 3.9. Результаты экспериментов показывают, что *MDD* демонстрирует ускорение от 40 до 60 и параллельную эффективность от 50 до 90% (в зависимости от длины искомого диссонанса), если количество ни-



(a) Ускорение (b) Параллельная эффективность

Рис. 3.8. Масштабируемость *MDD* при обработке ряда SCD-1M

тей, на которых запущен алгоритм, совпадает с количеством физических ядер системы Intel Xeon Phi.



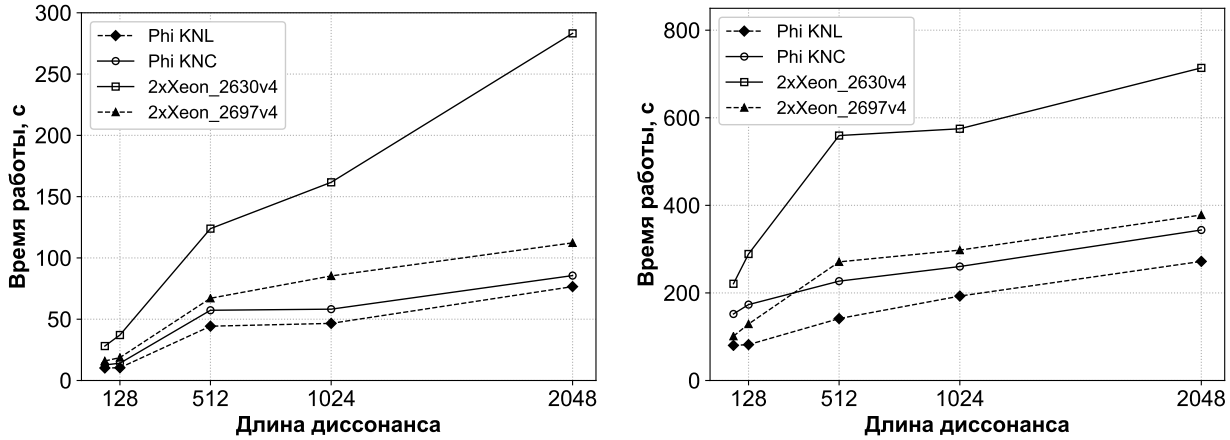
(a) Ускорение (b) Параллельная эффективность

Рис. 3.9. Масштабируемость *MDD* при обработке ряда SCD-10M

При увеличении количества нитей, запускаемых на одном физическом ядре системы, ускорение является сублинейным, равно как наблюдается и падение параллельной эффективности. При этом наилучшие показатели ускорения и параллельной эффективности ожидаемо наблюдаются при бóльших значениях параметров m и n (длина исходного временного ряда и длина искомого диссонанса соответственно), обеспечивающих алгорит-

му наибольшую вычислительную нагрузку. Например, при обработке ряда длиной 1 млн. 288 нитями наблюдаются ускорение 100 и параллельная эффективность 40% при $n = 64$, тогда как при $n = 2048$ эти показатели равны 140 и 50% соответственно.

Полученные результаты позволяют сделать заключение о хорошей масштабируемости разработанного алгоритма и эффективном использовании им возможностей векторизации вычислений на многоядерной системе Intel Xeon Phi.



(a) Набор данных SCD-1M

(b) Набор данных SCD-10M

Рис. 3.10. Производительность алгоритма *MDD*

Результаты экспериментов ***по исследованию производительности*** алгоритма представлены в рис. 3.10. Результаты экспериментов показывают, что алгоритм *MDD* работает, как правило, быстрее на платформе многопроцессорной системы Intel Xeon Phi, чем на платформе двухпроцессорного узла Intel Xeon. Данный факт всегда имеет место для более производительного устройства поколения Knights Landing; менее производительное устройство поколения Knights Corner уступает 64-ядерному двухпроцессорному узлу Intel Xeon при поиске диссонансов длины $n \leqslant 128$ в ряде из 10 млн. точек. В целом на платформе Intel Xeon Phi KNL алгоритм опережает себя на платформе 64-ядерного двухпроцессорного узла Intel Xeon в 1.5–2 раза.

Табл. 3.7. Сравнение алгоритма *MDD* с аналогами

Условия эксперимента				Быстродействие, с	
Аналог		Длина ряда	К-во ядер (нитей) Intel MIC	***MDD***	Аналог
Алгоритм	Платформа				
MR-DADD [253]	8 CPU 3.0 ГГц	10^6	8 (32)	**101.6**	240
DDD [248]	4 CPU 2.13 ГГц	10^7	4 (16)	**1 745.3**	5 382
PDD [107]	10 CPU 1.2 ГГц	10^7	10 (40)	**833.3**	399 600

Результаты экспериментов ***по сравнению алгоритма с аналогами*** представлены в табл. 3.7. Алгоритм *MDD* значительно проигрывает разработанному алгоритму, поскольку массово использует обмены данными между узлами кластера, что существенно снижает производительность. Алгоритмы *DDD* и *MR-DADD*, хотя и не используют массовые обмены, также уступают по быстродействию разработанному алгоритму примерно в три и два раза соответственно. В качестве основной причины следует указать накладные расходы на дисковый ввод-вывод, которые обусловлены природой алгоритма, использующего хранение данных на диске, а не в оперативной памяти: как показывают эксперименты [253], такие расходы могут составить более половины общего времени работы алгоритма. Кроме того, выполняемая алгоритмом *MDD* параллельная обработка матричных структур данных в оперативной памяти эффективно векторизуется компилятором. Результаты экспериментов позволяют заключить, что алгоритм *MDD* опережает аналоги.

3.3. Выводы по главе 3

В данной главе рассмотрены параллельные методы интеллектуального анализа временных рядов на платформе современных многопроцессорных многоядерных вычислительных систем. Рассмотрены две следующие задачи: поиск похожих подпоследовательностей во временном ряде и поиск диссонансов во временном ряде.

Предложен оригинальный параллельный алгоритм *PBM (Parallel Best Match)* поиска похожих подпоследовательностей во временном ряде для

кластерных систем с узлами на базе многоядерных ускорителей. Алгоритм предполагает двухуровневое распараллеливание вычислений: на уровне всех узлов кластера и внутри одного узла кластера. Временной ряд разбивается на фрагменты по узлам кластерной системы. Для предотвращения потери результирующих подпоследовательностей на стыке фрагментов предложена техника разбиения временного ряда на фрагменты с перекрытием. Для обменов данными между вычислительными узлами кластера используется технология MPI, для распараллеливания в рамках одного вычислительного узла кластера используется технология OpenMP. Для эффективного использования возможностей векторизации вычислений ускорителя в рамках одного вычислительного узла используются дополнительные матричные структуры данных и избыточные вычисления. Проведены вычислительные эксперименты, показывающие высокую производительность и масштабируемость алгоритма *PBM* как на синтетических и реальных наборах данных как на кластерной системе в целом, так и в рамках одного вычислительного узла кластера и превосходство в производительности над алгоритмами-аналогами.

Предложен оригинальный параллельный алгоритм *MDD (Many-core Discord Discovery)* поиска диссонансов во временном ряде для многоядерных ускорителей. Распараллеливание выполнено с помощью технологии программирования OpenMP. Алгоритм состоит из двух стадий: подготовка и поиск. На стадии подготовки выполняется построение вспомогательных матричных структур данных, обеспечивающих эффективную векторизацию вычислений на ускорителе. На стадии поиска алгоритм находит диссонанс с помощью построенных структур. Проведены вычислительные эксперименты, исследующие производительность и масштабируемость алгоритма на реальных наборах данных. Результаты экспериментов показали хорошую масштабируемость алгоритма *MDD* и превосходство в производительности над алгоритмами-аналогами.

Основные результаты, полученные в данной главе, опубликованы в работах [7, 10, 11, 26, 125–127, 158, 162, 163, 264].

Глава 4. Интеграция в СУБД параллельных алгоритмов анализа данных

В данной главе предлагается новый подход к интеграции интеллектуального анализа данных и реляционных СУБД. Указанный подход предполагает внедрение параллельных алгоритмов интеллектуального анализа данных в реляционную СУБД открытым кодом на основе определяемых пользователем функций. В качестве целевой площадки реализации подхода рассмотрена СУБД PostgreSQL. Описана системная архитектура и методы реализации подхода. Приведены результаты вычислительных экспериментов, исследующих эффективность предложенного подхода.

4.1. Базовые идеи и мотивационный пример

Целью интеграции интеллектуального анализа данных и реляционной СУБД является обеспечение прикладного программиста баз данных эффективными, удобными и прозрачными средствами, которые позволяют выполнять интеллектуальный анализ данных, хранить и обрабатывать результаты этого анализа, не выходя за рамки СУБД. Подход к решению указанной проблемы, предлагаемый в данной главе, базируется на следующих основных идеях.

— *Инкапсуляция параллельных алгоритмов для современных многоядерных процессоров.* Аналитические алгоритмы, интегрируемые в реляционную СУБД, должны быть параллельными и ориентированными на современные многоядерные процессоры. Детали реализации параллельного исполнения аналитических алгоритмов должны быть скрыты от прикладного программиста баз данных.

— *Предвычисления.* Интеграция алгоритмов анализа данных в реляционную СУБД должна обеспечивать буферный пул для долговременного хранения в оперативной памяти предварительно вычисляемых структур данных, которые могут быть многократно использованы в дальнейшем аналитическими алгоритмами.

— *Промежуточное представление результатов анализа данных.* Результаты анализа данных должны сохраняться в промежуточном (нереляционном) представлении, в качестве которого используется формат JSON (JavaScript Object Notation) [65]. Прикладной программист должен быть обеспечен функциями, которые позволяют выполнить выборку нужных результатов из промежуточного представления и сохранить эти результаты в виде реляционных таблиц в базе данных.

Указанные идеи имеют следующее обоснование:

— Использование параллельных алгоритмов для многоядерных процессоров обеспечит производительность аналитических функций СУБД. При этом инкапсуляция параллелизма обеспечивает прозрачное использование аналитических функций прикладным программистом баз данных.

— Предварительное вычисление, хранение в оперативной памяти и повторное использование структур данных снизит накладные вычислительные расходы при выполнении аналитических алгоритмов.

— Использование формата JSON обеспечивает для различных базовых аналитических задач (поиск шаблонов, кластеризация и др.) возможность сохранения результатов, которые не всегда могут быть вписаны в предопределенную реляционную схему данных, требующую атомарности значений в ячейках реляционных таблиц (например, найденные частые наборы при решении задачи поиска шаблонов).

В качестве целевой площадки реализации описываемого подхода далее будет рассматриваться свободная СУБД PostgreSQL, соответствующая библиотека прикладного программиста PostgreSQL названа *pgMining*.

```
-- Установка подключения к серверу PostgreSQL, инициализация библиотеки
pgdmConnect("dbname=postgres hostaddr=10.4.5.204 port=5432");

-- Кластеризация данных с помощью алгоритма PAM
-- Таблица с точками данных: Points
-- Размерность точек (количество столбцов): 3
-- Количество кластеров: 6
SELECT pgdmPAM("Points", 3, 6);

-- Протокол и результаты кластеризации выводятся
-- в виде документа в промежуточном формате JSON
SELECT *
FROM pgdmClusteringResults
WHERE id = pgdmResult();

-- Результат кластеризации сохраняется в виде таблицы
-- MyCluster(номер точки, номер кластера, координаты точки)
CREATE TABLE MyCluster AS
  SELECT *
  FROM pgdmClusters(pgdmResult()) as Clusters, Points
  WHERE Clusters.id = Points.id;

-- Кластеризация данных с помощью алгоритма PAM
-- Таблица с точками данных: Points
-- Размерность точек (количество столбцов): 3
-- Количество кластеров: 4
SELECT pgdmPAM("Points", 3, 4);

-- Результат кластеризации выводится в виде таблицы
-- (номер точки, номер кластера, координаты точки)
SELECT *
FROM pgdmClusters(pgdmResult()) as Clusters, Points
WHERE Clusters.id = Points.id;
```

Рис. 4.1. Пример использования библиотеки *pgMining*

На рис. 4.1 представлен пример использования системы *pgMining*. Пример отражает сценарий, в котором исследователю требуется дважды выполнить кластеризацию точек данных, хранящихся в реляционной таблице, с различным количеством кластеров в каждом случае. Кластеризация выполняется с помощью алгоритма PAM (Partitioning Around Medoids) [207].

Вызов аналитической функции `pgdmPAM` инкапсулирует параллельное исполнение аналитического алгоритма на многоядерном процессоре, а также запись промежуточного представления результатов анализа в формате JSON в служебную таблицу библиотеки `pgdmClusteringResults`.

Идущий следом запрос к служебной таблице позволяет получить JSON-

документ с протоколом и результатами анализа, и выполнить его разбор или визуальное изучение. В данном запросе используется библиотечная функция `pgdmResult`, которая возвращает уникальный идентификатор последней выполненной аналитической операции.

Далее запрос позволяет сохранить результаты кластеризации в виде таблицы со следующими полями: номер точки, номер кластера, а также отдельное поле для каждой координаты точки.

После этого повторный вызов аналитической функции `pgdmPAM` выполняет кластеризацию вышеупомянутых данных, но с другим значением количества кластеров. Как и в первом случае, вызов инкапсулирует параллелизм аналитического алгоритма. В данном вызове также инкапсулировано повторное использование матрицы расстояний между кластеризуемыми точками, которая предварительно вычислена при первом вызове аналитической функции.

4.2. Системная архитектура

Системная архитектура интеграции интеллектуального анализа данных в реляционную СУБД с открытым кодом представлена на рис. 4.2. Компонент *pgMining* обеспечивает библиотечные функции, выполняющие интеллектуальный анализ данных внутри СУБД. Компонент *mcMining* обеспечивает библиотеку функций, которые реализуют параллельные алгоритмы интеллектуального анализа данных в оперативной памяти для многоядерных процессоров. В структуре решения присутствует компонент СУБД, обеспечивающий низкоуровневые функции доступа к данным. Для СУБД PostgreSQL в качестве данного компонента используется PostgreSQL *SPI (Server Programming Interface)*.

Компонент *pgMining* состоит из двух основных подсистем: *Frontend* и *Backend*. Подсистема *Frontend* представляет собой интерфейс прикладного программиста баз данных и обеспечивает функции, позволяющие вы-

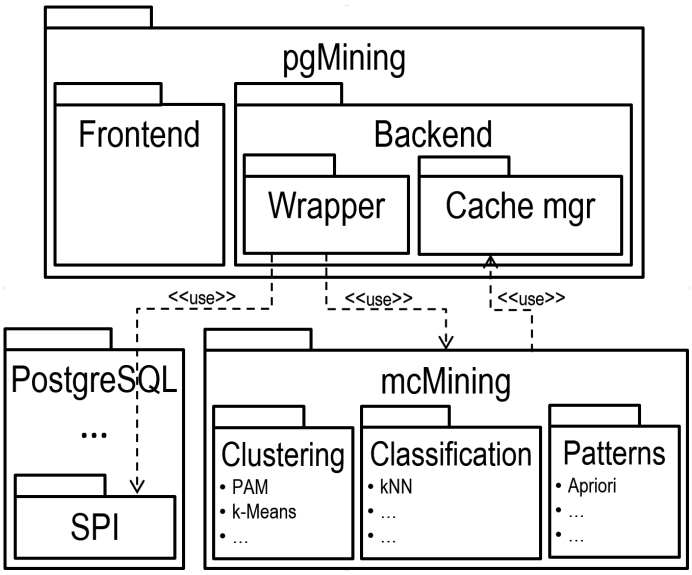


Рис. 4.2. Системная архитектура интеграции интеллектуального анализа данных в СУБД PostgreSQL

полнять анализ данных внутри СУБД и извлекать результаты анализа, сохраненные в виде промежуточного описания в формате JSON.

Подсистема *Backend* реализует внутренний интерфейс и состоит из следующих модулей: *Wrapper* и *Cache manager*. Модуль *Wrapper* обеспечивает обертки для функций библиотеки *mcMining*, позволяющие осуществлять вызов параллельного аналитического алгоритма и сохранять результат анализа в базе данных в виде промежуточного описания в формате JSON.

Модуль *Cache manager* обеспечивает буферный пул для долговременного хранения в оперативной памяти предварительно вычисляемых структур данных, которые могут быть многократно использованы в дальнейшем для интеллектуального анализа данных.

4.2.1. Внешний и внутренний интерфейсы

Внешний интерфейс (подсистема *Frontend*) обеспечивает функции двух видов: анализаторы и экстракторы. *Функция-анализатор* выполняет интеллектуальный анализ данных в рамках СУБД и формирует промежуточное представление результатов анализа в формате JSON. *Функция-экс-*

трактор трансформирует результаты анализа из промежуточного описания в формате JSON в табличный вид.

```
-- Кластеризация данных с помощью алгоритма PAM
FUNCTION pgdmPAM(
    text name, -- Имя таблицы с точками данных
    integer dim, -- Размерность точек
    integer k, -- Количество кластеров
) RETURNS integer -- ИД операции
```

Рис. 4.3. Интерфейс функции-анализатора

Пример интерфейса функции-анализатора представлен на рис. 4.3. В качестве параметров в функцию передаются имя таблицы с данными, подлежащими анализу, а также другие параметры, которые являются специфичными для конкретного аналитического алгоритма (например, количество кластеров для задачи кластеризации или пороговое значение минимальной поддержки для задачи поиска шаблонов).

Параметры функции-анализатора имеют схожую семантику, что и соответствующая ей функция библиотеки *mcMining* (см. раздел 4.2.3). Функция-анализатор возвращает уникальный идентификатор аналитической операции, который в дальнейшем используется прикладным программистом для преобразования результатов анализа в реляционные таблицы.

```
FUNCTION pgdmClusters(
    clusteringResultId integer -- ИД операции
) RETURNS TABLE(
    pointId integer, -- Номер точки данных
    clusterId integer) -- Номер кластера
```

Рис. 4.4. Интерфейс функции-экстрактора

Интерфейс функции-экстрактора представлен на рис. 4.4. В качестве параметров в функцию передаются уникальный идентификатор аналитической операции, выполненной ранее с помощью вызова функции-анализатора. Функция возвращает соответствующую таблицу данных, полученных из JSON-элемента данных.

```
1 // Обертка системного уровня для алгоритма кластеризации PAM.
2 // Входные данные соответствуют входным данным обертки функции-анализатора.
3 // Возвращает результат в виде текста промежуточного описания в формате JSON.
4 Datum wrap_pgPAM(PG_FUNCTION_ARGS);
```

Рис. 4.5. Интерфейс обертки системного уровня

Модуль *Wrapper*, входящий в подсистему *Backend*, для каждой функции-анализатора обеспечивает обертку параллельного аналитического алгоритма. На рис. 4.5 приведен интерфейс обертки для функции-анализатора, представленной выше на рис. 4.3.

Обертка системного уровня возвращает значения типа `Datum`, представляющий собой универсальный тип данных СУБД PostgreSQL, к которому может быть приведено значение любого типа. Стандартная макроподстановка СУБД PostgreSQL `PG_FUNCTION_ARGS` используется для передачи параметров в пользовательские функции.

4.2.2. Управление буферным пулом

```
1  // Загрузить объект в кэш.
2  // Возвращает 0 в случае успеха или отрицательный код ошибки.
3  int cache_putObject(
4    char * objID, // Уникальный идентификатор объекта, размещаемого в кэше
5    void * data,  // Указатель на буфер с данными
6    int size);    // Размер данных
7
8  // Поиск в кэше объекта с заданным уникальным идентификатором.
9  // Возвращет указатель на объект или NULL в случае отсутствия объекта.
10 void * cache_getObject(char * objID);
```

Рис. 4.6. Интерфейс подсистемы *Cache manager*

Модуль *Cache manager* экспортирует основные функции, представленные на рис. 4.6, которые обеспечивают управление буферным пулом для хранения в оперативной памяти предварительно вычисляемых структур данных.

Функция `cache_getObject` выполняет поиск в буферном пуле предвычисленной структуры данных с указанным идентификатором. Функция

`cache_putObject` выполняет загрузку указанной структуры данных в буферный пул (если она не была загружена ранее). В случае, если в буферном пуле отсутствует запрашиваемое пространство, выполняется выбор структуры данных-«жертвы», которая должна быть замещена поступившей. Выбор «жертвы» осуществляется в соответствии со стратегией замещения, которая является параметром системы. В качестве стратегии замещения используется один из известных алгоритмов замещения страниц в буферном пуле, применяемых в операционных системах и СУБД: LRU-K [170], LFU-K [223] и др.

4.2.3. Библиотека параллельных алгоритмов

```
1 // Кластеризация данных PAM (Partitioning Around Medoids).
2 // Возвращает 0 в случае успеха или отрицательный код ошибки.
3 int mcPAM(
4   void * inpData, // Указатель на буфер с точками
5   int dim,        // Размерность точек
6   int N,          // Количество точек
7   int k,          // Количество кластеров
8   void * outData,  // Указатель на буфер с выходными данными
9   void * distMatr); // Предвычисленная матрица расстояний
```

Рис. 4.7. Интерфейс функции библиотеки *mcMining*

Компонент *mcMining* представляет собой библиотеку функций, которые реализуют параллельные алгоритмы интеллектуального анализа данных в оперативной памяти для многоядерных процессоров. Пример интерфейса библиотечной функции представлен на рис. 4.7.

В качестве параметров в функцию передаются указатель на буфер с исходными данными, подлежащими анализу, указатель на буфер для записи результатов анализа, а также другие параметры, которые являются специфичными для конкретного аналитического алгоритма (например, количество кластеров для задачи кластеризации или пороговое значение минимальной поддержки для задачи поиска шаблонов).

В интерфейс библиотечной функции также включается один или более параметров, обеспечивающих использование параллельным аналити-

ческим алгоритмом предварительно вычисленных структур данных. Количество и семантика указанных параметров зависят от конкретного аналитического алгоритма.

Типичным примером предварительно вычисляемой структуры данных может служить матрица расстояний, в которой хранятся значения расстояний (схожести) между каждой парой элементов исходного множества. Матрица расстояний может быть многократно использована для выполнения кластеризации данных или классификации по принципу ближайших соседей элементов исходного множества с различными параметрами (количество кластеров, количество «соседей» и др.).

4.3. Методы реализации

4.3.1. Организация хранения системных данных

```
1  -- Таблица для хранения схем JSON-документов с результатами анализа данных
2  CREATE TABLE pgMiningSchemaTab (
3    ID integer primary key, -- Уникальный идентификатор описания
4    Schema JSON, -- Формальное описание структуры JSON-документа
5    Name char(50)) -- Наименование аналитической задачи
6
7  -- Таблица для хранения промежуточных представлений результатов кластеризации
8  CREATE TABLE pgClusteringTab (
9    ID integer primary key, -- Уникальный идентификатор операции
10   SchemaID integer -- Внешний ключ, ссылающийся на схему
11       foreign key SchemaID references pgMiningSchemaTab (ID),
12   Result JSON) -- Промежуточное представление в формате JSON
13
14 -- Генератор уникальных идентификаторов операций
15 CREATE SEQUENCE pgMiningID;
```

Рис. 4.8. Схема системной базы данных

На рис. 4.8 представлена схема базы данных для хранения системных данных, необходимых для реализации описываемого подхода. Таблица `pgMiningSchemaTab` предназначена для хранения JSON-схем промежуточного описания результатов анализа данных. Схема представляет собой формальное описание структуры JSON-документа, выполненное на языке

JSON Schema [196], который, в свою очередь, использует синтаксис JSON. Данная таблица заполняется однократно во время инициализации системы.

```
{
  "dataset" : { // Данные о кластеризуемых данных
    "name" : "Название набора данных",
    "pointsNumber" : 132, // Количество точек
    "pointsDimension" : 3, // Размерность точек
    "points" : [ // Координаты точек
      [3.24, -24.18, 0], // 1-я точка
      [18.45, 32.11, 2.2], // 2-я точка
  ...
    ]
  },
  "clusteringResults": { // Данные о результатах кластеризации
  "clusteringMethod" : "PAM", // Алгоритм кластеризации
  "clustersNumber": 3, // Количество кластеров
  "clustersCapacity" : [57, 33, 42], // Количество точек в кластерах
  "medoids" : [45, 12, 8], // Номера медоидов (точек-представителей кластеров)
    "centroids" : [ // Координаты центров кластеров
    [1.0, 2.8, 7.0],
      [12.8, 7.2, -3.1],
    [-2.2, -3.7, 15.8]
  ],
    "clusters" : [ // Номера точек, разбитых по кластерам
    [2, 7, 33, 42, 44, 45, 87, ...], // Точки 1-го кластера
    [1, 3, 4, 6, 12, 15], // Точки 2-го кластера
    [5, 8, 9, 10, 11, 13, 14, ...] // Точки 3-го кластера
    ]
  }
}
```

Рис. 4.9. Пример JSON-описания результатов анализа данных

Таблица `pgClusteringTab` предназначена для хранения промежуточных JSON-описаний с результатами кластерного анализа данных и включает в себя уникальный идентификатор аналитической операции (как первичный ключ), внешний ключ, обеспечивающий ссылку на схему описания из таблицы `pgMiningSchemaTab`, и собственно JSON-описание. Результатом каждого вызова функции-анализатора является добавление записи в данную таблицу. Пример промежуточного описания результатов кластеризации данных представлен на рис. 4.9. В текстовом виде в формате JSON хранятся данные о данных, подвергаемых кластеризации и результатах анализа.

Объект схемы хранения системных данных `pgMiningID` обеспечивает генерацию уникальных идентификаторов аналитических операций в виде последовательности уникальных значений.

4.3.2. Подсистема *Frontend*

```
// Кластеризация данных алгоритмом PAM в СУБД PostgreSQL
// Возвращает положительный ИД операции или отрицательный код ошибки.
int pgdmPAM (
  char * inpTab, // Имя таблицы с точками для кластеризации
  int dim,       // Размерность точек (количество столбцов таблицы)
  int k)         // Количество кластеров
{
  // Регистрация хранимой процедуры-обертки в схеме базы данных
  PQexec(pgdm_conninfo, "CREATE FUNCTION
    wrap_pgPAM(text, integer, integer, real) RETURNS text AS
      'pgmining', 'wrap_pgPAM' LANGUAGE C STRICT;");
  // Соединение с сервером и запуск хранимой процедуры-обертки
  return PQexec(conn,
    "INSERT INTO pgClusteringTab (ID, SchemaID, Result) VALUES (
        nextval('pgMiningID'), // Формирование ИД операции
        ClusteringSchema,      // ИД схемы для операции кластеризации
        SELECT wrap_pgPAM(%s, %d, %d);", // Вызов системной обертки
    inpTab, dim, k)); // Передача параметров в системную обертку
}
```

Рис. 4.10. Схема реализации функции-анализатора

Схема реализации функции-анализатора представлена на рис. 4.10. С помощью стандартных функций СУБД PostgreSQL выполняется регистрация в схеме базы данных хранимой процедуры-обертки, обеспечиваемой модулем *Wrapper*. Далее СУБД осуществляет вызов обертки с параметрами, переданными в функцию-анализатор. Данный вызов выполняется в рамках запроса, который осуществляет вставку в базу данных промежуточного описания результатов анализа в формате JSON, сформированного оберткой.

Схема реализации функции-экстрактора представлена на рис. 4.11. Входными данными функции-экстрактора являются уникальный идентификатор аналитической операции, имя JSON-элемента с промежуточным описанием результатов, которые необходимо сохранить в реляционном виде,

```
// Извлечение результатов кластеризации данных в форме реляционной таблицы
// Возвращает 0 в случае успеха или отрицательный код ошибки.
int pgClusterExtract (
  int ID,      // Уникальный идентификатор аналитической операции
  char * resName, // Имя JSON-элемента с промежуточным описанием результатов
  char * outTab)  // Имя таблицы для записи результатов
{
  char * relSchema = _pgdmExtractRelScheme(ID, resName);
  return PQexec(pgdm_conninfo,
    "INSERT INTO %s
        SELECT json_to_record(Result->%s) as x(%s)
        FROM pgClusteringTab, pgMiningSchemaTab
        WHERE pgClusteringTab.ID = %d AND
              pgClusteringTab.SchemaID = pgMiningSchemaTab.ID",
    outTab, resName, relSchema, ID);
}
```

Рис. 4.11. Схема реализации функции-экстрактора

и имя таблицы для сохранения результатов. В реализации используются стандартные функции обработки JSON-данных СУБД PostgreSQL, которые позволяют получить типы данных JSON-элементов и преобразовать JSON-элементы в реляционные кортежи, которые сохраняются в реляционной таблице.

4.3.3. Подсистема *Backend*

Модуль *Wrapper* для каждой функции-анализатора обеспечивает обертку параллельного аналитического алгоритма. Пример такой обертки приведен на рис. 4.12. Реализация обертки организована следующим образом. Сначала осуществляется разбор параметров, переданных от функции-анализатора. Далее с помощью методов, экспортируемых бекендом, выполняется проверка наличия входных и предвычисленных данных в буферном пуле системы и их загрузка в кэш в случае отсутствия. Далее осуществляется вызов функции библиотеки *mcMining*, которая непосредственно исполняет аналитический алгоритм на многоядерном ускорителе. По завершении работы данной функции полученные результаты сериализуются в текст с JSON-описанием. Итоговый текст возвращается в качестве результата работы функции-обертки.

```
// Обертка системного уровня для алгоритма кластеризации PAM.
// Входные данные соответствуют входным данным обертки функции-анализатора.
// Возвращает результат в виде текста промежуточного описания в формате JSON.
Datum wrap_pgPAM(PG_FUNCTION_ARGS)
{
  // Извлечение и преобразование входных параметров
  // Имя таблицы с точками для кластеризации
  char * inpTab = text_to_cstring(PG_GETARG_TEXT_P(0));
  int dim = PG_GETARG_INT32(1); // Размерность точек данных
  int k = PG_GETARG_INT32(2); // Количество кластеров
  void * inpData; // Указатель на буфер с данными для размещения в кэше

  // Проверка наличия предвычисленной структуры данных в кэше
  void * distMatr = cache_getObject(strcat(inpTab, "_distMatr"));
  if (distMatr == NULL) {
    // Проверка наличия данных в кэше
  inpData = cache_getObject(inpTab);
  if (inpData == NULL) {
    // Загрузка данных в кэш
    inpData = (float *) malloc(dim * N * sizeof(float));
    wrap_tabRead(inpData, inpTab, dim, &N);
    cache_putObject(inpTab, inpData, sizeof(inpData));
  }
  distMatr = mcCalcMatrix(inpData, dim, N);
  cache_putObject(strcat(inpTab, "_distMatr"), distMatr, sizeof(distMatr));
  }
  // Выполнение кластеризации и сохранение результатов
  mcPAM_res * outData = mcPAM_resCreate();
  mcPAM(N, k, outData, distMatr);
  PG_RETURN_TEXT(JSON_pgPAM(N, k, outData, distMatr));
}
```

Рис. 4.12. Схема реализации обертки системного уровня

4.4. Библиотека параллельных алгоритмов

В данном разделе описаны два параллельных алгоритма для многоядерных ускорителей, включаемых в библиотеку *mcMining* (см. раздел 4.2): алгоритм *PBlockwise* для вычисления матрицы Евклидовых расстояний и алгоритм *PPAM* для кластеризации точек Евклидова пространства.

4.4.1. Алгоритм *PBlockwise* вычисления матрицы евклидовых расстояний

Под матрицей Евклидовых расстояний (МЕР) в данной работе понимается прямоугольная матрица, содержащая квадраты расстояний[1] между точками двух непустых множеств в Евклидовом пространстве. МЕР играют важную роль в приложениях, связанных с кластеризацией данных, где необходимо вычислять расстояние между центрами кластеров и точек данных, подвергаемых кластеризации: в задачах сегментирования медицинских изображений [132, 263], иерархической [54] и нечеткой [69] кластеризации данных ДНК-микрочипов и др.

Рассмотрим два непустых конечных множества из n и m точек в d-мерном евклидовом пространстве. Точки первого множества рассматриваются как строки матрицы $A \in \mathbb{R}^{n\times d}$, точки второго множества — как строки матрицы $B \in \mathbb{R}^{m\times d}$. Обозначим как $A(1,\cdot),\ \ldots,\ A(n,\cdot)$ и $B(1,\cdot),\ \ldots,\ B(m,\cdot)$, где $A(i,\cdot), B(j,\cdot) \in \mathbb{R}^d$, строки матриц A и B, соответственно. Тогда матрица евклидовых расстояний $M \in \mathbb{R}^{n\times m}$ состоит из строк $M(1,\cdot),\ \ldots,$ $M(n,\cdot)$, где $M(i,\cdot) \in \mathbb{R}^m$, $M(i,j) = \|A(i,\cdot) - B(j,\cdot)\|^2$, и $\|\cdot\|$ означает евклидову норму[2].

Предлагаемый алгоритм, получивший название *PBlockwise*, итеративно вычисляет несколько расстояний от одной точки из первого множества до блока точек из второго множества, где количество точек в блоке, *block* — параметр алгоритма (см. алг. 4.1).

Внешний цикл (строка 2) перебирает первое множество точек и распараллеливается. Внутренний цикл в строке 3 перебирает блоки точек из второго множества. Цикл в строке 5 выполняет вычисления по координатам блока точек. Цикл в строке 8 вычисляет расстояния и компилируется в две векторные операции. Строки 6 и 7 подсказывают компилятору о выравнивании в памяти точки из первого множества и блока точек из второго

[1] Строго говоря, матрица должна содержать расстояния, но не их квадраты. Тем не менее, большинство авторов работ о матрицах Евклидовых расстояний придерживается данного соглашения [73].

[2] Заметим, что данное определение покрывает случай $A \equiv B$.

Алг. 4.1. PBlockwise(in A, $\tilde{B}$, *block*; out D)

```
1:  #pragma omp parallel for
2:  for i from 1 to n do
3:      for j from 1 to ⌈m/block⌉ do
4:          sum ← 0̄
5:          for k from 1 to d do
6:              __assume_aligned(A(i, ·), 64)
7:              __assume_aligned(B(j + k, ·), 64)
8:              for ℓ from 1 to block do
9:                  sum_ℓ ← sum_ℓ + (A(i, k) − B̃(j + k, ℓ))^2
10:             end for
11:         end for
12:         __assume_aligned(D(i, ·), 64)
13:         for k from 1 to block do
14:             D(i, j · block + k) ← sum_k
15:         end for
16:     end for
17: end for
18: return D
```

множества соответственно. Наконец, цикл в строке 13 сохраняет вычисленные расстояния в матрицу расстояний, причем эта операция компилируется в одну векторную операцию. Данному циклу предшествует подсказка для компилятора (строка 12) о выравнивании результирующей матрицы расстояний в памяти.

Описанный алгоритм предполагает, что мощность второго множества точек m кратна *block*. Если это не так, необходимо увеличить m до ближайшего целого числа, кратного *block*, путем дополнения матрицы B фиктивными нулевыми строками. Для гарантии эффективной векторизации операций над матрицей B параметр *block* должен быть кратен значению ширины векторного регистра. Шириной векторного регистра называют максимальное количество элементов данных, которое можно поместить в регистр. Для Intel Xeon Phi размер векторного регистра составляет 512 бит, что позволяет разместить в нем 16 вещественных чисел с одинарной точностью. Для получения большей выгоды от векторизации вычислений в

качестве матрицы B необходимо взять матрицу, которая хранит наибольшее по мощности множество точек из двух заданных.

```
typedef struct {
  float x;
  float y;
  float z;
} AoS;
```

AoS $\mathbf{B}[m]$;

(a) Массив из структур

```
typedef struct {
  float x[m];
  float y[m];
  float z[m];
} SoA;
```

SoA $\mathbf{B}$;

(b) Структура из массивов

```
typedef struct {
  float x[block];
  float y[block];
  float z[block];
} ASA;
```

ASA $\mathbf{B}[\lceil \frac{m}{block} \rceil]$;

(c) Массив структур из массивов

Рис. 4.13. Основные способы компоновки массивов в памяти

Алгоритм *PBlockwise* предполагает также, что данные второго множества точек хранятся в памяти с применением способа компоновки ASA (Array of Structures of Arrays). На рис. 4.13 представлены основные способы компоновки массивов в памяти (в виде объявлений типов данных языка C) [37].

Компоновка *AoS (Array of Structures)* предполагает хранение множества точек в виде массива, элементами которого являются структуры, и часто используется как компоновка данных по умолчанию. В случае компоновки данных *SoA (Structure of Arrays)* для хранения каждой координаты точек используются отдельные массивы. Это может приводить к конфликтам совместного доступа нитей к памяти, если сценарий доступа к данным предполагает обработку смежных элементов. Компоновка *ASA (Array of Structures of Arrays)* разбивает данные на блоки в соответствии с параметром *block*. ASA-*block* обобщает компоновки данных, рассмотренные выше: ASA-1 соответствует AoS, а ASA-m соответствует SoA. Такая усложненная компоновка данных способствует уменьшению кэш-промахов процессора и, соответственно, повышает производительность вычисления матрицы расстояний.

Для изменения компоновки второго множества точек применяется параллельный алгоритм, выполняющий перестановку элементов матрицы исходных данных (см. алг. 4.2). Для заданного размера *block* и матрицы

Алг. 4.2. PERMUTE(IN B, $block$; OUT $\tilde{B}$)

```
1: #pragma omp parallel for
2: for j from 1 to ⌈m/block⌉ do
3:     for i from 1 to d do
4:         for k from 1 to block do
5:             B̃(j · d + i, k) ← B(j · block + k, i)
6:         end for
7:     end for
8: end for
9: return B̃
```

$B \in \mathbb{R}^{m \times d}$ с компоновкой данных AoS, алгоритм создает матрицу $\tilde{B} \in \mathbb{R}^{d \cdot \lceil \frac{m}{block} \rceil \times block}$ с компоновкой данных ASA-$block$ (или SoA, если $block = m$).

4.4.2. Алгоритм *PPAM* кластеризации данных на основе техники медоидов

Алг. 4.3. PPAM(IN X, k, $block$, L; OUT C)

```
1: X_ASA ← PERMUTE(X, block)
2: M ← PBLOCKWISE(X, X_ASA, block)
3: C ← PBUILD(M, k, L)
4: repeat
5:     {T_min, c_min, x_min} ← PSWAP(C, M, L)
6:     C_{c_min} ← x_min
7: until T_min < 0
8: return C
```

Алгоритм *PPAM* (см. алг. 4.3) представляет собой параллельную версию алгоритма *PAM* [115], описанного в разделе 1.3.2, для многоядерных систем архитектуры Intel MIC [78], и использует ранее введенные соответствующие обозначения. Множество кластеризуемых объектов представляется в виде матрицы $X \in \mathbb{R}^{n \times d}$. В отличие от последовательного алгоритма, используется предварительное вычисление расстояний между объектами кластеризуемого множества X, которые сохраняются в матрице $M \in \mathbb{R}^{n \times n}$.

Алгоритм действует следующим образом. Сначала с помощью вспомогательного алгоритма *Permute* (см. алг. 4.2) элементы множества кластеризуемых объектов переставляются в соответствии с способом компоновки данных ASA (см. рис. 4.13). После этого выполняется предварительное вычисление матрицы Евклидовых расстояний с помощью вспомогательного алгоритма *PBlockwise* (см. раздел 4.4.1). Далее выполняется фаза *Build* алгоритма (см. алг. 4.4). Затем циклически выполняются следующие действия, реализующие фазу *Swap* алгоритма. Находится один из кластеризуемых объектов и один из медоидов, перестановка которых минимизирует значение целевой функции, и указанные объекты меняются местами. Цикл выполняется до тех пор, пока возможно улучшение значения целевой функции (найдены все медоиды, такие, что любая перестановка любого из них с кластеризуемым объектом ухудшит значение целевой функции). Алгоритм возвращает индексы медоидов в исходном множестве кластеризуемых объектов. Объект исходного множества принадлежит тому кластеру, медоид которого является ближайшим к данному объекту.

В параллельной реализации фаз алгоритма *Build* и *Swap* используется техника *тайлинга (tiling)* [37]. Тайлинг предполагает разбиение множества итераций счетчика исходного цикла на непересекающиеся подмножества меньшей мощности. Это приводит к разбиению исходного обрабатываемого массива на блоки *(тайлы)* меньшего размера, которые могут быть помещены в кэш-память процессора. Хранение данных в кэш-памяти для их неоднократного использования в процессе обработки тайла снижает количество кэш-промахов и увеличивает общую эффективность алгоритма. Указанная техника также дает компилятору больше информации о контексте исполнения, и позволяет выполнить векторизацию вычислений в теле цикла или развертку цикла [37]. Размер тайла является параметром алгоритма, обозначим его как L. В реализации используется рекомендуемый в руководстве [111] размер тайла L=32.

Для более компактной записи циклов алгоритма, к которым применена техника тайлинга, введем сокращенную запись оператора цикла, представ-

```
1 for i:=1 to N
    do
2   Process(A[i])
```

(a) Исходный цикл

```
1 for i:=1 to N step L do
2   for ii:=i to i+L do
3     Process(A[ii])
```

(b) Цикл с тайлингом

```
1 for i:=1 to N tile L
    do
2   Process(A[i])
```

(c) Краткая запись цикла с тайлингом

Рис. 4.14. Тайлинг цикла

ленную на рис. 4.14.

Алг. 4.4. PBuild(in M, k, L; out C)

```
1: D ← ∞̄
2: for ℓ from 1 to k do
3:     min ← ∞
4:     #pragma omp parallel for reduction(func_min(sum) : {min, C_ℓ})
5:     for i from 1 to n tile L do
6:         sum ← 0
7:         for j from 1 to n tile L do
8:             if D_j is ∞ then
9:                 sum ← sum + M(i, j)
10:            else if D_j > M(i, j) then
11:                sum ← sum + M(i, j) − D_j
12:            end if
13:        end for
14:        if sum < min then
15:            min ← sum; C_ℓ ← i
16:        end if
17:    end for
18:    #pragma omp parallel for
19:    for i from 1 to n do
20:        D_i ← min(D_i, M(C_ℓ, i))
21:    end for
22: end for
23: return C
```

Параллельная реализация фазы *Build* представлена в алг. 4.4. Алгоритм действует следующим образом. Вектор расстояний от точек до ближайшего медоида, D, инициализируется значением ∞. Далее выполняется цикл по всем кластерам, в котором осуществляется поиск медоида, мини-

мизирующего значение целевой функции. Тело указанного цикла организовано следующим образом. Выполняется сканирование всех кластеризуемых объектов, и для каждого объекта вычисляет значение целевой функции относительно данного объекта и ранее найденных медоидов. Объект, на котором достигнут минимум целевой функции, добавляется в множество медоидов. Данный цикл распараллеливается с помощью стандартной директивы компилятора OpenMP `#pragma omp parallel for`.

Алг. 4.5. PSWAP(IN C, M, L; OUT T_{min}, c_{min}, x_{min})

```
1:  D ← ∞; S ← ∞
2:  #pragma omp parallel for
3:  for h from 1 to n tile L do
4:      for i from 1 to k do
5:          if D_h > M(h, C_i) then
6:              S_h ← D_h
7:              D_h ← M(h, C_i)
8:          else if S_h > M(h, C_i) then
9:              S_h ← M(h, C_i)
10:         end if
11:     end for
12: end for
13: #pragma omp parallel for reduction(func_{min(T)} : {T_min, c_min, o_min})
14: for h from 1 to n tile L do
15:     for i from 1 to k do
16:         T ← 0
17:         for ℓ from 1 to n tile L do
18:             T ← T + DELTA(M, ℓ, C_i, h, D_ℓ, S_ℓ)
19:         end for
20:         if T < T_min then
21:             T_min ← T; c_min ← i; x_min ← h
22:         end if
23:     end for
24: end for
25: return {T_min, c_min, x_min}
```

Сканирование кластеризуемых объектов осуществляется с помощью описанной выше техники тайлинга. Этот же прием применяется во вложенном цикле, где выполняется вычисление значений целевой функции.

В качестве завершающей тело цикла операции алгоритм обновляет значения вектора расстояний от каждой точки до ближайшего к ней медоида. Алгоритм возвращает начальную расстановку медоидов, используемую далее в фазе *Swap*.

Параллельная реализация фазы *Swap* представлена в алг. 4.5. Алгоритм действует следующим образом. Выполняется сканирование кластеризуемых объектов, и выполняются следующие действия: замена каждого медоида на кластеризуемый объект и вычисление соответствующего изменения значения целевой функции. При этом осуществляется поиск минимального значения целевой функции и индексов медоида и кластеризуемого объекта, на которых найденный минимум достигается. Цикл, выполняющий сканирование, распараллеливается с помощью стандартной директивы компилятора OpenMP `pragma omp parallel for`.

Алг. 4.6. DELTA(IN M, j, c, h, d, s; OUT $delta$)

```
1:  if M(j,c) > d and M(j,h) > d then
2:      delta ← 0
3:  else if M(j,c) = d then
4:      if M(j,h) < s then
5:          delta ← M(j,h) − d
6:      else
7:          delta ← s − d
8:      end if
9:  else if M(j,h) < d then
10:     delta ← M(j,h) − d
11: end if
12: return delta
```

Реализация вычисления изменения значения целевой функции соответствует классическому алгоритму PAM (см. алг. 1.4) и представлена в алг. 4.6.

4.5. Вычислительные эксперименты

Цели экспериментов

В экспериментах исследовалась эффективность предложенных алгоритмов *PBlockwise* и *PPAM*, а также эффективность использования этих алгоритмов в рамках библиотеки *pgMining*.

Производительность и масштабируемость алгоритма *PBlockwise* сравнивалась с прямолинейным алгоритмом вычисления MEP, описанным в работе [132] (далее будет упоминаться как *Straightforward*), и реализацией вычисления MEP с помощью функции из библиотеки Intel Math Kernel Library (MKL) [270], оптимизированной для Intel Xeon Phi.

Также исследовалось качество кластеризации, которое обеспечивает алгоритм *PPAM*, чтобы подтвердить его робастность (устойчивость к шумам в данных). Для этих целей использована мера *силуэтного коэффициента*, определяемая следующим образом [115] (где X — множество из n кластеризуемых объектов, ρ — функция расстояния, $C_1, \ldots, C_k$ — полученные кластеры, объекты $x \in C_i$, $y \in C_j$):

$$\begin{aligned} \mathcal{S}(X) &:= \tfrac{1}{n} \sum_{x \in X} \tfrac{b(x) - a(x)}{\max\{a(x), b(x)\}}, \\ a(x) &:= \tfrac{\sum_{y \in C_i \wedge x \neq y} \rho(x,y)}{|C_i| - 1}, \\ b(x) &:= \min_{C_j:\ 1 \leqslant j \leqslant k \wedge j \neq i} \tfrac{\sum_{y \in C_j} \rho(x,y)}{|C_j|}. \end{aligned} \tag{4.1}$$

Областью значений силуэтного коэффициента является интервал $(-1, 1)$, лучшему качеству соответствует большее значение [99].

Наборы данных

В экспериментах, исследующих производительность алгоритма, использовались наборы данных, описанные в табл. 4.1. Наборы данных Census [153], FCS Human [80] и Haiti [131] взяты из реальных приложений. Наборы данных MixSim и ADS являются синтетическими и получены с помощью

Табл. 4.1. Наборы данных для экспериментов

Набор	d	n	m	**Вид**	**Семантика**
MixSim	5	$35 \cdot 2^{10}$	$35 \cdot 2^{10}$	Синтетический	Получен генератором из [155]
Census	67	$35 \cdot 2^{10}$	$35 \cdot 2^{10}$	Реальный	Результаты переписи населения США [153]
FCS Human	423	$18 \cdot 2^{10}$	$18 \cdot 2^{10}$	Реальный	Агрегированная информация о геноме человека [80]
Power	3	$35 \cdot 2^{10}$	$35 \cdot 2^{10}$	Реальный	Информация о домашнем энергопотреблении [136]
ADS-16 ADS-64 ADS-256	16 64 256	10^6	10^3	Синтетический	Наборы данных из [132]
Haiti	3	$45 \cdot 2^{10}$	$45 \cdot 2^{10}$	Реальный	Спутниковое фото высокого разрешения [131]

программ-генераторов, описанных в работах [155] и [166] соответственно. Группа наборов данных ADS (Aligned Data Set) использовалась для экспериментальной оценки алгоритма *Straightforward* в работе [132].

Исследование качества кластеризации проводилось на наборах данных BLE RSSI [159] (более 6 500 15-атрибутных объектов) и Avila [224] (более 20.5 тыс. 10-атрибутных объектов), в которых у 2% объектов в два атрибута вносились шумы.

Аппаратная платформа

Табл. 4.2. Характеристики аппаратной платформы

Характеристика	**Хост**	**Сопроцессор MIC**
Модель, Intel Xeon	X5680	Phi (KNC), SE10X
Количество физических ядер	2×6	61
Гиперпоточность	2	4
Количество логических ядер	24	244
Частота, ГГц	3.33	1.1
Размер VPU, бит	128	512
Пиковая производительность, TFLOPS	0.371	1.076

Эксперименты проведены на узле вычислительного кластера Торнадо ЮУрГУ [6]. Характеристики хост и MIC системы узла приведены в табл. 4.2.

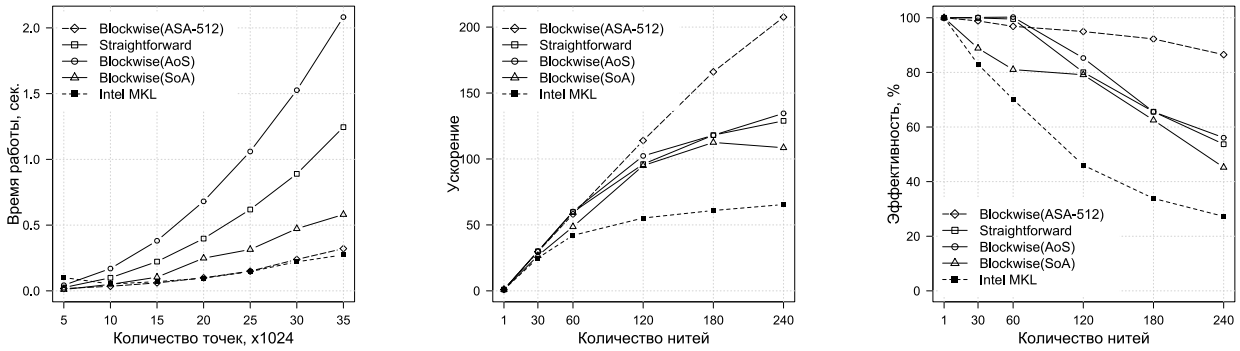
Результаты

Эффективность алгоритма *PBlockwise*. На рис. 4.15 и рис. 4.16 показаны время работы, ускорение и параллельная эффективность исследуемых алгоритмов вычисления квадратных и прямоугольных MEP соответственно.

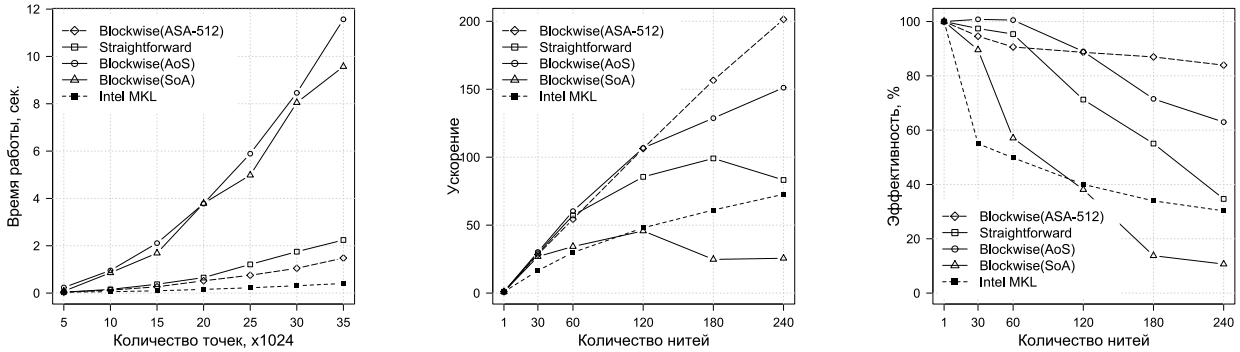
Результаты экспериментов с квадратными MEP показывают, что алгоритм *PBlockwise*(ASA-512) занимает второе место после Intel MKL, и почти сравнивается с Intel MKL на наборе данных MixSim, когда значение d выровнено на 16. В то же время Intel MKL показывает почти худшие ускорение и параллельную эффективность среди исследуемых алгоритмов.

Алгоритмы, за исключением Intel MKL и *PBlockwise*(SoA), показывают близкое к линейному ускорение и эффективность примерно 80%, когда количество нитей равно количеству физических ядер аппаратной платформы. При этом, если количество нитей превышает количество ядер, то только алгоритм *PBlockwise*(ASA-512) сохраняет ранее описанное поведение, показывая ускорение до 200 и эффективность как минимум 80%. В то же время ускорение и параллельная эффективность других алгоритмов перестают увеличиваться или даже резко падают.

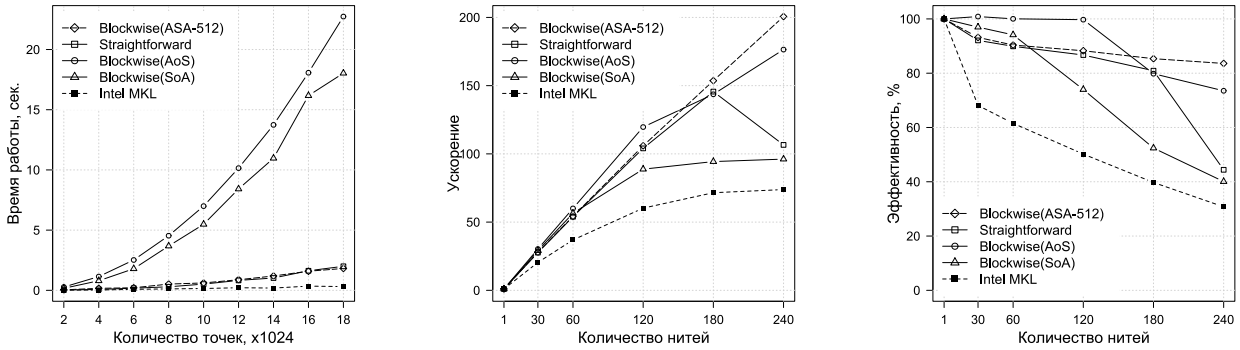
Эксперименты с вычислением прямоугольных матриц показывают следующие результаты. *PBlockwise*(ASA-512) превосходит по скорости работы все прочие алгоритмы на наборе данных ADS-16 и показывает сравнимый результат с алгоритмом Intel MKL на наборе ADS-64. На наборе данных ADS-256 алгоритм Intel MKL превосходит прочие алгоритмы. Что касается масштабируемости, можно видеть примерно ту же картину, что и для квадратных матриц. Если количество нитей совпадает с количеством физических ядер, *PBlockwise*(ASA-512) показывает ускорение, близкое к линейному, и параллельную эффективность 90%. В диапазоне от 60 до 240



(a) Набор данных MixSim (d выровнено до 16)



(b) Набор данных Census (d выровнено до 80)

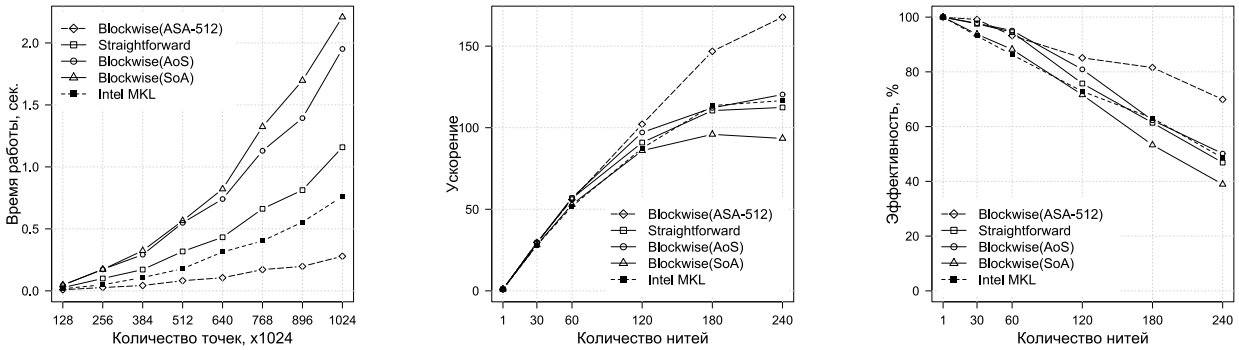


(c) Набор данных FCS Human (d выровнено до 432)

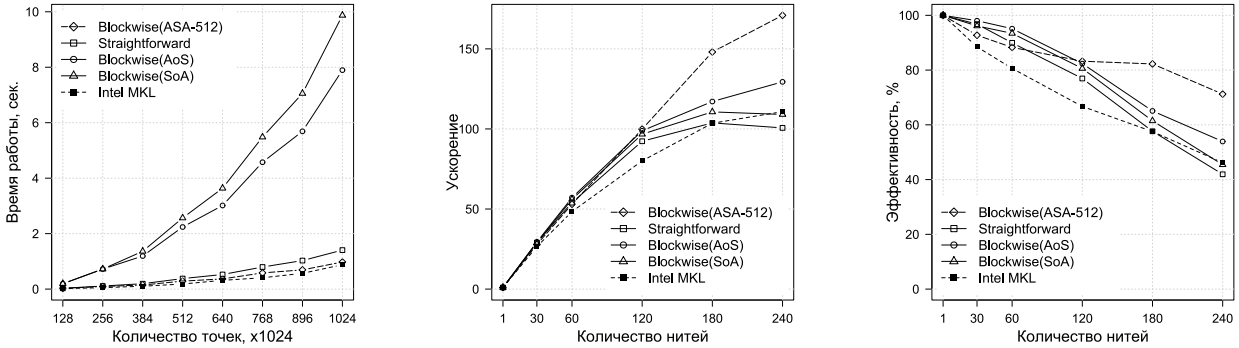
Рис. 4.15. Время работы, ускорение и эффективность алгоритма *PhiBlockwise* на квадратных матрицах

нитей *PBlockwise* показывает лучшую масштабируемость, ускоряясь до 160 раз и показывая параллельную эффективность 70%. Можно заключить, что алгоритм *PBlockwise*(ASA-512) показывает лучшие результаты на прямоугольных матрицах малой размерности (примерно при $d \leqslant 32$).

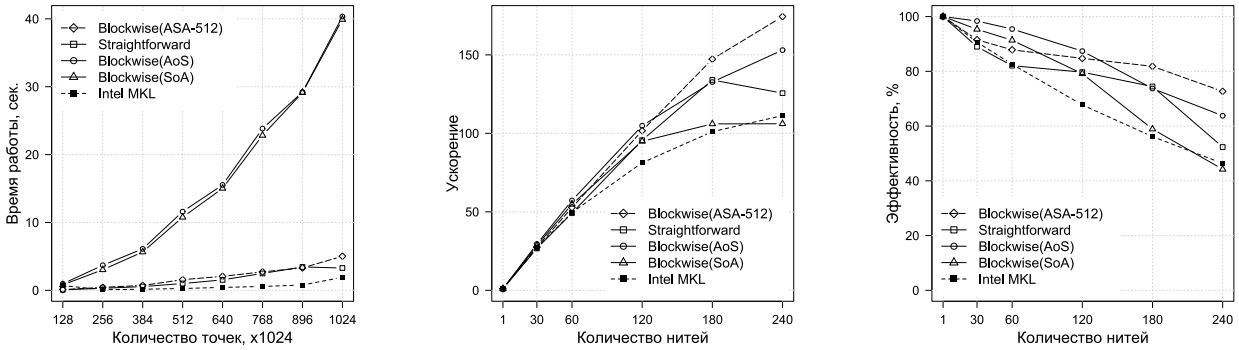
Производительность алгоритма *PBlockwise*(ASA-512) на системах Intel Xeon и Intel Xeon Phi в сравнении с *Straightforward* представлена в табл. 4.3.



(a) Набор данных ADS-16



(b) Набор данных ADS-64



(c) Набор данных ADS-256

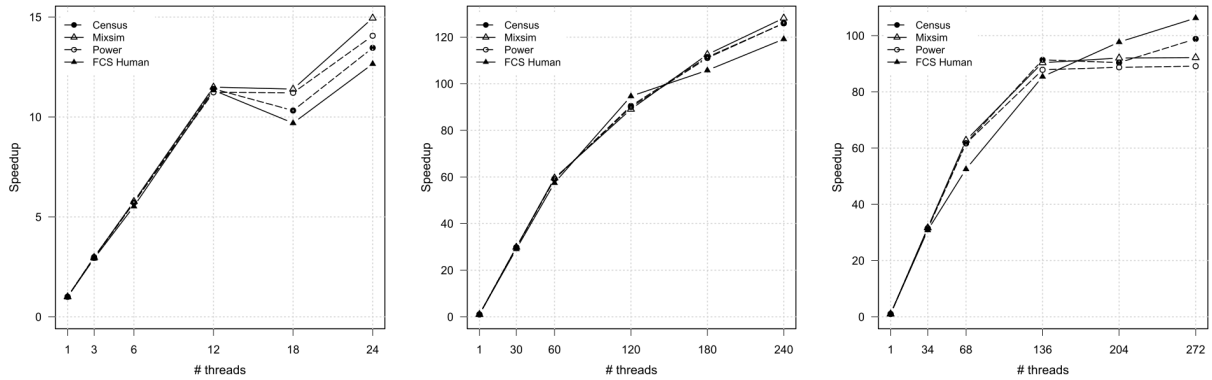
Рис. 4.16. Время работы, ускорение и эффективность алгоритма *PBlockwise* на прямоугольных матрицах

Можно видеть, что *PBlockwise*(ASA-512) работает в 3.5–8 раз быстрее на Intel Xeon Phi, чем на хосте с двумя процессорами Intel Xeon. Алгоритм *Straightforward*, также как и *PBlockwise*(ASA-512), быстрее работает на Intel Xeon Phi, чем на Intel Xeon. В то же время *PBlockwise* показывает лучшее время на указанных платформах. Отметим также, что алгоритм Intel MKL превосходит *PBlockwise*(ASA-512) на данных большой размерности

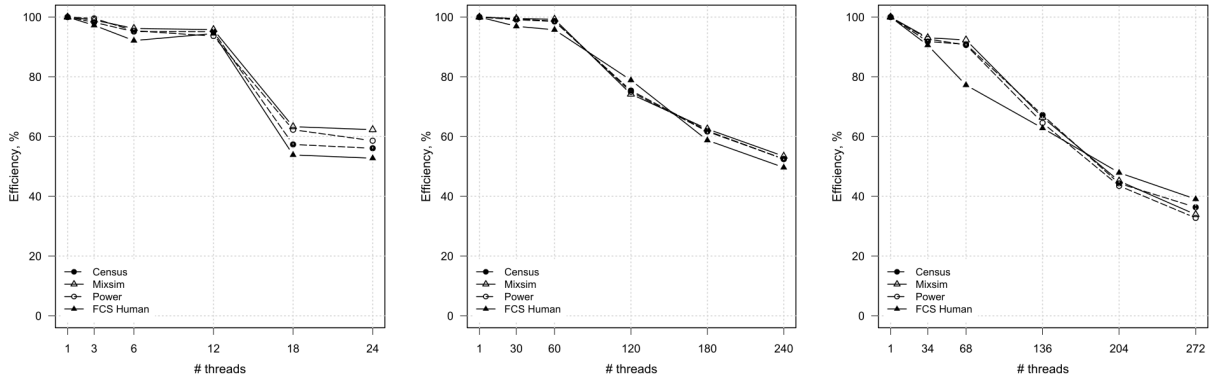
Табл. 4.3. Время работы на наборах данных ADS, с

Набор	**Intel Xeon Phi (KNC)** 1.076 TFLOPS			2×**Intel Xeon CPU** 0.371 TFLOPS			**Соотношение** 2×**CPU**/**Phi**	
	PBlockwise (ASA-512)	Intel MKL	Straight-forward	PBlockwise (ASA-512)	Intel MKL	Straight-forward	PBlockwise (ASA-512)	Straight-forward
ADS-16	0.28	0.76	1.05	1.04	3.02	1.00	3.7×	1.0×
ADS-64	0.98	0.88	1.36	3.78	3.81	4.25	3.9×	3.1×
ADS-256	3.71	1.92	3.79	30.32	5.14	31.41	8.2×	8.3×

(примерно при $d > 32$) на обеих платформах.



(a) Ускорение

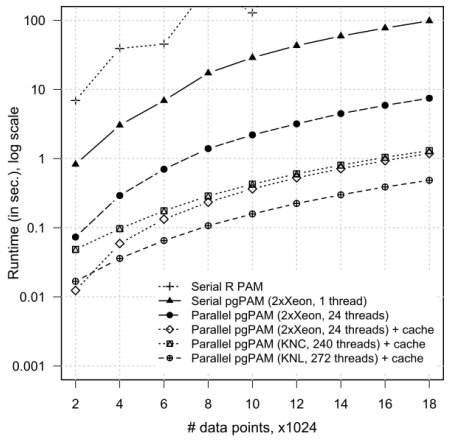


(b) Параллельная эффективность

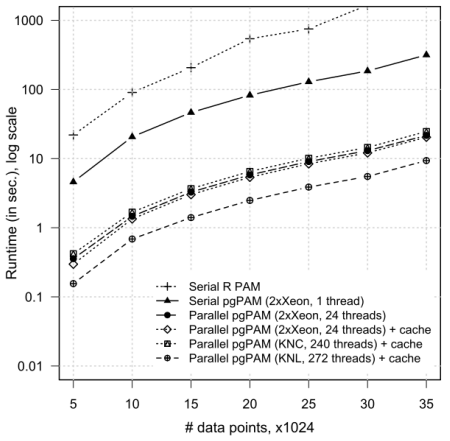
Рис. 4.17. Масштабируемость алгоритма *PPAM* на различных платформах (слева направо: 2×Intel Xeon, Intel Xeon Phi KNC, Intel Xeon Phi KNL)

Эффективность алгоритма *PPAM*. Результаты экспериментов представлены на рис. 4.17. Ускорение алгоритма близко к линейному, а параллельная эффективность, в свою очередь, — к 100%, когда количество нитей совпадает с количеством физических ядер, на которых выполняется алго-

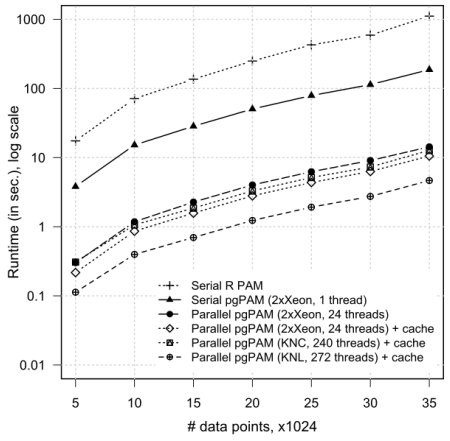
ритм (для всех рассматриваемых аппаратных платформ: на 12 ядрах для Intel Xeon, на 60 ядрах для Intel Xeon Phi KNC и 68 ядрах для Intel Xeon Phi KNL, соответственно). Ускорение и параллельная эффективность падают, когда алгоритм использует более одной нити на физическое ядро. Результаты экспериментов позволяют заключить, что после ускорения вычисления матрицы Евклидовых расстояний алгоритм *PPAM* показывает хорошую масштабируемость на рассматриваемых аппаратных платформах.



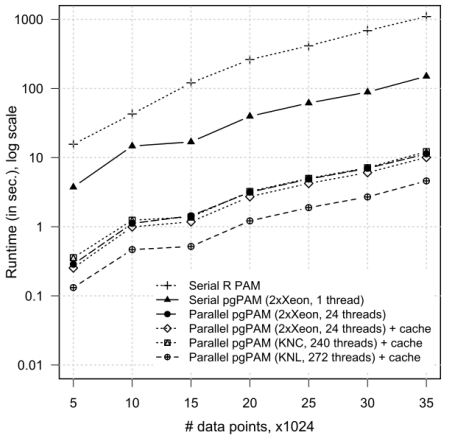
(a) Набор данных FCS Human



(b) Набор данных MixSim



(c) Набор данных Census



(d) Набор данных Power

Рис. 4.18. Производительность алгоритма *pgPAM*

Сравнение *PPAM* с аналогами. В табл. 4.4 представлены результаты сравнения быстродействия алгоритмов *PPAM* и *GPUPAM* на основе набора данных и результатов экспериментов из работы [131]. *PPAM* запускается на двух ядрах ускорителя Intel MIC, обеспечивающих пиковую про-

Табл. 4.4. Сравнение быстродействия алгоритмов *GPUPAM* и *PPAM*

Алгоритм	GPUPAM	PPAM
Платформа	NVIDIA GeForce410	Intel Xeon Phi 7290
Кол-во физ. ядер	48	2
Частота, ГГц	1.15	1.5
Пик. произв-ть, GFLOPS	73	96
Время работы, с	197	91

изводительность ускорителя, относительно близкую к производительности той платформы, на которой запускался алгоритм *GPUPAM*. Результаты показывают, что алгоритм *PPAM*, использующий предвычисление матрицы расстояний и технику тайлинга циклов в реализации фаз *Build* и *Swap*, в итоге опережает конкурента.

В табл. 4.5 представлены результаты экспериментов по сравнению качества кластеризации алгоритма *PPAM* с аналогами. Показано отношение времени выполнения параллельных версий различных алгоритмов кластеризации на одной и той же аппаратной платформе (в секундах) к значению силуэтного коэффициента [115] от результата кластеризации. Можно видеть, что разработанный алгоритм опережает конкурентов, поскольку демонстрирует более высокое качество кластеризации за счет большей устойчивости к шумам в данных.

Табл. 4.5. Сравнение качества кластеризации *PPAM* и аналогов

Алгоритм	Время/Качество кластеризации	
	Набор BLE RSSI [159] (2% шумов)	Набор Avila [224] (2% шумов)
k-Medians [97]	81.1	163.2
k-Medoids [168]	83.3	52.7
PPAM	**38.3**	**42.2**

Эффективность алгоритмов *PBlockwise* и *PPAM* в рамках библиотеки *pgMining*. Производительность алгоритма-обертки *pgPAM* сравнивалась с реализацией алгоритма *PAM*, имеющейся в системе интеллектуального анализа данных *R* [155] (на платформе процессора Intel Xeon).

Алгоритм *pgPAM* выполнялся на различных аппаратных платформах, при этом рассматривались два различных случая выполнения: a) матрица Евклидовых расстояний предварительно вычислена и помещена в буферный пул с помощью функций подсистемы *Cache Manager* и b) матрица матрица Евклидовых расстояний отсутствует в буферном пуле.

Результаты экспериментов представлены на рис. 4.18. Можно видеть, что реализация *PAM* из системы *R* уступает реализациям *pgPAM* на вышеуказанных платформах. В рамках каждой платформы предвычисление матрицы Евклидовых расстояний обеспечивает более высокое быстродействие алгоритма *pgPAM*. В силу эффективной векторизации вычислений матрицы расстояний и поиска кластеров алгоритм *pgPAM* показывает более высокое быстродействие на платформах Intel Xeon Phi, чем на центральном процессоре Intel Xeon.

4.6. Выводы по главе 4

В данной главе предложен подход к интеграции интеллектуального анализа данных в реляционную СУБД. Интеллектуальный анализ данных, внедренный в СУБД, позволяет избежать накладных расходов на экспорт данных из СУБД во внешнюю аналитическую систему и импорт результатов анализа обратно в СУБД. Использование аналитических функций в рамках СУБД обеспечивает конечному пользователю полный спектр сервисов, встроенных в СУБД (оптимизация запросов, поиск на основе индексирования данных, управление буферным пулом и др.). В качестве целевой площадки реализации подхода выбрана свободная СУБД PostgreSQL, представлены архитектура и методы реализации предложенного подхода. Предложенный подход базируется на следующих основных идеях.

Параллельные алгоритмы анализа данных для современных многоядерных ускорителей инкапсулируются в хранимых процедурах СУБД. Инкапсуляция параллелизма обеспечивает прозрачное использование аналитических функций прикладным программистом баз данных.

СУБД обеспечивает буферный пул для долговременного хранения в оперативной памяти предварительно вычисляемых структур данных (например, матрица расстояний между объектами подвергаемого кластеризации множества), которые могут быть многократно использованы в дальнейшем аналитическими алгоритмами. Повторное использование предвычисленных структур данных снизит накладные вычислительные расходы при выполнении аналитических алгоритмов.

Результаты интеллектуального анализа данных сохраняются в базе данных в промежуточном (нереляционном) представлении, в качестве которого используется формат JSON. Для каждой базовой задачи анализа данных схема (формальное описание структуры) JSON-документа с результатами анализа предварительно разрабатывается системным программистом и сохраняется в базе данных. Прикладной программист обеспечивается функциями, позволяющими выполнить выборку нужных результатов из промежуточного представления и сохранить эти результаты в виде реляционных таблиц в базе данных. Использование формата JSON обеспечивает возможность сохранения результатов интеллектуального анализа, которые не всегда могут быть вписаны в предопределенную реляционную схему данных, требующую атомарности значений в ячейках реляционных таблиц. JSON-представление может включать в себя описание не только результатов анализа, но также и его метаданных (например, время выполнения алгоритма в различных фазах его работы), что может быть использовано в экспериментальных исследованиях для интеллектуального анализа научных данных.

Предложенный подход предполагает внедрение параллельных алгоритмов интеллектуального анализа данных в реляционную СУБД открытым кодом на основе определяемых пользователем функций. Описана системная архитектура и методы реализации подхода. Приведены результаты вычислительных экспериментов, исследующих эффективность предложенного подхода.

Предложен новый параллельный алгоритм *PBlockwise* вычисления мат-

рицы Евклидовых расстояний для многоядерных ускорителей. *PBlockwise* входит в библиотеку параллельных алгоритмов, которая является частью предложенного подхода, и обеспечивает предвычисление указанной структуры данных, используемой в алгоритмах разделительной кластеризации. Алгоритм *PBlockwise* использует специализированную компоновку данных в памяти и выравнивание данных, обеспечивающие эффективную векторизацию вычислений. Результаты проведенных экспериментов показали близкое к линейному ускорение и параллельную эффективность не ниже 80% алгоритма при совпадении количества используемых нитей с количеством физических ядер ускорителя.

Предложен новый параллельный алгоритм *PPAM* кластеризации данных на основе техники медоидов для многоядерных ускорителей. Данный алгоритм использует в качестве вспомогательного вышеуказанный алгоритм вычисления матрицы расстояний *PBlockwise*. Эффективная векторизация вычислений достигается за счет использования выровненных данных и тайлинга циклов. Результаты проведенных экспериментов показали близкое к линейному ускорение и параллельную эффективность не ниже 80% алгоритма при совпадении количества используемых нитей с количеством физических ядер ускорителя.

Эксперименты, исследующие эффективность предложенных алгоритмов в рамках СУБД, показали, что предвычисление матрицы Евклидовых расстояний обеспечивает более высокое быстродействие, и реализация *PPAM* в PostgreSQL опережает аналог из системы *R*.

Результаты, описанные в этой главе, опубликованы в работах [21, 22, 25, 26, 204–208].

Глава 5. Интеграция в СУБД фрагментного параллелизма

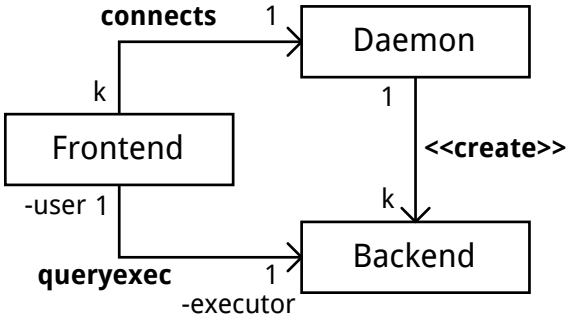
Данная глава посвящена проблеме внедрения фрагментного параллелизма последовательные СУБД с открытым исходным кодом. Описаны архитектура и принципы реализации параллельной СУБД *PargreSQL*, созданной на основе свободной СУБД PostgreSQL. Представлены результаты вычислительных экспериментов, исследующих масштабируемость разработанного решения.

5.1. Архитектура параллельной СУБД на базе PostgreSQL

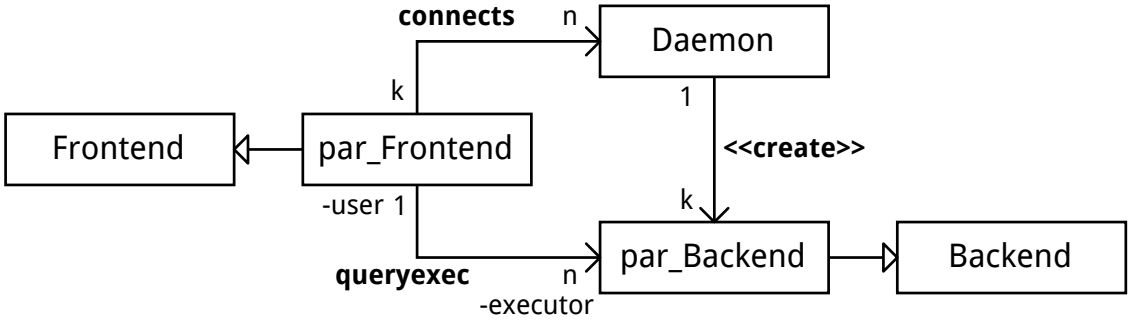
В данном разделе описана архитектура параллельной СУБД, построение которой осуществляется путем внедрения фрагментного параллелизма в последовательную реляционную СУБД с открытым кодом. В качестве целевой последовательной СУБД рассмотрена СУБД PostgreSQL, которая в настоящее время является одной из наиболее надежных и популярных реляционных СУБД с хорошо документированным открытым исходным кодом [84, 194]. Результирующая параллельная СУБД получила название PargreSQL.

5.1.1. Взаимодействие процессов СУБД

В основе архитектуры СУБД PargreSQL лежит модель «клиент-сервер», наследуемая от СУБД PostgreSQL. Взаимодействие процессов СУБД PargreSQL в сравнении с базовой СУБД представлено на рис. 5.1. В сеансе работы с СУБД PostgreSQL участвуют три вида взаимодействующих процессов: *процесс-клиент (frontend)*, *процесс-сервер (backend)* и *демон (daemon)*. Демон выполняет прием соединений от клиентов, создавая



(a) Последовательная СУБД PostgreSQL



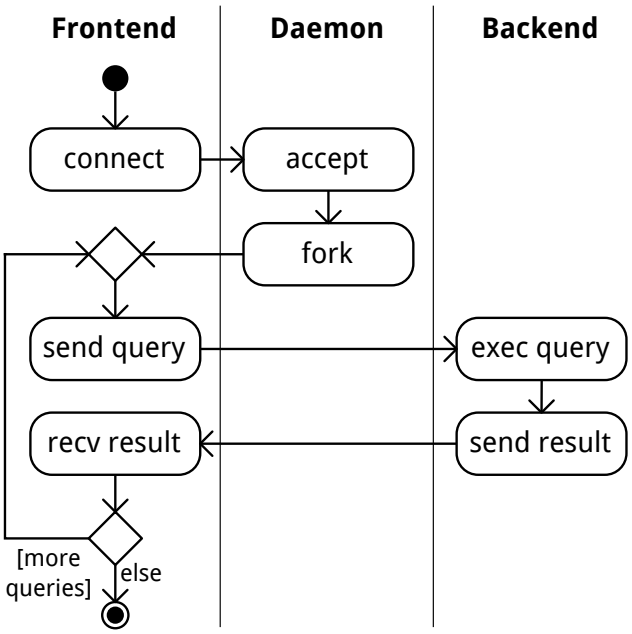
(b) Параллельная СУБД PargreSQL

Рис. 5.1. Процессы СУБД

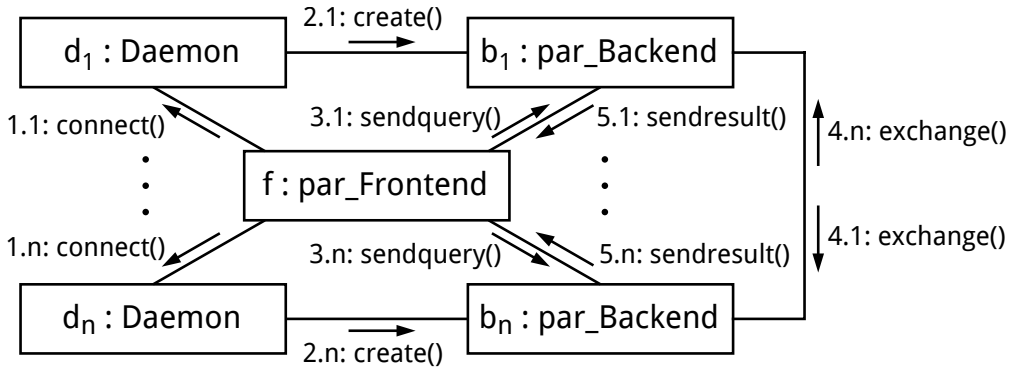
выделенный процесс-сервер для обработки запросов каждого отдельного клиента.

В отличие от последовательной СУБД, в PargreSQL предполагается, что 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* дополняется методами, которые обеспечивают обмены кортежами. В компонент *Frontend* добавляются методы, обеспечивающие тиражирование запросов и работу с несколькими экземплярами компонента *Backend*.

В последовательной СУБД взаимодействие клиента и сервера осуществляется следующим образом (см. рис. 5.2a): клиент выполняет установку соединения с демоном, который после приема соединения с помощью системного вызова `fork()` создает серверный процесс для данного клиента. Далее клиент направляет запрос процессу-серверу, который обрабатывает запрос и отправляет полученные результаты клиенту. В отличие от последовательной СУБД, взаимодействие между сервером СУБД PargreSQL и кли-



(a) Последовательная СУБД PostgreSQL



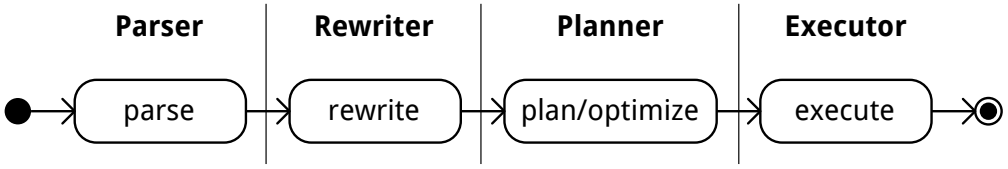
(b) Параллельная СУБД PargreSQL

Рис. 5.2. Клиент-серверное взаимодействие

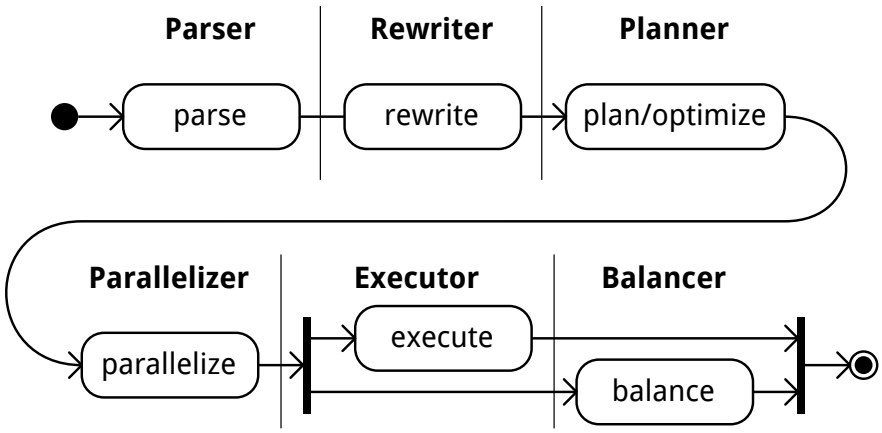
ентским приложением осуществляется следующим образом (см. рис. 5.2b). Клиентское приложение подключается последовательно к каждому демону СУБД, в результате чего на каждом вычислительном узле кластерной системы запускается компонент *par_Backend*. Далее клиент последовательно отправляет каждому из этих компонентов запрос. После получения запроса каждый экземпляр компонента *par_Backend* выполняет его над своим фрагментом базы данных. При выполнении запроса компонент *par_Backend* обменивается данными с другими экземплярами в силу наличия в плане запроса оператора *exchange*. По завершении обработки запроса клиентское приложение получает от экземпляров результаты и выполняет их

слияние в финальный результат.

5.1.2. Обработка запроса



(a) Последовательная СУБД PostgreSQL



(b) Параллельная СУБД PargreSQL

Рис. 5.3. Обработка запроса в СУБД

Схема обработки запроса в СУБД PargreSQL представлена на рис. 5.3. Выполнение этапов *parse*, *rewrite*, *plan/optimize* и *execute* осуществляется так же, как и в последовательной СУБД.

На этапе *parse* осуществляется синтаксический разбор запроса на языке SQL и формирование дерева разбора. На этапе *rewrite* выполняется преобразование дерева разбора в соответствии с правилами, которые заданы администратором базы данных, например, в случае реализации пользовательских представлений базы данных. Этап *plan/optimize* заключается в построении последовательного плана запроса. Результатом данного этапа является физический план запроса, который имеет оптимальную оценку сложности [167]. Этап *execute* предполагает выполнение плана запроса. Поскольку операции в плане запроса имеют итераторный интерфейс, исполнение запроса реализуется посредством последовательного получения

всех кортежей из корня плана запроса и применения к данным кортежам соответствующей операции (вставка, удаление, обновление и др.).

На добавляемом в СУБД PargreSQL этапе *parallelize* выполняется формирование параллельного плана запроса (см. описание реализации параллелизатора в разделе 5.2.3).

На этапе *balance* выполняется балансировка загрузки серверных процессов [49], которая осуществляется одновременно с исполнением запроса.

5.1.3. Модульная структура

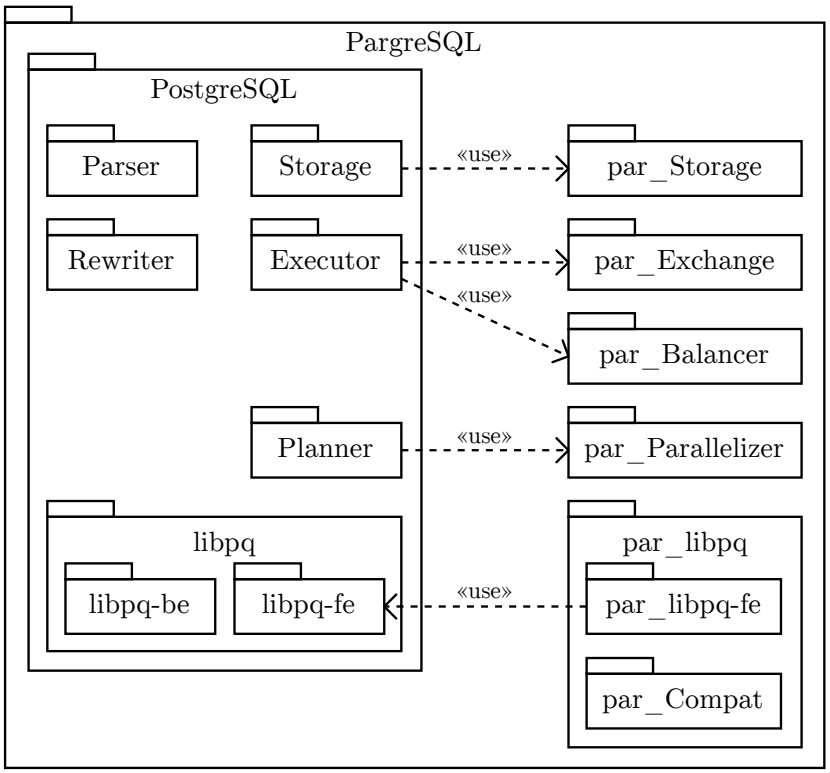


Рис. 5.4. Архитектура СУБД PargreSQL

Модульная структура СУБД PargreSQL представлена на рис. 5.4. Оригинальная СУБД PostgreSQL рассматривается *как подсистема* в рамках СУБД PargreSQL. СУБД PostgreSQL содержит следующие подсистемы:

— `Parser` — подсистема, которая осуществляет синтаксический разбор SQL-запросов;

— `Rewriter` — подсистема, выполняющая преобразование запроса в соответствии с правилами подстановки, которые хранятся в базе данных (например, для реализации представлений);

— `Storage` — подсистема, отвечающая за хранение данных и метаданных;

— `Planner` — подсистема, которая выполняет построение плана запроса;

— `Executor` — подсистема, которая реализует исполнение плана запроса;

— `libpq` — библиотека, реализующая протокол взаимодействия клиента (подсистема `libpq-fe`) и сервера (подсистема `libpq-be`).

Для реализации СУБД PargreSQL необходимо внести изменения в исходный код следующих подсистем СУБД PostgreSQL: `Storage`, `Executor` и `Planner`. Данные изменения обеспечивают внедрение следующих новых подсистем:

— *par_Storage* — подсистема, обеспечивающая хранение и обработку метаданных о фрагментации отношений;

— *par_Exchange* — подсистема, реализующая оператор *exchange* [222], который обеспечивает пересылку кортежей между экземплярами СУБД во время выполнения запроса;

— *par_Parallelizer* — подсистема, выполняющая добавление в нужные места последовательного плана запроса операторов *exchange*;

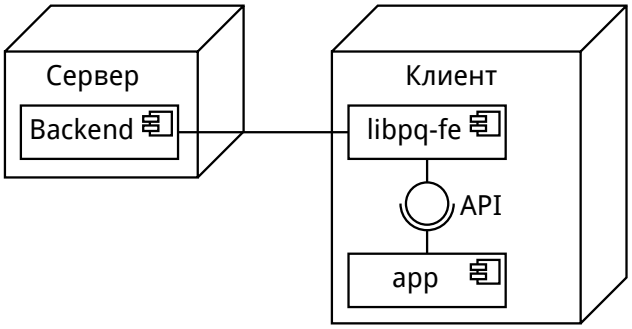
— *par_Balancer* — подсистема, выполняющая динамическую балансировку загрузки серверных процессов.

СУБД PargreSQL включает в себя следующие *новые подсистемы*, разработка которых не предполагает внесение изменений в исходные тексты СУБД PostgreSQL:

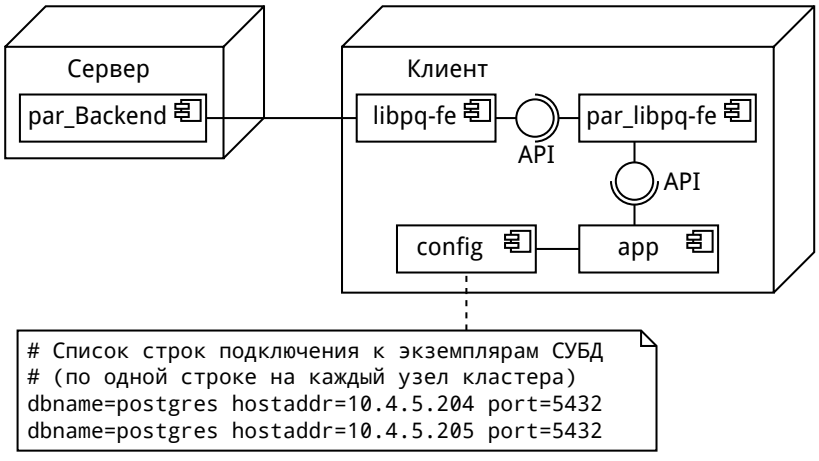
— *par_ libpq-fe* — надстройка над стандартной библиотекой `libpq-fe` СУБД PostgreSQL, реализующая тиражирование запроса (отправку запроса множеству серверов, на которых запущены экземпляры СУБД PargreSQL);

— *par_ Compat* — подсистема портирования, которая реализует прозрачное подключение библиотеки *par_ libpq-fe* к пользовательскому приложению и обеспечивает использование параллельной СУБД PargreSQL вместо оригинальной последовательной СУБД PostgreSQL без внесения существенных изменений в исходный код приложения.

5.1.4. Развертывание компонентов



(a) Последовательная СУБД PostgreSQL



(b) Параллельная СУБД PargreSQL

Рис. 5.5. Развертывание компонентов СУБД

Развертывание компонентов СУБД представлено на рис. 5.5. В СУБД PostgreSQL на клиенте размещается библиотека `libpq` и приложение поль-

зователя, а остальные компоненты размещаются на сервере. В СУБД PargreSQL на клиенте размещаются библиотеки *par_libpq* и `libpq-fe`, а компонент *Backend* размещается на каждом вычислительном узле-сервере кластерной системы.

В СУБД PargreSQL прикладной программист подключает к своему приложению библиотеки *par_libpq* и `libpq-fe` и хранит на клиенте конфигурационный файл с параметрами доступа к вычислительным узлам кластерной системы (ip-адрес, порт, имя пользователя, пароль, имя базы данных и др.).

5.2. Методы интеграции параллелизма в реляционную СУБД на примере PostgreSQL

Данный раздел содержит описание основных методов, на которых базируется реализация параллельной СУБД PargreSQL, полученная путем внедрения фрагментного параллелизма в свободную СУБД PostgreSQL.

5.2.1. Подсистема тиражирования

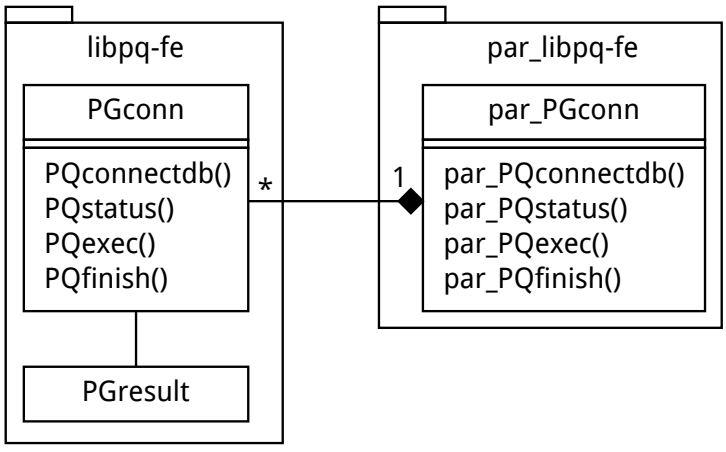


Рис. 5.6. Классы, реализующие подсистему тиражирования

Диаграмма классов, реализующих подсистему тиражирования, показана на рис. 5.6. Подсистема тиражирования *par_libpq-fe* представляет собой

надстройку над оригинальной библиотекой `libpq-fe` СУБД PostgreSQL и выполняет отправку запроса множеству серверов, на которых запущены экземпляры СУБД PargreSQL. Подсистема *par_ libpq-fe* имеет интерфейс, аналогичный интерфейсу `libpq-fe`.

Табл. 5.1. Сравнение API

API PostgreSQL (`libpq-fe`)	**API PargreSQL** (***par_ libpq-fe***)
`struct PGconn` Хранит данные о соединении с сервером СУБД PostgreSQL.	`struct par_PGconn` Хранит данные о соединениях с экземплярами СУБД PargreSQL. Представляет собой массив структур `PGconn`.

`PGconn *PQconnectdb(`
`const char *conninfo`
`)`
Устанавливает соединение с сервером СУБД PostgreSQL, указанным в строке подключения `conninfo`.
	`par_PGconn *par_PQconnectdb(`
	`void`
	`)`
	Устанавливает соединение с экземплярами СУБД PargreSQL, указанными в конфигурационном файле.

`ConnStatusType PQstatus(`
`const PGconn *conn`
`)`
Возвращает статус соединения с сервером СУБД PostgreSQL.
	`ConnStatusType par_PQstatus(`
	`const par_PGconn *conn`
	`)`
	Возвращает статус соединения с СУБД PargreSQL. Результат является агрегацией результатов вызова функции `PQstatus()` всеми экземплярами.

`PGresult *PQexec(`
`PGconn *conn,`
`const char *query`
`)`
Отправляет запрос серверу СУБД PostgreSQL.
	`PGresult *par_PQexec(`
	`PGconn *conn,`
	`const char *query`
	`)`
	Отправляет запрос каждому экземпляру СУБД PargreSQL посредством вызова `PQexec()`.

`void PQfinish(`
`PGconn *conn`
`)`
Завершает соединение с сервером СУБД PostgreSQL.
	`void par_PQfinish(`
	`par_PGconn *conn`
	`)`
	Завершает соединения с экземплярами СУБД PargreSQL.

В табл. 5.1 приведены основные отличия интерфейсов прикладного программирования *par_ libpq-fe* от `libpq-fe`.

5.2.2. Оператор обмена (*exchange*)

Структура оператора *exchange*

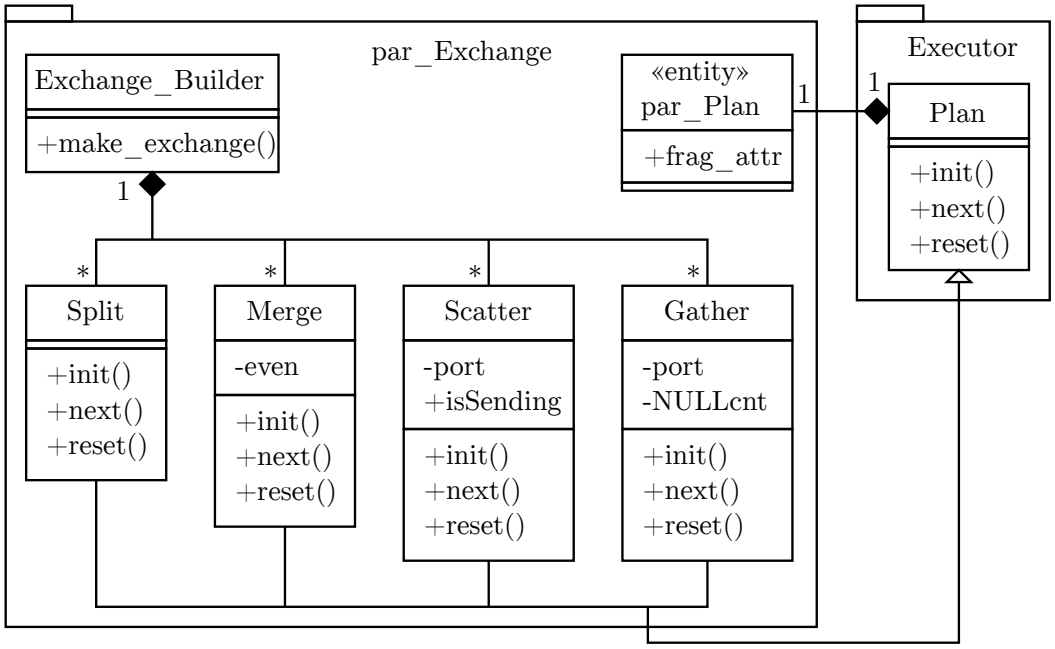


Рис. 5.7. Диаграмма классов оператора *exchange*

Реализация оператора *exchange* связана с добавлением в исходный код оригинальной СУБД PostgreSQL новых функций и типов данных. Пакет *par_Exchange*, содержащий новые классы, добавляемые в СУБД PostgreSQL, представлен на рис. 5.7.

Данный пакет содержит классы *Merge*, *Split*, *Scatter* и *Gather*, которые реализуют одноименные операторы-составные части *exchange*. Класс *Exchange_Builder* представляет собой класс-строитель, создающий указанные узлы параллельного плана запроса в виде цельного оператора *exchange*.

Хранение атрибута фрагментации реляционного отношения обеспечивается добавлением в класс *Plan*, который в СУБД PostgreSQL реализует узел плана запроса, целочисленного атрибута `frag_attr`.

Реализация метода `next` класса *Split* представлена на рис. 5.8. Узел *Split* вызывает метод `next` левого сына и вычисляет значение функции пересылки от результирующего кортежа. Если функция пересылки возвращает значение, которое совпадает с номером текущего вычислительного

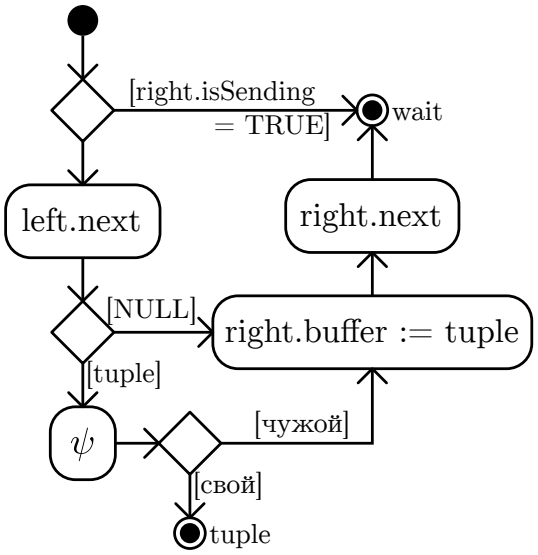


Рис. 5.8. Метод `next` класса *Split*

узла кластерной системы (результирующий кортеж является «своим»), то узел *Split* завершает работу и возвращает значение данного кортежа в качестве результата. Иначе результирующий кортеж является «чужим» (должен быть отправлен соответствующему вычислительному узлу кластерной системы) и помещается в буфер правого сына (узел *Scatter*), выполняется вызов метода `next` узла *Scatter*, и оператор *exchange* переводится в состояние ожидания.

На рис. 5.9 представлена реализация метода `next` класса *Merge*. Узел *Merge* попеременно вызывает методы `next` своего левого и правого сыновей — узлов *Gather* и *Split*. Вызовы осуществляются до тех пор, пока оператор *exchange* находится в состоянии ожидания. Если оба сына вернули пустое значение `NULL`, то входной поток кортежей пуст, и узел *Merge* завершает работу, возвращая значение `NULL`. Если хотя бы один из сыновей возвратил кортеж в качестве результата, то происходит завершение работы узла *Merge*, и данный кортеж возвращается как результат работы.

Реализация метода `next` класса *Scatter* показана на рис. 5.10. Узел *Scatter* не имеет сыновей, и вызов метода `next` запускает отправку кортежа, который был размещен оператором *Split* в его входном буфере, на тот вычислительный узел кластерной системы, номер которого возвратит

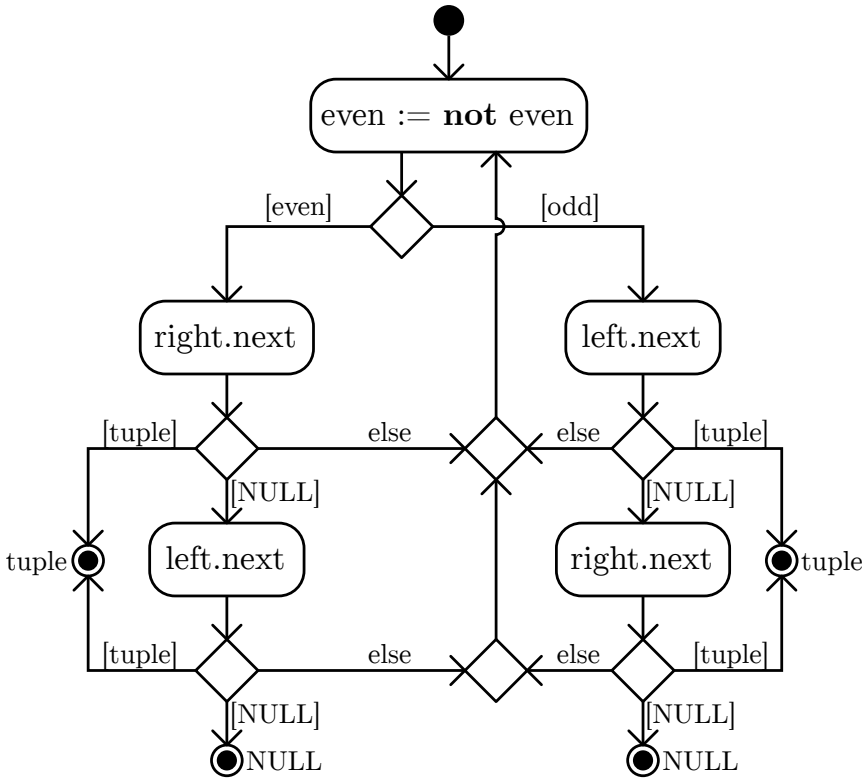


Рис. 5.9. Метод `next` класса *Merge*

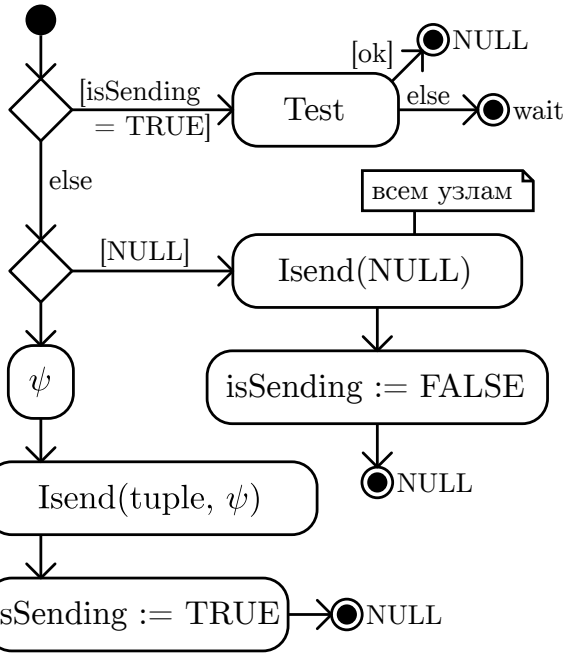


Рис. 5.10. Метод `next` класса *Scatter*

функция пересылки, примененная к данному кортежу. Если во время вызова метода `next` предыдущая отправка кортежа не завершена, то возвращается значение `WAIT`.

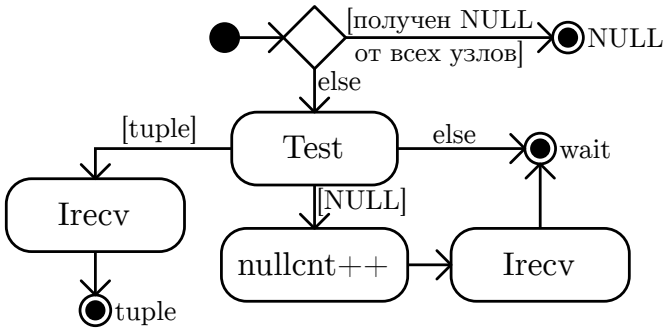


Рис. 5.11. Метод `next` класса *Gather*

Реализация метода `next` класса *Gather* (см. рис. 5.11) выглядит следующим образом. Оператор *Gather* инициирует получение кортежей от всех вычислительных узлов кластерной системы. При вызове метода `next` данного оператора проверяются статусы операций получения, и если получен кортеж от некоторого вычислительного узла, то запускается новая операция получения кортежа от данного узла, а полученный кортеж возвращается в качестве результата. Если от всех кортежей получено значение `NULL`, значит, обрабатываемое отношение исчерпано, и метод возвращает в качестве результата `NULL` как признак конца данных.

Менеджер сообщений

Архитектура СУБД PargreSQL предполагает (см. раздел 5.1.1), что серверные процессы должны порождаться динамически. В соответствии с этим для реализации обменов данными стандартной технологии параллельного программировании для систем с распределенной памятью MPI (Message Passing Interface, интерфейс обмена сообщениями) [96] требует определенной модификации.

В СУБД PargreSQL для организации обменов сообщениями, реализующих описанные выше операторы *scatter* и *gather*, на основе MPI разработан отдельный *менеджер сообщений*. Менеджер сообщений имеет сходный с MPI интерфейс, представленный на рис. 5.12.

Менеджер сообщений состоит из двух частей: демон и библиотека. *Демон* представляет собой MPI-программу, которая запускается независимо

```
1  // Выдать номер текущего вычислительного узла
2  int CommRank();
3
4  // Выдать количество вычислительных узлов
5  int CommSize();
6
7  // Запустить операцию отправки сообщения.
8  // Параметры:
9  // dest — номер узла–получателя
10 // port — порт операции
11 // size — длина сообщения
12 // buf — указатель на буфер с отправляемым сообщением
13 // request — дескриптор операции.
14 // Возвращаемое значение: 0 в случае успеха или отрицательный код ошибки.
15 int Isend(dest, port, size, *buf, *request);
16
17 // Запустить операцию приема сообщения.
18 // Параметры:
19 // port — порт операции
20 // size — длина сообщения
21 // buf — указатель на буфер с принятым сообщением
22 // request — дескриптор операции.
23 // Возвращаемое значение: 0 в случае успеха или отрицательный код ошибки.
24 int Irecv(src, port, size, *buf, *request);
25
26 // Проверить завершение заданной операции приема или отправки сообщения.
27 // request — дескриптор операции
28 // flag — флаг завершения операции (TRUE или FALSE).
29 // Возвращаемое значение: 0 в случае успеха или отрицательный код ошибки.
30 int Test(request, *flag);
```

Рис. 5.12. Интерфейс библиотеки обмена сообщениями

от СУБД PargreSQL в одном экземпляре на каждом вычислительном узле кластерной системы. *Библиотека* предоставляет серверным процессам СУБД PargreSQL интерфейс для подключения к демону через общую память и организует обмен сообщениями.

5.2.3. Параллелизатор плана запроса

Подсистема *параллелизатор (par_Parallelizer)* обеспечивает создание параллельного плана запроса путем вставки операторов *exchange* в последовательный план запроса. Параллелизатор осуществляет концевой обход дерева последовательного плана и выполняет вставку оператора *exchange* ниже узла с операцией соединения отношений, если атрибут фрагментации

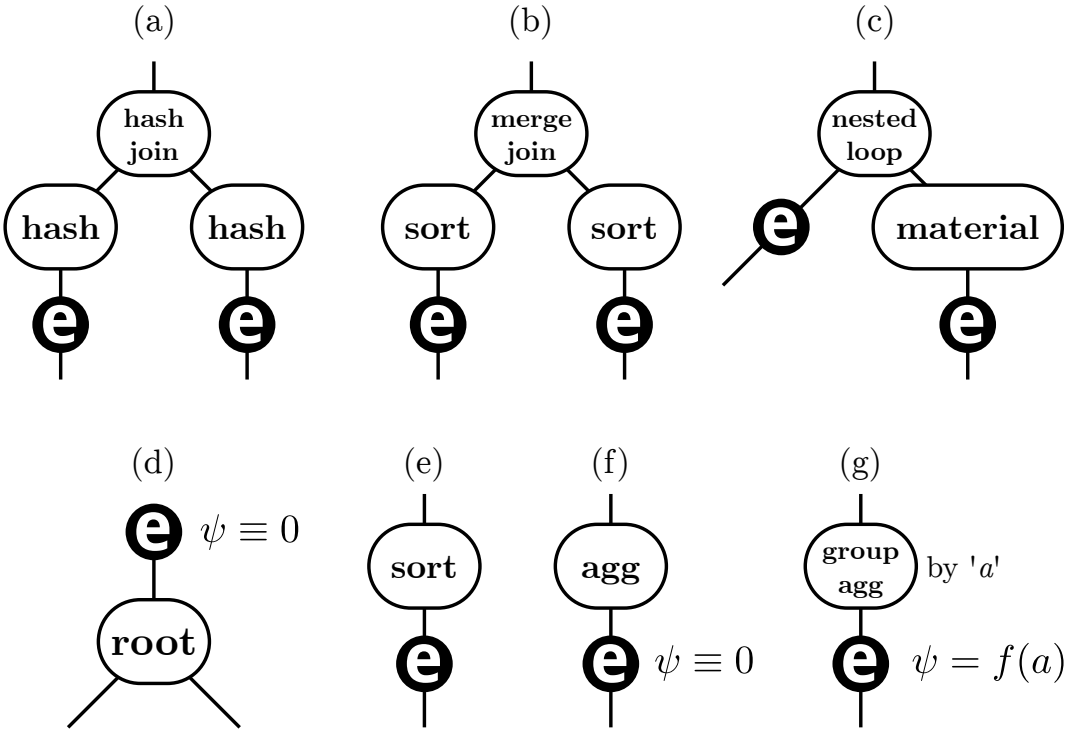


Рис. 5.13. Вставка оператора *exchange*

сына не совпадает с атрибутом, по которому производится соединение [222]. При этом атрибут фрагментации распространяется по дереву от листьев к корню, поэтому в каждом узле плана запроса известно, по какому атрибуту фрагментирован результат операции. Различные случаи вставки оператора обмена в последовательный план запроса показаны на рис. 5.13.

При формировании плана запроса в СУБД PostgreSQL используются следующие типы узла, реализующего операцию соединения: соединение хешированием (HashJoin) [260], соединение слиянием (MergeJoin) [262] и соединение вложенными циклами (NestedLoop) [261]. Вставка оператора обмена в каждом из этих случаев имеет следующие особенности.

Операция HashJoin предполагает формирование хеш-таблицы для каждого из отношений-аргументов операции соединения. 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. 5.13а), чтобы создание хеш-таблицы осуществлялось *после* того, как будут получены кортежи, отправленные

другими вычислительными узами с помощью оператора *exchange* на текущий вычислительный узел кластерной системы.

Операция MergeJoin предполагает предварительное выполнение сортировки отношений-аргументов операции соединении сыновьями *Sort* узла *MergeJoin*. Вставка оператора обмена осуществляется *между* узлом *Sort* и его поддеревом (см. рис. 5.13b), чтобы сортировка осуществлялась *после* получения на текущем вычислительном узле кортежей от других вычислительных узлов кластерной системы.

Операция NestedLoop предполагает, что правое отношение-аргумент операции целиком размещено в оперативной памяти для осуществления его многократного сканирования во внутреннем цикле соединения. Правым сыном узла *NestedLoop* является узел *Material*, который выполняет загрузку данных своего сына в память. Вставка оператора обмена осуществляется *между* узлом *Material* и его поддеревом (см. рис. 5.13c), чтобы загрузка данных в память выполнялась *после* того, как будут получены кортежи, отправленные другими вычислительными узами с помощью оператора *exchange* на текущий вычислительный узел кластерной системы. В противном случае пересылка кортежей правого отношения будет осуществляться столько раз, сколько кортежей в левом отношении. Это приводит к взаимной блокировке серверных процессов, если фрагменты левого отношения на разных вычислительных узлах кластерной системы имеют различное количество кортежей.

На рис. 5.13d 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 узле-координаторе кластерной системы (см. раздел 1.2.2). Оператор обмена, вставляемый в корень плана запроса, имеет функцию пересылки, которая тождественна номеру вычислительного узла кластерной системы, объявленного координатором.

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 выполняет сортировку кортежей, поступающих из поддерева и оператор *exchange* может нарушить порядок кортежей, будучи размещенным на уровень выше. В связи с этим в подобных случаях (см. рис. 5.13e) оператор *exchange* вставляется в качестве дочернего, а не родительского узла *Sort*, чтобы выполнить сортировку кортежей после обменов.

Операция Agg используется для вычисления агрегатных функций `MIN()`, `MAX()`, `SUM()` и др. в запросах на выборку без группировки. Поскольку операция агрегации должна обработать кортежи, находящиеся во всех фрагментах отношения, то для получения корректных результатов потомком узла *Agg* вставляется оператор *exchange* (см. рис. 5.13f), в котором функция пересылки возвращает номер вычислительного узла-координатора, обеспечивая пересылку всех кортежей на вычислительный узел-координатор и корректное вычисление агрегатной функции.

Операция GroupAgg используется для вычисления агрегатных функций в запросах на выборку с группировкой кортежей. В отличие от предыдущего случая для получения корректного результата необходимо обработать каждую группу целиком. Для этого в качестве потомка узла *GroupAgg* вставляется оператор *exchange* с функцией пересылки, зависящей от атрибута группировки (см. рис. 5.13g). Это обеспечивает пересылку всех кортежей из одной группы на один узел и, соответственно, корректное вычисление агрегатной функции для каждой группы.

5.2.4. Обработка запросов на изменение данных

Рассмотренная в разделе 1.2.2 схема построения параллельного плана предназначена для исполнения запросов на выборку кортежей из отношений, фрагменты которых распределены по вычислительным узлам кластерной системы. Для корректного выполнения запросов на вставку

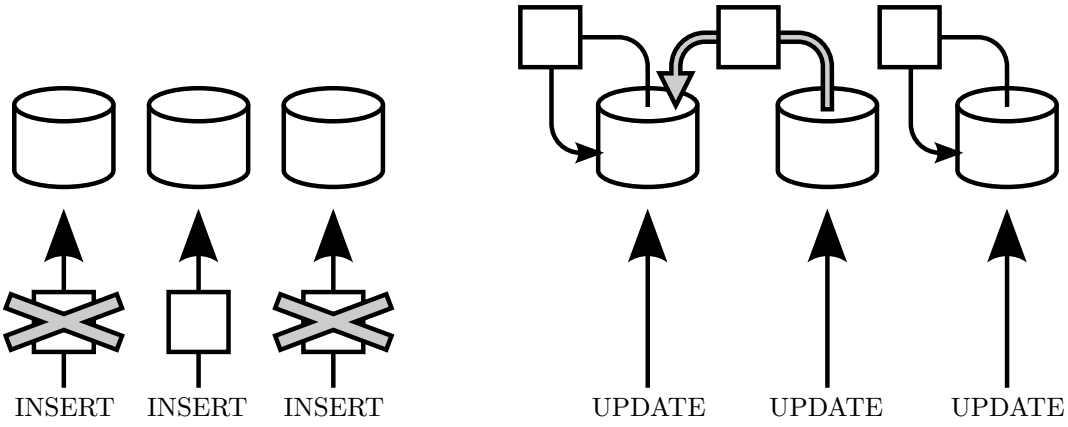


Рис. 5.14. Вставка и обновление кортежа в СУБД PargreSQL

и обновление кортежей данная схема должна быть модифицирована (см. рис. 5.14).

При выполнении запроса `INSERT` необходимо обеспечить вставку кортежа только в один из фрагментов (номер которого совпадает с значением, которое возвращает функция фрагментации для данного кортежа), 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 на соответствующий вычислительный узел.

Алгоритмы, приведенные в разделе 5.2.2, предназначены для запросов на выборку данных (результирующие кортежи пересылаются клиенту) и нуждаются в модификации для запросов на вставку и обновление данных. В случае запроса `INSERT` результирующий кортеж должен быть вставлен в соответствующую таблицу, в случае запроса `UPDATE` необходимо выполнить удаление старой версии кортежа и вставку новой версии.

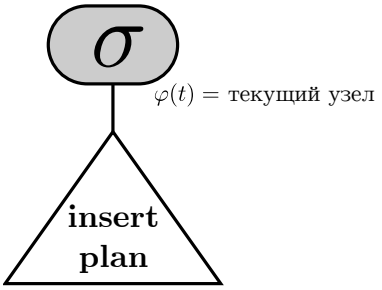


Рис. 5.15. Параллельный план запроса `INSERT`

При обработке *запросов на вставку данных* алгоритмы модифицируются следующим образом (см. рис. 5.15). В корень параллельного плана запроса вставляется узел, представляющий собой операцию выборки с условием $\varphi(t) = mynode$, где φ — функция фрагментации, $mynode$ — номер текущего вычислительного узла кластерной системы. Данное условие обеспечит отбрасывание кортежей, которые должны быть вставлены в фрагменты указанной таблицы базы данных, хранящихся на вычислительных узлах кластерной системы, отличных от текущего.

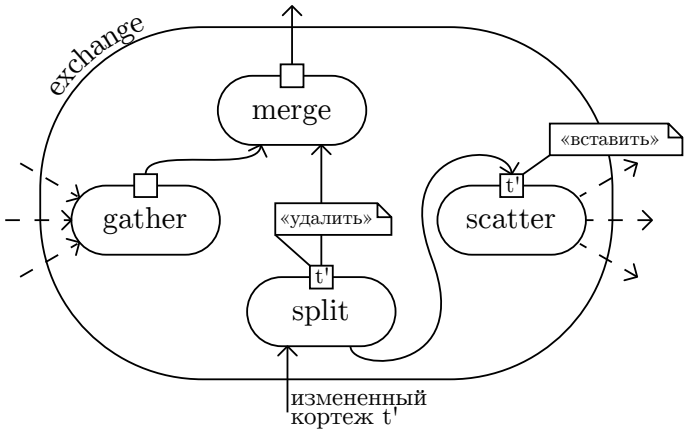


Рис. 5.16. Поток кортежей в операторе обмена при запросе `UPDATE`

Для обработки *запросов на обновление данных* оператор обмена модифицируется следующим образом (см. рис. 5.16). Оператор обмена обнаруживает кортежи, у которых изменилось значение атрибута фрагментации, и создает копию таких кортежей. После этого один из этих экземпляров передается далее по плану с пометкой «удалить», второй экземпляр кортежа передается на соответствующий вычислительный узел кластерной системы с пометкой «вставить».

Таким образом, модифицированный алгоритм позволяет корректно обновлять кортежи, которые в результате изменения у них значения атрибута фрагментации стали «чужими»: если $\varphi(t') \neq \varphi(t)$, то на вычислительном узле с номером $\varphi(t)$ кортеж t удаляется, а на вычислительном узле с номером $\varphi(t')$ вставляется кортеж t'.

5.2.5. Хранение 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.

Для организации хранения данных о фрагментации в словаре параллельной СУБД PargreSQL в СУБД PostgreSQL в метаданные таблицы вводится новый атрибут *fragattr*, который имеет строковый тип и задает имя атрибута фрагментации данной таблицы.

```
1 CREATE TABLE Person (
2   id int,
3   name varchar(30),
4   gender char(1),
5   birth date
6 ) WITH (fragattr = id);
```

Рис. 5.17. Создание таблицы в PargreSQL

Атрибут *fragattr* указывается при создании таблицы с помощью конструкции `WITH`. Пример соответствующего запроса `CREATE TABLE` приведен на рис. 5.17. Атрибут с именем, указанным в параметре *fragattr* таблицы, определяется на домене целых неотрицательных чисел и используется при обработке запросов на обновление и вставку данных, обеспечивая распределение данных по вычислительным узлам кластерной системы в соответствии с функцией фрагментации $\varphi(t) = t.\text{fragattr} \bmod N$, где N — количество узлов в кластерной системе, mod — операция взятия остатка от целочисленного деления.

5.2.6. Прозрачное портирование приложений

```
1 // postgresapp.c
2 #include <libpq-fe.h>
3 void main(void)
4 {
5   PGconn c = PQconnectdb(...);
6   PGresult r = PQexec(c, ...);
7   ...
8   PQfinish(c);
9 }
```

(a) Приложение PostreSQL

```
1 // pargresapp.c
2 #include <par_libpq-fe.h>
3 void main(void)
4 {
5   PGconn c = PQconnectdb(...);
6   PGresult r = PQexec(c, ...);
7   ...
8   PQfinish(c);
9 }
```

(b) Приложение PargreSQL

```
1 // par_Compat.h.c
2 #define PQconnectdb(...) \
3   par_PQconnectdb(...)
4 #define PGconn \
5   par_PGconn
6 ...
```

(c) Подсистема *par_ Compat*

Рис. 5.18. Перенос приложений PostgreSQL в PargreSQL

Подсистема портирования *par_ Compat* позволяет использовать библиотеку *par_ libpq-fe* с интерфейсом оригинальной библиотеки `libpq-fe`. Подсистема портирования реализуется в виде набора макроподстановок компилятора, которые позволяют минимизировать изменения в исходных текстах пользовательского приложения для СУБД PostgreSQL при портировании приложений в параллельную СУБД PargreSQL. Данный набор макросов заменяет функции оригинальной библиотеки `libpq-fe` на вызовы соответствующих функций подсистемы тиражирования *par_ libp-fe*.

Таким образом, для переноса PostgreSQL-приложения в СУБД PargreSQL в исходном тексте приложения требуется изменение одной строки кода, выполняющей подключение библиотеки (см. рис. 5.18).

```
1  // origfile.c
2  #include "newfile.c"
3  typedef struct origstruct {
4    ...
5    newstruct ns;
6  } origstruct;
7
8  // newfile.c
9  typedef struct newstruct {
10   ...
11 } newstruct;
```

(a) Добавление полей в структуру

```
1  // origfile.c
2  #include "newfile.c"
3  int origfunc() {
4    ...
5    newfunc();
6    ...
7  }
8
9  // newfile.c
10 int newfunc()
11 { ... }
```

(b) Добавление оператора вызова в функцию

Рис. 5.19. Внесение изменений в исходный код последовательной СУБД

5.2.7. Мягкая модификация исходных текстов

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

Предлагаемый способ 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. 5.19a показан пример применения данного метода для добавления новых полей в оригинальную структуру данных. В новом файле исходных текстов определяется тип `newstruct`, содержащий новые поля, а в оригинальную структуру добавляется новое поле, имеющее тип данных `newstruct`. На рис. 5.19b показан пример применения данного метода для изменения оригинальных алгоритмов. В тело оригинальной функции добавляется вызов новой функции `newfunc()`, при этом функция `newfunc()` определяется в файле исходных текстов новой подсистемы.

5.3. Вычислительные эксперименты

Цели и аппаратная платформа экспериментов

Целью вычислительных экспериментов являлась оценка эффективности предложенных методов и алгоритмов, реализованных в параллельной СУБД PargreSQL, при обработке данных на платформе многопроцессорной вычислительной системы с кластерной архитектурой. В соответствии с этим были проведены эксперименты, в которых исследована масштабируемость СУБД PargreSQL и выполнено сравнение производительности СУБД PargreSQL с производительностью аналогичных систем.

Масштабируемость означает адекватное увеличение производительности при добавлении в систему дополнительных процессоров, модулей памяти дисков и других аппаратных компонент. *Сравнение производительности* предполагает выполнение СУБД PargreSQL и другими системами некоторого эталонного теста на одной и той же аппаратной платформе. В качестве эталонного теста использовалась спецификация стандартного теста производительности TPC-C, разработанных консорциумом TPC (Transaction Processing Council) [165].

Тест TPC-C измеряет производительность СУБД в ходе обработки смеси коротких транзакций, осуществляя моделирование деятельности типичного склада (прием заказов, управлением учетом и доставкой товаров и др.). В качестве меры производительности в тесте TPC-C используется коммерческая пропускная способность, отражающая количество обработанных в минуту заказов. Мера производительности выражается двумя показателями: пиковая скорость выполнения транзакций tpm-C (transactions-per-minute-C, количество транзакций в минуту) и нормализованная стоимость системы $/tpm-C. Стоимость системы включает все аппаратные средства и программное обеспечение, используемые в тесте, и стоимость обслуживания оборудования в течение пяти лет.

Основными качественными характеристиками масштабируемости па-

раллельной СУБД являются ускорение и расширяемость, определяемые следующим образом [23].

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 $a_{AB} = \frac{t_{QA}}{t_{QB}}$, 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 $Q_1, Q_2, \ldots$, количественно превосходящих некоторый фиксированный тест Q в i раз, где i — номер соответствующего теста и конфигурации параллельной машины баз данных $A_1, A_2, \ldots$, 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 $e_{km} = \frac{t_{Q_k A_k}}{t_{Q_m A_m}}$. Расширяемость позволяет измерить эффективность наращивания системы на бóльших задачах.

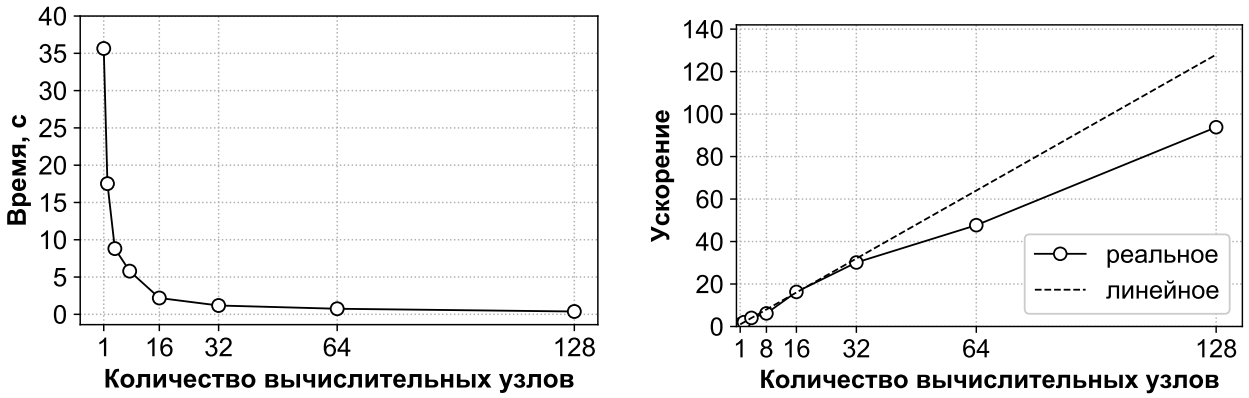
Говорят, что параллельная система хорошо масштабируема, если она демонстрирует ускорение и расширяемость, близкие к линейным. *Линейное ускорение* означает, что существует константа $k > 0$, что $a_{AB} = k \cdot \frac{d_B}{d_A}$ для любых конфигураций A и B (где d — количество процессоров в соответствующей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

Эксперименты проводились на суперкомпьютере «СКИФ-Аврора ЮУрГУ» [6], характеристики которого представлены в табл. 5.2.

Табл. 5.2. Аппаратная платформа экспериментов

Характеристика	Значение
Кол-во выч. узлов/процессоров/ядер	736/1 472/8 832
Тип процессора	Intel Xeon X5680
Оперативная память, Тб	3
Пиковая производительность, TFLOPS	117
Производительность LINPACK, TFLOPS	100.4

Результаты экспериментов



(a) Время (b) Ускорение

Рис. 5.20. Ускорение СУБД PargreSQL

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. 5.20. Можно видеть, что СУБД PargreSQL демонстрирует ускорение, близкое к линейному: от 75% до 100% от линейного.

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

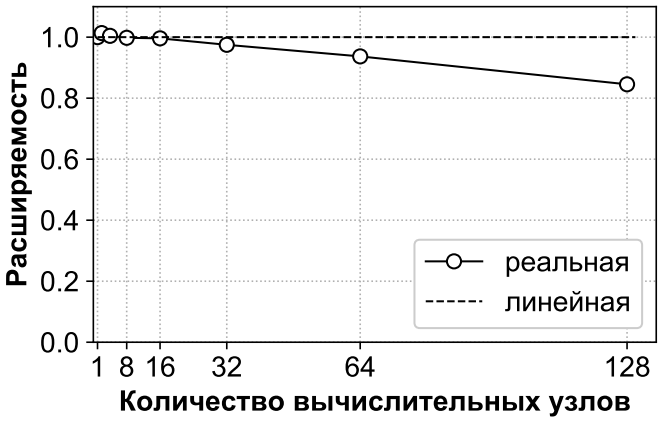


Рис. 5.21. Расширяемость СУБД PargreSQL

(т.е. при использовании 128 узлов осуществлялось соединение отношений из 1 536 млн. и 38.4 млн. кортежей соответственно). Результаты данных экспериментов представлены на рис. 5.21. Эксперименты показывают, что расширяемость СУБД PargreSQL близка к линейной: от 85% до 100% от линейной.

Табл. 5.3. Результаты теста TPC-C

К-во клиентов	tpm-C ↓	К-во клиентов	tpm-C ↓	К-во клиентов	tpm-C ↓	К-во клиентов	tpm-C ↓
29	2 202 531	24	2 165 413	16	1 882 353	8	1 156 626
26	2 197 183	23	2 156 250	15	1 747 572	7	1 150 684
30	2 195 122	22	2 146 341	14	1 647 058	5	857 142
32	2 194 285	20	2 068 965	13	1 529 411	6	847 058
27	2 189 189	19	2 054 054	12	1 358 490	4	657 534
31	2 188 235	18	2 037 735	11	1 346 938	3	444 444
28	2 181 818	21	2 016 000	10	1 290 322	2	328 767
25	2 173 913	17	1 961 538	9	1 270 588	1	150 000

Результаты исследования эффективности СУБД PargreSQL на тесте TPC-C приведены в табл. 5.3 в порядке убывания показателя tpm-C. В тесте использовалось от 1 до 30 параллельно работающих клиентов, выполняющих запросы к СУБД PargreSQL, запущенной на 12 узлах. Размер

базы данных составлял 12 «складов». Данный результат позволяет СУБД PargreSQL попасть в пятерку лидеров рейтинга TPC-C среди параллельных СУБД для кластеров на сентябрь 2013 г. (см. табл. 5.4).

Табл. 5.4. Лидеры рейтинга TPC-C среди параллельных СУБД

№ п/п	Кластерная система	СУБД	К-во узлов	К-во 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
	Торнадо 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

5.4. Выводы по главе 5

В данной главе описаны новые методы построения параллельной СУБД для кластерных вычислительных систем путем внедрения концепции фрагментного параллелизма в свободную СУБД. Данные методы применяется к свободной СУБД PostgreSQL, на основе которой создается параллельная СУБД PargreSQL. В рамках представленных методов последовательная СУБД рассматривается как подсистема параллельной СУБД. Методы предполагают модификацию подсистем оригинальной СУБД и разработку новых подсистем.

Изменениям подвергаются следующие основные подсистемы: подсистема хранения данных, подсистема построения плана запроса и исполнитель

запросов. Изменения в подсистеме хранения данных обеспечивают хранение метаданных о фрагментации отношений в словаре СУБД.

Изменения в подсистеме построения плана запроса обеспечивают построение параллельного плана запроса путем вставки в нужные места последовательного плана операторов *exchange*. Оператор *exchange* инкапсулирует параллелизм, реализуя обмены кортежами между экземплярами СУБД во время выполнения запроса.

Новыми подсистемами, разработка которых необходима для внедрения фрагментного параллелизма в последовательную СУБД, являются подсистема тиражирования запроса и подсистема портирования исходных текстов пользовательских программ. Подсистема тиражирования запроса реализует отправку запроса множеству серверов, на которых запущены экземпляры параллельной СУБД. Подсистема портирования реализует прозрачное подключение подсистемы тиражирования запроса к пользовательским приложениям, написанным для последовательной СУБД. Разработка указанных подсистем не предполагает модификацию исходных текстов подсистем оригинальной СУБД.

Описаны алгоритмы реализации оператора обмена и методы его использования при выполнении запросов на выборку, вставку и обновление данных.

Проведены вычислительные эксперименты по исследованию ускорения, расширяемости и производительности параллельной СУБД *PargreSQL*, результаты которых подтвердили эффективность предложенных методов и алгоритмов.

Результаты, описанные в этой главе, опубликованы в работах [12, 15, 17–19, 186, 188]. На разработанное программное обеспечение получены свидетельства Роспатента о государственной регистрации программ [8, 24].

Заключение

Итоги исследования

В диссертационной работе были рассмотрены вопросы интеграции методов интеллектуального анализа данных в реляционные системы баз данных, и разработки параллельных алгоритмов интеллектуального анализа данных для кластерных вычислительных систем с узлами на базе современных многоядерных ускорителей.

Разработан подход к интеграции интеллектуального анализа данных в реляционную СУБД, предполагающий встраивание в СУБД аналитических алгоритмов, которые инкапсулируют параллельное исполнение на современных многоядерных ускорителях. Описана системная архитектура и методы реализации подхода для свободной СУБД PostgreSQL и многоядерных ускорителей архитектуры Intel Many Integrated Core (MIC). Предложен новый параллельный алгоритм кластеризации данных на основе техники медоидов для многоядерных ускорителей, включаемый в библиотеку параллельных алгоритмов, которая является частью предложенного подхода. Проведены вычислительные эксперименты, подтверждающие масштабируемость разработанных алгоритмов и эффективность предложенного подхода.

Разработан метод интеграции фрагментного параллелизма в последовательную свободную СУБД посредством модернизации ее исходных кодов, подразумевающей отсутствие масштабных изменений в реализации существующих подсистем. Метод позволяет получить эффективное и относительно недорогое решение для организации хранения и обработки сверхбольших объемов данных, обладающее хорошей масштабируемостью. На основе данного подхода разработан прототип параллельной СУБД PargreSQL, основанный на свободной последовательной СУБД PostgreSQL. Проведены вычислительные эксперименты, подтверждающие эффективность

и масштабируемость данного прототипа.

Разработаны новые параллельные алгоритмы анализа временных рядов. Параллельный алгоритм поиска похожих подпоследовательностей в сверхбольших временных рядах для вычислительных кластеров с узлами на базе многоядерных ускорителей предполагает двухуровневое распараллеливание обработки (на уровне кластера и внутри одного узла) и дополнительные структуры данных, обеспечивающие эффективную векторизацию вычислений. Параллельный алгоритм поиска диссонансов временного ряда для многоядерных ускорителей использует дополнительные структуры данных и специализированную их компоновку в памяти, которые обеспечивают эффективную векторизацию вычислений. Проведены вычислительные эксперименты, подтверждающие высокую эффективность и масштабируемость разработанных алгоритмов.

Разработаны новые аналитические алгоритмы для для параллельной СУБД, выполняющие обработку данных сверхбольших объемов, которые не могут быть размещены в оперативной памяти: алгоритм кластеризации графа и алгоритм нечеткой кластеризации данных. Предложенные алгоритмы предполагают хранение исходных и промежуточных данных в реляционной базе данных и реализацию вычислений в виде запросов SQL. Проведены вычислительные эксперименты с СУБД PargreSQL, показавшие эффективность предложенных алгоритмов. Разработанные алгоритмы опережают аналоги по производительности, поскольку последние в том числе требуют накладных расходов на экспорт анализируемых данных из базы данных и импорт результатов в базу данных.

Разработаны новые параллельные алгоритмы поиска частых наборов для многоядерных вычислительных систем: ускорителя архитектуры Intel MIC и процессора IBM Cell BE. Предложенные алгоритмы предполагают специализированные представления данных в памяти и реализацию вычислений с использованием логических битовых функций, позволяющих эффективно векторизовать вычисления. Проведены вычислительные эксперименты, подтверждающие высокую эффективность и масштабируемость

разработанных алгоритмов.

Отличия исследования от предыдущих работ

Основные результаты, полученные в ходе выполнения диссертационного исследования являются новыми и не покрываются ранее опубликованными научными работами других авторов, обзор которых был дан в главе 1. Отметим основные отличия.

Подход к интеграции интеллектуального анализа данных в СУБД, предложенный в данном исследовании, идеологически близок к исследованиям научной группы под руководством Ордонеза (Ordonez) [178, 179, 182] и системе MADlib [103]. Они основаны на применении хранимых процедур, которые реализованы на языке высокого уровня C и используют сторонние библиотеки для решения базовых математических задач (LAPACK, Intel MKL и др.). Предложенный подход предполагает реализацию аналитических алгоритмов на основе повторно используемых структур данных (матрица расстояний между объектами и др.). Реализация соответствующей системы анализа данных предполагает наличие модуля, который обеспечивает удерживание в буферном пуле предвычисленных структур данных, что обеспечивает более высокую производительность при их повторном использовании.

В работах [45–47] предложен подход к интеграции интеллектуального анализа данных в СУБД на основе виртуальных аналитических представлений, которые обеспечивают логическое хранение (в отличие от физического хранения таблиц базы данных) результатов интеллектуального анализа данных. Предложенный подход идеологически близок данному в том смысле, что предполагает хранение результатов анализа не в виде таблиц базы данных, а в промежуточном представлении в формате JSON. Однако JSON-представление может включать в себя описание не только результатов анализа, но также и его метаданных (точность вычислений, последовательность значений целевой функции в процессе вычислений, время

выполнения и производительность аналитического алгоритма в различных фазах его работы и др.), что может быть использовано в экспериментальных исследованиях для интеллектуального анализа научных данных. Кроме того, аналитические алгоритмы предложенного подхода инкапсулируют параллелизм для многоядерных ускорителей.

Система DAnA [147] для интеллектуального анализа данных в СУБД, так же как и предложенный подход, инкапсулирует параллелизм многоядерных ускорителей. Ускорители FPGA, используемые в DanA, обеспечивают высокую производительность анализа данных, однако специфика их архитектуры обуславливает необходимость включения в состав данной системы специализированных аппаратных устройств (страйдеров), которые имеют прямой интерфейс доступа к буферному пулу СУБД и выполняют извлечение, очистку и обработку данных, передаваемых затем на FPGA для выполнения аналитического алгоритма. Подход в данном исследовании ориентирован на ускорители Intel MIC, которые в общем случае проигрывают ускорителям FPGA по производительности, но основаны на широко распространенной архитектуре Intel x86, в силу чего предложенный подход не требует дополнительных аппаратных устройств, и может быть перенесен на практически любую свободную реляционную СУБД (что потребует нетривиальных, но все же механических усилий по разработке).

Метод внедрения фрагментного параллелизма в реляционные СУБД, предложенные в данной работе, имеет следующие отличия от коммерческих параллельных СУБД Teradata [185], Greenplum [237], IBM DB2 Parallel Edition [39] и др. Коммерческие параллельные СУБД имеют высокую стоимость и ориентированы на специфические аппаратно-программные платформы, в то время как предложенные методы базируются на платформе вычислительного кластера. Коммерческие СУБД покажут ожидаемо бóльшую эффективность в сравнении с параллельной СУБД для кластеров. Однако параллельная СУБД, полученная посредством внедрения фрагментного параллелизма, потенциально способна показать сравнимую коммерческими параллельными СУБД масштабируемость, достигаемую путем

добавления в кластер новых вычислительных узлов, но оставаясь при этом финансово менее затратным решением.

В данном исследовании разработан ряд параллельных алгоритмов анализа данных для многоядерных ускорители архитектуры Intel MIC, в то время как аналоги ориентированы на платформу NVIDIA GPU. Вычислительные системы Intel MIC основаны на архитектуре Intel x86 и поддерживают те же модели и инструменты параллельного программирования, что и обычные процессоры Intel Xeon, обеспечивая при этом сравнимую с GPU производительность. Однако достижение наивысшей производительности на ускорителях Intel MIC требует, с одной стороны, организации вычислений так им образом, чтобы они включали в себя как можно большее количество векторизуемых компиляторов циклов обработки данных, а с другой стороны — большого объема вовлеченных в вычисления данных (от сотни тысяч до десятков миллионов элементов). В силу этого предложенные в данном исследовании алгоритмы *PBM* для поиска похожих подпоследовательностей и *MDD* для поиска диссонансов во временном ряде, в отличие от алгоритмов-аналогов в работах [106, 216, 219] и [107, 245] соответственно, применимы для временных рядов из миллионов элементов. Используемая техника предвычислений позволяет в итоге получить высокую масштабируемость алгоритмов и преимущество в производительности перед аналогами на данных указанных размеров.

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

Положения, выносимые на защиту

На защиту выносятся следующие новые научные результаты.

1. Разработан и исследован комплекс параллельных алгоритмов интеллектуального анализа данных средствами СУБД на кластерных вычислительных системах с многоядерными ускорителями:

1) параллельный алгоритм поиска похожих подпоследовательностей временного ряда для кластеров с многоядерными ускорителями;
2) параллельный алгоритм поиска диссонансов временного ряда для многоядерных ускорителей;
3) алгоритм кластеризации графа для параллельной СУБД на основе фрагментного параллелизма;
4) алгоритм нечеткой кластеризации данных для параллельной СУБД на основе фрагментного параллелизма;
5) параллельный алгоритм кластеризации данных для многоядерных ускорителей;
6) параллельный алгоритм поиска частых наборов для многоядерных ускорителей.

2. Предложен подход к внедрению параллельных алгоритмов интеллектуального анализа данных в реляционные СУБД.

3. Предложен метод внедрения фрагментного параллелизма в последовательные СУБД с открытым исходным кодом.

Степень достоверности результатов

Разработанные методы и алгоритмы подтверждены вычислительными экспериментами над реальными и синтетическими данными, проведенными в соответствии с общепринятыми стандартами.

Публикации соискателя по теме диссертации

Основные результаты диссертации опубликованы в следующих научных работах.

Статьи в журналах из перечня ВАК

1. *Пан К.С., Цымблер М.Л.* Внедрение фрагментного параллелизма в СУБД с открытым кодом // Программирование. 2015. Т. 41, № 5. С. 18–32.

2. *Пан К.С., Соколинский Л.Б., Цымблер М.Л.* Интеграция параллелизма в СУБД с открытым кодом // Открытые системы. СУБД. 2013. № 9. С. 56–58.

3. *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. № 18(277). Вып. 12. С. 112–120.

4. *Мовчан А.В., Цымблер М.Л.* Обнаружение подпоследовательностей во временных рядах // Открытые системы. СУБД. 2015. № 2. С. 42–43.

5. *Миниахметов Р.М., Цымблер М.Л.* Интеграция алгоритма кластеризации Fuzzy c-Means в PostgreSQL // Вычислительные методы и программирование: Новые вычислительные технологии. 2012. Т. 13. С. 46–52.

6. *Пан К.С., Цымблер М.Л.* Параллельный алгоритм решения задачи анализа рыночной корзины на процессорах Cell // Вестник ЮУрГУ. Серия: Математическое моделирование и программирование. 2010. № 16(192). Вып. 5. С. 48–57.

7. *Речкалов Т.В., Цымблер М.Л.* Параллельный алгоритм вычисления матрицы Евклидовых расстояний для многоядерного процессора Intel Xeon Phi Knights Landing // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. 2018. Т. 7, № 3. С. 65–82. DOI: 10.14529/cmse180305

8. *Краева Я.А., Цымблер М.Л.* Совместное использование технологий MPI и OpenMP для параллельного поиска похожих подпоследовательностей

в сверхбольших временных рядах на вычислительном кластере с узлами на базе многоядерных процессоров Intel Xeon Phi Knights Landing // Вычислительные методы и программирование: Новые вычислительные технологии. 2019. Т. 20, № 1. С. 29–43. DOI: 10.26089/NumMet.v20r104

9. *Цымблер М.Л.* Параллельный алгоритм поиска диссонансов временного ряда для многоядерных ускорителей // Вычислительные методы и программирование: Новые вычислительные технологии. 2019. Т. 20, № 3. С. 211–223. DOI: 10.26089/NumMet.v20r320

10. *Речкалов Т.В., Цымблер М.Л.* Параллельный алгоритм кластеризации данных для многоядерных ускорителей Intel MIC // Вычислительные методы и программирование: Новые вычислительные технологии. 2019. Т. 20, № 2. С. 104–115. DOI: 10.26089/NumMet.v20r211

11. *Цымблер М.Л.* Параллельный поиск частых наборов на многоядерных ускорителях Intel MIC // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. 2019. Т. 8, № 1. С. 54–70. DOI: 10.14529/cmse190104

12. *Цымблер М.Л.* Обзор методов интеграции интеллектуального анализа данных в СУБД // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. 2019. Т. 8, № 2. С. 32–62. DOI: 10.14529/cmse190203

Статьи в изданиях, индексируемых в Scopus и Web of Science

13. *Zymbler M.* Parallel Algorithm for Frequent Itemset Mining on Intel Many-core Systems // Journal of Computing and Information Technology. 2018. Vol. 26, No. 4. P. 209–221. DOI: 10.20532/cit.2018.1004382

14. *Zymbler M.* Best-match Time Series Subsequence Search on the Intel Many Integrated Core Architecture // Proceedings of the 19th East-European Conference on Advances in Databases and Information Systems, ADBIS 2015 (September 8–11, 2015, Poitiers, France). Lecture Notes in Computer Science. Springer, 2015. Vol. 9282. P. 275–286. DOI: 10.1007/978-3-319-23135-8_19

15. *Kraeva Ya., Zymbler M.* Scalable Algorithm for Subsequence Similarity Search in Very Large Time Series Data on Cluster of Phi KNL // 20th International Conference on Data Analytics and Management in Data Intensive Domains, DAMDID/RCDL 2018, Moscow, Russia, October 9–12, 2018, Revised Selected Papers. Communications in Computer and Information Science. 2019. Vol. 1003. P. 149–164. 10.1007/978-3-030-23584-0_9

16. *Movchan A., Zymbler M.* Parallel Algorithm for Local-best-match Time Series Subsequence Similarity Search on the Intel MIC Architecture // Procedia Computer Science. 2015. Vol. 66. P. 63–72. DOI: 110.1016/j.procs.2015.11.009

17. *Rechkalov T., Zymbler M.* Accelerating Medoids-based Clustering with the Intel Many Integrated Core Architecture // Proceedings of the 9th International Conference on Application of Information and Communication Technologies, AICT'2015 (October 14–16, 2015, Rostov-on-Don, Russia). IEEE, 2015. P. 413–417. DOI: 10.1109/ICAICT.2015.7338591

18. *Rechkalov T., Zymbler M.* Integrating DBMS and Parallel Data Mining Algorithms for Modern Many-Core Processors // Revised Selected Papers of the 19th International Conference on Data Analytics and Management in Data Intensive Domains, DAMDID/RCDL 2017 (October 10–13, 2017, Moscow, Russia). Communications in Computer and Information Science. Springer, 2018. Vol. 822. P. 230–245. DOI: 10.1007/978-3-319-96553-6_17

19. *Pan C., Zymbler M.* Taming Elephants, or How to Embed Parallelism into PostgreSQL // Proceedings of the 24th International Conference on Database and Expert Systems Applications, DEXA 2013 (August 26–29, 2013, Prague, Czech Republic). Lecture Notes in Computer Science. Springer, 2013. Vol. 8055. Part I. P. 153–164. DOI: 10.1007/978-3-642-40285-2_15

20. *Pan C., Zymbler M.* Very Large Graph Partitioning by Means of Parallel DBMS // Proceedings of the 17th East-European Conference on Advances in Databases and Information Systems, ADBIS 2013 (September 1–4, 2013, Genoa, Italy). Lecture Notes in Computer Science. Springer, 2013. Vol. 8133. P. 388–399. DOI: 10.1007/978-3-642-40683-6_29

21. *Rechkalov T., Zymbler M.* A Study of Euclidean Distance Matrix Computation on Intel Many-core Processors // Revised Selected Papers of the 12th International Conference, PCT 2018 (April 2–6, 2018, Rostov-on-Don, Russia). Communications in Computer and Information Science. Springer, 2018. Vol. 910. P. 200–215. DOI: 10.1007/978-3-319-99673-8_15

Свидетельства о регистрации программ

22. *Мовчан А.В., Цымблер М.Л.* Свидетельство Роспатента о государственной регистрации программы для ЭВМ «Программный комплекс для поиска похожих подпоследовательностей временного ряда на многоядерном сопроцессоре Intel Xeon Phi» № 2015618537 от 11.08.2015.

23. *Соколинский Л.Б., Цымблер М.Л., Пан К.С., Медведев А.А.* Свидетельство Роспатента о государственной регистрации программы для ЭВМ «Параллельная СУБД PargreSQL» № 2012614599 от 23.05.2012.

24. *Цымблер М.Л., Пан К.С.* Свидетельство Роспатента о государственной регистрации программы для ЭВМ «Программный комплекс для реше-

ния задачи анализа рыночной корзины на многоядерных процессорах с поддержкой векторных вычислений» № 2011610732 от 11.01.2011.

Личный вклад. Все совместные публикации, кроме работы [2], выполнены М.Л. Цымблером в соавторстве с аспирантами и студентами, у которых он являлся научным руководителем. В статье [1] М.Л. Цымблеру принадлежат разделы 1–3, 5 (введение, обзор работ, описание методов внедрения фрагментного параллелизма и архитектуры СУБД PargreSQL, заключение, стр. 18–28, 30), К.С. Пану принадлежит раздел 4 (описание результатов экспериментов, стр. 28–30). В статье [2] М.Л. Цымблеру принадлежит описание архитектуры СУБД PargreSQL (стр. 57), Л.Б. Соколинскому принадлежит вводная часть (стр. 56), К.С. Пану принадлежит описание результатов тестирования СУБД PargreSQL (стр. 58). В статье [3] М.Л. Цымблеру принадлежат введение, разделы 2–4 и заключение (описание архитектуры и принципов реализации СУБД PargreSQL, стр. 112, 113–118), К.С. Пану принадлежит раздел 1 (обзор работ, стр. 112–113). В статье [19] М.Л. Цымблеру принадлежат разделы 1–3, 5–6 (введение, описание принципов проектирования и реализации СУБД PargreSQL, обзор работ, заключение, стр. 153–162, 163), К.С. Пану принадлежит раздел 4 (описание результатов экспериментов, стр. 162). В статье [20] М.Л. Цымблеру принадлежит разделы 1–4, 6 (введение, обзор работ, описания СУБД PargreSQL и алгоритма, заключение, стр. 388–396, 397), К.С. Пану принадлежит раздел 5 (описание результатов экспериментов, стр. 396–397). В статье [6] М.Л. Цымблеру принадлежат введение, разделы 1–3, 5 (формальная постановка задачи, обзор работ и описание алгоритма, заключение, стр. 48–54, 57), К.С. Пану принадлежит раздел 4 (описание результатов экспериментов, стр. 54–56). В статье [4] М.Л. Цымблеру принадлежат описания алгоритма и результатов экспериментов (стр. 43), А.В. Мовчану принадлежит вводная часть (стр. 42). В статье [16] М.Л. Цымблеру принадлежит разделы 1–3, 5 (введение, формальная постановка задачи, описание алгоритма, заключение, стр. 65–69, 71), А.В. Мовчану принадлежит раздел 4 (описание результатов экспериментов, стр. 69–71). В статье [8] М.Л. Цымблеру

принадлежат разделы 1–4 (введение, формальная постановка задачи, описание алгоритма, заключение, стр. 33–36, 40), Я.А. Краевой принадлежит раздел 4 (описание результатов экспериментов, стр. 36–40). В статье [15] М.Л. Цымблеру принадлежат разделы 1–4, 6 (введение, формальная постановка задачи, обзор работ, описание алгоритма, заключение, стр. 149–156, 161–162), Я.А. Краевой принадлежит раздел 5 (описание результатов экспериментов, стр. 156–161). В статье [5] М.Л. Цымблеру принадлежат разделы 1–3 (введение, формальная постановка задачи, описание алгоритма, заключение, стр. 47–50, 51), Р.М. Миниахметову принадлежит раздел 4 (описание результатов экспериментов, стр. 50–51). В статье [18] М.Л. Цымблеру принадлежат разделы 1–3, 5 (введение, обзор работ, описание подхода к интеграции анализа данных в СУБД, заключение, стр. 230–238, 242–243), Т.В. Речкалову принадлежит раздел 4 (описание результатов экспериментов, стр. 238–242). В статье [7] М.Л. Цымблеру принадлежат введение, разделы 1–2, заключение (обзор работ, описание алгоритма, стр. 65–71, 77–78), Т.В. Речкалову принадлежит раздел 3 (описание результатов экспериментов, стр. 71–77). В статье [21] М.Л. Цымблеру принадлежат разделы 1–3, 5 (введение, обзор работ, описание алгоритма, заключение, стр. 200–206, 213–214), Т.В. Речкалову принадлежит раздел 4 (описание результатов экспериментов, стр. 206–213). В статье [10] М.Л. Цымблеру принадлежат разделы 1–3, 5 (введение, формальная постановка задачи и обзор работ, описание алгоритма, заключение, стр. 104–109, 111, 113), Т.В. Речкалову принадлежит раздел 4 (описание результатов экспериментов, стр. 110, 112). В статье [17] М.Л. Цымблеру принадлежат разделы 1–3, 5 (введение, формальная постановка задачи, описание алгоритма, обзор работ, заключение, стр. 413–415, 416–417), Т.В. Речкалову принадлежит раздел 4 (описание результатов экспериментов, стр. 415–416).

Апробация результатов исследования

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

— ПаВТ'2019: «Параллельные вычислительные технологии 2019» (2–4 апреля 2019 г., Калининград);

— DAMDID'2018: 20th International Conference on Data Analytics and Management in Data Intensive Domains (9–12 октября 2018 г., Москва);

— ПаВТ'2018: «Параллельные вычислительные технологии 2018» (3–5 апреля 2018 г., Ростов-на-Дону);

— DAMDID'2017: 19th International Conference on Data Analytics and Management in Data Intensive Domains (10–13 октября 2017 г., Москва);

— ПаВТ'2016: «Параллельные вычислительные технологии 2016» (29–31 марта 2016 г., Архангельск);

— SISAP'2015: 18th International Conference on Similarity Search and Applications (12–14 октября 2015 г., Глазго, Великобритания);

— AICT'2015: 9th International Conference on Application of Information and Communication Technologies (14–16 октября 2015 г., Ростов-на-Дону);

— «Суперкомпьютерные дни в России 2015» (28–29 сентября 2015 г., Москва);

— ADBIS'2015: 19th East-European Conference on Advances in Databases and Information Systems (8–11 сентября 2015 г., Пуатье, Франция);

— ПаВТ'2015: «Параллельные вычислительные технологии 2015» (31 марта – 2 апреля 2015 г., Екатеринбург);

— «Научный сервис в сети Интернет 2014: многообразие суперкомпьютерных миров» (22–27 сентября 2014 г., Новороссийск);

— ПаВТ'2014: «Параллельные вычислительные технологии 2014» (1–3 апреля 2014 г., Ростов-на-Дону);

— DEXA'2013: 24th International Conference on Database and Expert Systems Applications (26–30 августа 2013 г., Прага, Чехия);

— ADBIS'2013: 17th East-European Conference on Advances in Databases and Information Systems (1–4 сентября 2013 г., Генуя, Италия);

— «Научный сервис в сети Интернет 2013: все грани параллелизма» (23–28 сентября 2013 г., Новороссийск);

— ПаВТ'2011: «Параллельные вычислительные технологии 2011» (29–31 марта 2011 г., Москва);

— ПаВТ'2010: «Параллельные вычислительные технологии 2010» (30 марта – 1 апреля 2010 г., Уфа).

Направления будущих исследований

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

— Разработка специализированной параллельной СУБД для хранения и обработки сверхбольших временных рядов на платформе вычислительной кластерной системы с узлами на базе многоядерных ускорителей.

— Обобщение разработанных в диссертации методов и подходов на многомерные данные (мультивариативные временные ряды и др.).

Исходные тексты программ, реализующих методы и алгоритмы, разработанные в рамках диссертационного исследования, свободно доступны в сети Интернет по адресу: https://github.com/mzym/thesis/.

Диссертационное исследование выполнено при финансовой поддержке Российского фонда фундаментальных исследований (грант № 17-07-00463) и Министерства науки и высшего образования РФ (государственное задание 2.7905.2017/8.9).

Литература

1. Абдуллаев С.М., Ленская О.Ю., Гаязова А.О. и др. Алгоритмы краткосрочного прогноза с использованием радиолокационных данных: оценка трасляции и композиционный дисплей жизненного цикла // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. 2014. Т. 3, № 1. С. 17–32. URL: https://doi.org/10.14529/cmse140102.

2. Воеводин В.В., Воеводин В.В. Параллельные вычисления. СПб.: БХВ-Петербург, 2002. 608 с.

3. Дышаев М.М., Соколинская И.М. Представление торговых сигналов на основе адаптивной скользящей средней Кауфмана в виде системы линейных неравенств // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. 2013. Т. 2, № 4. С. 103–108. URL: https://doi.org/10.14529/cmse130408.

4. Епишев В.В., Исаев А.П., Миниахметов Р.М. и др. Система интеллектуального анализа данных физиологических исследований в спорте высших достижений // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. 2013. Т. 2, № 1. С. 44–54. URL: https://doi.org/10.14529/cmse130105.

5. Костенецкий П.С., Лепихов А.В., Соколинский Л.Б. Технологии параллельных систем баз данных для иерархических многопроцессорных сред // Автоматика и телемеханика. 2007. № 5. С. 112–125.

6. Костенецкий П.С., Сафонов А.Ю. Суперкомпьютерный комплекс ЮУрГУ // Параллельные вычислительные технологии (ПаВТ'2016): труды международной научной конференции (28 марта – 1 апреля 2016 г., г. Архангельск). Издательский центр ЮУрГУ, 2016. С. 561–573. URL: http://omega.sp.susu.ru/PaVT2016/short/119.pdf.

7. Краева Я.А., Цымблер М.Л. Совместное использование технологий MPI и OpenMP для параллельного поиска похожих подпоследовательностей в сверхбольших временных рядах на вычислительном кластере с узлами на базе многоядерных процессоров Intel Xeon Phi Knights Landing // Вычислительные методы и программирование: Новые вычислительные технологии. 2019. Т. 20, № 1. С. 29–43. URL: https://doi.org/10.26089/NumMet.v20r104.

8. Лепихов А.В., Соколинский Л.Б., Цымблер М.Л. Свидетельство Роспатента о государственной регистрации программы для ЭВМ «Параллельная СУБД “Омега” для кластерных систем» 2008614996 от 03.10.2008. 2008.

9. Миниахметов Р.М., Цымблер М.Л. Интеграция алгоритма кластеризации Fuzzy c-Means в PostgreSQL // Вычислительные методы и программирование: Новые вычислительные технологии. 2012. Т. 13. С. 46–52.

10. Мовчан А.В., Цымблер М.Л. Обнаружение подпоследовательностей во временных рядах // Открытые системы. СУБД. 2015. № 2. С. 42–43.

11. Мовчан А.В., Цымблер М.Л. Параллельная реализация поиска самой похожей подпоследовательности временного ряда для систем с распределенной памятью // Параллельные вычислительные технологии (ПаВТ'2016): труды международной научной конференции (28 марта – 1 апреля 2016 г., г. Архангельск). Издательский центр ЮУрГУ, 2016. С. 615–628. URL: http://omega.sp.susu.ru/PaVT2016/short/196.pdf.

12. Пан К.С., Соколинский Л.Б., Цымблер М.Л. Интеграция параллелизма в СУБД с открытым кодом // Открытые системы. СУБД. 2013. № 9. С. 56–58.

13. Пан К.С., Цымблер М.Л. Параллельный алгоритм решения задачи анализа рыночной корзины на процессорах Cell // Вестник Южно-Уральского государственного университета. Серия: Математическое моделирование и программирование. 2010. № 16(192). С. 48–57.

14. Пан К.С., Цымблер М.Л. Решение задачи анализа рыночной корзины на процессорах Cell // Параллельные вычислительные технологии (ПаВТ'2010): Труды международной научной конференции (Уфа, 29 марта – 2 апреля 2010 г.). Челябинск: Издательский центр ЮУрГУ, 2010. С. 551–560.

15. Пан К.С., Цымблер М.Л. Архитектура и принципы реализации параллельной СУБД PargreSQL // Параллельные вычислительные технологии (ПаВТ'2011): труды международной научной конференции (Москва, 28 марта – 1 апреля 2011 г.). Издательский центр ЮУрГУ, 2011. С. 577–584.

16. Пан К.С., Цымблер М.Л. Использование параллельной СУБД PargreSQL для интеллектуального анализа сверхбольших графов // Суперкомпьютерные технологии в науке, образовании и промышленности. 2012. № 1. С. 113–120.

17. Пан К.С., Цымблер М.Л. Разработка параллельной СУБД на основе последовательной СУБД PostgreSQL с открытым исходным кодом // Вестник Южно-Уральского государственного университета. Серия: Математическое моделирование и программирование. 2012. № 18(277). С. 112–120.

18. Пан К.С., Цымблер М.Л. Исследование эффективности параллельной СУБД PargreSQL // Научный сервис в сети Интернет: все грани параллелизма. Труды международной научной конференции (Новороссийск, 23–28 сентября 2013 г.). 2013. С. 148–149.

19. Пан К.С., Цымблер М.Л. Внедрение фрагментного параллелизма в СУБД с открытым кодом // Программирование. 2015. Т. 41, № 5. С. 18–32.

20. Речкалов Т.В. Подход к интеграции интеллектуального анализа данных в реляционную СУБД на основе генерации текстов хранимых процедур // Вестник Южно-Уральского государственного университета. Серия: Вычислительная математика и информатика. 2013. Т. 2, № 1. С. 114–121.

21. Речкалов Т.В., Цымблер М.Л. Параллельный алгоритм вычисления матрицы Евклидовых расстояний для многоядерного процессора Intel Xeon Phi Knights Landing // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. 2018. Т. 7, № 3. С. 65–82. URL: https://doi.org/10.14529/cmse180305.

22. Речкалов Т.В., Цымблер М.Л. Параллельный алгоритм кластеризации данных для многоядерных ускорителей Intel MIC // Вычислительные методы и программирование: Новые вычислительные технологии. 2019. Т. 20, № 2. С. 104–115. URL: https://doi.org/10.26089/NumMet.v20r211.

23. Соколинский Л.Б. Параллельные системы баз данных. Издательство Московского университета, 2013. 184 с.

24. Соколинский Л.Б., Цымблер М.Л., Пан К.С., Медведев А.А. Свидетельство Роспатента о государственной регистрации программы для ЭВМ «Параллельная СУБД PargreSQL» 2012614599 от 23.05.2012. 2012.

25. Цымблер М.Л. Обзор методов интеграции интеллектуального анализа данных в СУБД // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. 2019. Т. 8, № 2. С. 32–62. URL: https://doi.org/10.14529/10.14529/cmse190203.

26. Цымблер М.Л. Параллельный алгоритм поиска диссонансов временного ряда для многоядерных ускорителей // Вычислительные методы и программирование: Новые вычислительные технологии. 2019. Т. 20, № 3. С. 211–223. URL: https://doi.org/10.26089/NumMet.v20r320.

27. Цымблер М.Л. Параллельный поиск частых наборов на многоядерных ускорителях Intel MIC // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика. 2019. Т. 8, № 1. С. 54–70. URL: https://doi.org/10.14529/cmse190104.

28. Abadi D., Agrawal R., Ailamaki A. et al. The Beckman report on database research // Commun. ACM. 2016. Vol. 59, No. 2. P. 92–99. URL: https://doi.org/10.1145/2845915.

29. Aggarwal C.C., Wang H. Graph data management and mining: A survey of algorithms and applications // Managing and Mining Graph Data. 2010. P. 13–68. URL: https://doi.org/10.1007/978-1-4419-6045-0_2.

30. Agrawal R., Ailamaki A., Bernstein P.A. et al. The Claremont report on database research // Commun. ACM. 2009. Vol. 52, No. 6. P. 56–65. URL: https://doi.org/10.1145/1516046.1516062.

31. Agrawal R., Ailamaki A., Bernstein P.A. et al. The Claremont report on database research // SIGMOD Record. 2008. Vol. 37, No. 3. P. 9–19. URL: https://doi.org/10.1145/1462571.1462573.

32. Agrawal R., Faloutsos C., Swami A.N. Efficient similarity search in sequence databases // Proceedings of the 4th International Conference on Foundations of Data Organization and Algorithms, FODO'93, Chicago, Illinois, USA, October 13–15, 1993. 1993. P. 69–84. URL: https://doi.org/10.1007/3-540-57301-1_5.

33. Agrawal R., Shim K. Developing tightly-coupled Data Mining applications on a relational Database System // Proceedings of the Sec-

ond International Conference on Knowledge Discovery and Data Mining (KDD-96), Portland, Oregon, USA. 1996. P. 287–290. URL: http://www.aaai.org/Library/KDD/1996/kdd96-049.php.

34. Agrawal R., Srikant R. Fast algorithms for mining association rules in large databases // VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12–15, 1994, Santiago de Chile, Chile. 1994. P. 487–499. URL: http://www.vldb.org/conf/1994/P487.PDF.

35. Arevalo A., Matinata R.M., Pandian M. et al. Programming the IBM Cell Broadband Engine Architecture. Дата обращения: 03.10.2018. URL: http://www.redbooks.ibm.com/redbooks/pdfs/sg247575.pdf.

36. Athitsos V., Papapetrou P., Potamias M. et al. Approximate embedding-based subsequence matching of time series // Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10–12, 2008. 2008. P. 365–378. URL: https://doi.org/10.1145/1376616.1376656.

37. Bacon D.F., Graham S.L., Sharp O.J. Compiler transformations for high-performance computing // ACM Comput. Surv. 1994. Vol. 26, No. 4. P. 345–420. URL: https://doi.org/10.1145/197405.197406.

38. Baldi P., Sadowski P., Whiteson D. Searching for exotic particles in high-energy physics with deep learning // Nature Communications. 2014. Vol. 4. P. 4308. URL: https://doi.org/10.1038/ncomms5308.

39. 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. 1995. P. 460–462. URL: https://doi.org/10.1145/223784.223876.

40. Berndt D.J., Clifford J. Using Dynamic Time Warping to find patterns in time series // KDD Workshop. 1994. P. 359–370.

41. Bernstein P.A., Dayal U., DeWitt D.J. et al. Future Directions in DBMS Research - The Laguna Beach Participants // SIGMOD Record. 1989. Vol. 18, No. 1. P. 17–26.

42. Berthold M.R., Cebron N., Dill F. et al. KNIME - the Konstanz information miner: version 2.0 and beyond // SIGKDD Explorations. 2009. Vol. 11, No. 1. P. 26–31. URL: https://doi.org/10.1145/1656274.1656280.

43. Bezdek J.C. Pattern Recognition with Fuzzy Objective Function Algorithms. Springer, 1981. P. 256. URL: https://doi.org/10.1007/978-1-4757-0450-1.

44. Bezdek J.C., Ehrlich R., Full W. FCM: The fuzzy c-means clustering algorithm // Computers and Geosciences. 1984. Vol. 10, No. 2. P. 191–203. URL: https://doi.org/10.1016/0098-3004(84)90020-7.

45. Blockeel H., Calders T., Fromont É. et al. An inductive database prototype based on virtual mining views // Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Las Vegas, Nevada, USA, August 24–27, 2008. 2008. P. 1061–1064. URL: https://doi.org/10.1145/1401890.1402019.

46. Blockeel H., Calders T., Fromont É. et al. An inductive database system based on virtual mining views // Data Min. Knowl. Discov. 2012. Vol. 24, No. 1. P. 247–287. URL: https://doi.org/10.1007/s10618-011-0229-7.

47. Blockeel H., Calders T., Fromont É. et al. Inductive querying with virtual mining views // Inductive Databases and Constraint-Based Data Mining. Ed. by S. Dzeroski, B. Goethals, P. Panov. Springer, 2010. 2010. P. 265–287. URL: https://doi.org/10.1007/978-1-4419-7738-0_11.

48. Bogorny V., Kuijpers B., Alvares L.O. ST-DMQL: A semantic trajectory Data Mining query language // International Journal of Geographical Information Science. 2009. Vol. 23, No. 10. P. 1245–1276.

49. Bouganim L. Query load balancing in parallel database systems // Encyclopedia of Database Systems / Ed. by L. Liu, M.T. Özsu. Springer. 2009. P. 2268–2272. URL: https://doi.org/10.1007/978-0-387-39940-9_1080.

50. Brin S., Motwani R., Ullman J.D., Tsur S. Dynamic itemset counting and implication rules for market basket data // SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13–15, 1997, Tucson, Arizona, USA. 1997. P. 255–264. URL: https://doi.org/10.1145/253260.253325.

51. Burdick D., Calimlim M., Flannick J. et al. MAFIA: A Maximal Frequent Itemset Algorithm // IEEE Trans. Knowl. Data Eng. 2005. Vol. 17, No. 11. P. 1490–1504. URL: https://doi.org/10.1109/TKDE.2005.183.

52. Cadez I.V., Heckerman D., Meek C. et al. Visualization of navigation patterns on a Web site using model-based clustering // Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining, Boston, MA, USA, August 20–23, 2000. 2000. P. 280–284. URL: https://doi.org/10.1145/347090.347151.

53. Chakrabarti D., Zhan Y., Faloutsos C. R-MAT: A recursive model for graph mining // Proceedings of the 4th SIAM International Conference on Data Mining, Lake Buena Vista, Florida, USA, April 22–24, 2004. 2004. P. 442–446. URL: https://doi.org/10.1137/1.9781611972740.43.

54. Chang D., Jones N.A., Li D. et al. Compute pairwise Euclidean distances of data points with GPUs // Proceedings of the IASTED International

Symposium on Computational Biology and Bioinformatics, CBB'2008, November 16–18, 2008 Orlando, Florida, USA. 2008. P. 278–283.

55. Chang Y., Chen J., Tsai Y. Mining Subspace Clusters from DNA Microarray Data Using Large Itemset Techniques // Journal of Computational Biology. 2009. Vol. 16, No. 5. P. 745–768. URL: https://doi.org/10.1089/cmb.2008.0161.

56. Chaturvedi A., Green P.E., Carroll J.D. K-modes clustering // J. Classification. 2001. Vol. 18, No. 1. P. 35–55. URL: https://doi.org/10.1007/s00357-001-0004-3.

57. Chaudhuri S. An overview of query optimization in relational systems // Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1–3, 1998, Seattle, Washington, USA. 1998. P. 34–43. URL: https://doi.org/10.1145/275487.275492.

58. Chaudhuri S. What next?: a half-dozen data management research goals for Big Data and the cloud // Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012, Scottsdale, AZ, USA, May 20-24, 2012. 2012. P. 1–4. URL: https://doi.org/10.1145/2213556.2213558.

59. Chen R., Yang M., Weng X. et al. Improving large graph processing on partitioned graphs in the cloud // ACM Symposium on Cloud Computing, SOCC'12, San Jose, CA, USA, October 14–17, 2012. 2012. P. 3. URL: https://doi.org/10.1145/2391229.2391232.

60. Chen T., Raghavan R., Dale J.N., Iwata E. Cell Broadband Engine architecture and its first implementation - A performance view // IBM Journal of Research and Development. 2007. Vol. 51, No. 5. P. 559–572. URL: https://doi.org/10.1147/rd.515.0559.

61. Chen X., Petrounias I. Language support for temporal Data Mining // Principles of Data Mining and Knowledge Discovery, Second European Symposium, PKDD '98, Nantes, France, September 23–26, 1998, Proceedings. 1998. P. 282–290. URL: https://doi.org/10.1007/BFb0094830.

62. Cheung D.W., Hu K., Xia S. An Adaptive Algorithm for Mining Association Rules on Shared-Memory Parallel Machines // Distributed and Parallel Databases. 2001. Vol. 9, No. 2. P. 99–132. URL: https://doi.org/10.1023/A:1018951022124.

63. Chevalier C., Pellegrini F. PT-Scotch: A tool for efficient parallel graph ordering // Parallel Computing. 2008. Vol. 34, No. 6-8. P. 318–331. URL: https://doi.org/10.1016/j.parco.2007.12.001.

64. Codd E.F. A Relational Model of Data for Large Shared Data Banks // Commun. ACM. 1970. Vol. 13, No. 6. P. 377–387. URL: https://doi.org/10.1145/362384.362685.

65. Crockford D. The application/json media type for JavaScript Object Notation (JSON) // RFC. 2006. Vol. 4627. P. 1–10. URL: https://doi.org/10.17487/RFC4627.

66. Dan G. Algorithms on strings, trees, and sequences: computer science and computational biology. New York, NY, USA : Cambridge University Press, 1997.

67. de Souza Granha R.G.D. Hadoop // Encyclopedia of Big Data Technologies / Ed. by Sakr S., Zomaya A.Y. Springer. 2019. URL: https://doi.org/10.1007/978-3-319-63962-8_36-1.

68. Dean J., Ghemawat S. MapReduce: simplified data processing on large clusters // Commun. ACM. 2008. Vol. 51, No. 1. P. 107–113. URL: https://doi.org/10.1145/1327452.1327492.

69. Dembélé D., Kastner P. Fuzzy C-means Method for Clustering Microarray Data // Bioinformatics. 2003. Vol. 19, No. 8. P. 973–980. URL: https://doi.org/10.1093/bioinformatics/btg119.

70. Dempster A., Laird N., Rubin D. Maximum likelihood estimation from incomplete data via the EM algorithm // Journal of The Royal Statistical Society. 1977. Vol. 39, No. 1. P. 1–38.

71. DeWitt D.J., Gray J. Parallel database systems: the future of high performance database systems // Commun. ACM. 1992. Vol. 35, No. 6. P. 85–98. URL: https://doi.org/10.1145/129888.129894.

72. Ding H., Trajcevski G., Scheuermann P. et al. Querying and mining of time series data: experimental comparison of representations and distance measures // PVLDB. 2008. Vol. 1, No. 2. P. 1542–1552. URL: http://www.vldb.org/pvldb/1/1454226.pdf.

73. Dokmanic I., Parhizkar R., Ranieri J., Vetterli M. Euclidean distance matrices: essential theory, algorithms, and applications // IEEE Signal Processing Magazine. 2015. Vol. 32, No. 6. P. 12–30. URL: https://doi.org/10.1109/MSP.2015.2398954.

74. Donato D., Gionis A. A Survey of graph mining for web applications // Managing and Mining Graph Data / Ed. by C.C. Aggarwal, H. Wang. Springer, 2010. Vol. 40 of Advances in Database Systems. P. 455–485. URL: https://doi.org/10.1007/978-1-4419-6045-0_15.

75. Dong J., Han M. BitTableFI: An efficient mining frequent itemsets algorithm // Knowl.-Based Syst. 2007. Vol. 20, No. 4. P. 329–335. URL: https://doi.org/10.1016/j.knosys.2006.08.005.

76. Duan R., Strey A. Data Mining algorithms on the Cell Broadband Engine // Proceedings of the Euro-Par 2008 - Parallel Processing, 14th International Euro-Par Conference, Las Palmas de Gran Canaria, Spain,

August 26–29, 2008. 2008. P. 665–675. URL: https://doi.org/10.1007/978-3-540-85451-7_71.

77. Dunn J. A fuzzy relative of the ISODATA process and its use in detecting compact well-separated clusters // Journal of Cybernetics. 1973. Vol. 3, No. 3. P. 32–57. URL: https://doi.org/10.1080/01969727308546046.

78. Duran A., Klemm M. The Intel® Many Integrated Core architecture // 2012 International Conference on High Performance Computing & Simulation, HPCS 2012, Madrid, Spain, July 2–6, 2012. 2012. P. 365–366. URL: https://doi.org/10.1109/HPCSim.2012.6266938.

79. Eichinger F., Böhm K. Software-bug localization with graph mining // Managing and Mining Graph Data / Ed. by C.C. Aggarwal, H. Wang. Springer, 2010. Vol. 40 of Advances in Database Systems. P. 515–546. URL: https://doi.org/10.1007/978-1-4419-6045-0_17.

80. Engreitz J.M., Daigle B.J., Marshall J.J., Altman R.B. Independent component analysis: mining microarray data for fundamental human gene expression modules // Journal of Biomedical Informatics. 2010. Vol. 43, No. 6. P. 932–944. URL: https://doi.org/10.1016/j.jbi.2010.07.001.

81. Esling P., Agon C. Time-series Data Mining // ACM Comput. Surv. 2012. Vol. 45, No. 1. P. 12:1–12:34. URL: https://doi.org/10.1145/2379776.2379788.

82. Espenshade J., Pangborn A., von Laszewski G. et al. Accelerating Partitional Algorithms for Flow Cytometry on GPUs // IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA 2009, Chengdu, Sichuan, China, 10–12 August 2009. 2009. P. 226–233. URL: https://doi.org/10.1109/ISPA.2009.29.

83. Ester M., Kriegel H., Sander J., Xu X. A density-based algorithm for discovering clusters in large spatial databases with noise // Proceedings

of the 2nd International Conference on Knowledge Discovery and Data Mining (KDD-96), Portland, Oregon, USA. 1996. P. 226–231. URL: http://www.aaai.org/Library/KDD/1996/kdd96-037.php.

84. Evdoridis T., Tzouramanis T. A generalized comparison of open source and commercial Database Management Systems // Database Technologies: Concepts, Methodologies, Tools, and Applications (4 Volumes) / Ed. by J. Wang. IGI Global. 2009. P. 13–27. URL: https://doi.org/10.4018/978-1-60566-010-3.

85. Faloutsos C., Ranganathan M., Manolopoulos Y. Fast subsequence matching in time-series databases // Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24–27, 1994. 1994. P. 419–429. URL: https://doi.org/10.1145/191839.191925.

86. Fang J., Varbanescu A.L., Sips H.J. Sesame: A User-Transparent Optimizing Framework for Many-Core Processors // 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2013, Delft, Netherlands, May 13–16, 2013. 2013. P. 70–73. URL: https://doi.org/10.1109/CCGrid.2013.79.

87. Fjällström P.O. Algorithms for graph partitioning: A survey // Linköping Electronic Articles in Computer and Information Science. 1998. Vol. 3, No. 10. URL: http://www.ep.liu.se/ea/cis/1998/010/cis98010.pdf.

88. Frank E., Hall M.A., Holmes G. et al. WEKA - A Machine Learning workbench for Data Mining // The Data Mining and Knowledge Discovery Handbook. / Ed. by O. Maimon, L. Rokach. Springer, 2005. P. 1305–1314.

89. Frawley W.J., Piatetsky-Shapiro G., Matheus C.J. Knowledge Discovery in databases: an overview // Knowledge Discovery in Databases. AAAI/MIT Press, 1991. P. 1–30.

90. Fu A.W., Keogh E.J., Lau L.Y.H., Ratanamahatana C.A. Scaling and time warping in time series querying // Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 – September 2, 2005. 2005. P. 649–660. URL: http://www.vldb2005.org/program/paper/thu/p649-fu.pdf.

91. Fu A.W., Keogh E.J., Lau L.Y.H. et al. Scaling and time warping in time series querying // VLDB J. 2008. Vol. 17, No. 4. P. 899–921. URL: https://doi.org/10.1007/s00778-006-0040-z.

92. Fu T.C. A review on time series data mining // Eng. Appl. of AI. 2011. Vol. 24, No. 1. P. 164–181. URL: https://doi.org/10.1016/j.engappai.2010.09.007.

93. Ghadiri N., Ghaffari M., Nikbakht M.A. BigFCM: Fast, precise and scalable FCM on Hadoop // Future Generation Comp. Syst. 2017. Vol. 77. P. 29–39. URL: https://doi.org/10.1016/j.future.2017.06.010.

94. Graefe G. Encapsulation of parallelism in the Volcano query processing system // Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23–25, 1990. / Ed. by H. Garcia-Molina, H.V. Jagadish. ACM Press, 1990. P. 102–111. URL: https://doi.org/10.1145/93597.98720.

95. Graefe G. Query Evaluation Techniques for Large Databases // ACM Comput. Surv. 1993. Vol. 25, No. 2. P. 73–170. URL: https://doi.org/10.1145/152610.152611.

96. Gropp W. MPI 3 and beyond: why MPI is successful and what challenges it faces // EuroMPI. 2012. P. 1–9.

97. Guha S., Mishra N., Motwani R., O'Callaghan L. Clustering data streams // 41st Annual Symposium on Foundations of Computer Science, FOCS 2000, 12–14 November 2000, Redondo Beach, California,

USA. 2000. P. 359–366. URL: https://doi.org/10.1109/SFCS.2000.892124.

98. Han J., Fu Y., Wang W. et al. DBMiner: A system for mining knowledge in large relational databases // Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining (KDD-96), Portland, Oregon, USA. 1996. P. 250–255. URL: http://www.aaai.org/Library/KDD/1996/kdd96-041.php.

99. Han J., Kamber M. Data Mining: concepts and techniques. Morgan Kaufmann, 2006. P. 743.

100. Han J., Koperski K., Stefanovic N. GeoMiner: A system prototype for Spatial Data Mining // Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 1997, May 13–15, 1997, Tucson, Arizona, USA. 1997. P. 553–556. URL: https://doi.org/10.1145/253260.253404.

101. Han J., Pei J., Yin Y. Mining frequent patterns without candidate generation // Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16–18, 2000, Dallas, Texas, USA. 2000. P. 1–12. URL: https://doi.org/10.1145/342009.335372.

102. Harizopoulos S., Abadi D.J., Madden S., Stonebraker M. OLTP through the looking glass, and what we found there // Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10–12, 2008. 2008. P. 981–992. URL: https://doi.org/10.1145/1376616.1376713.

103. Hellerstein J.M., Ré C., Schoppmann F. et al. The MADlib analytics library or MAD skills, the SQL // PVLDB. 2012. Vol. 5, No. 12. P. 1700–1711. URL: https://doi.org/10.14778/2367502.2367510.

104. Hendrickson B. Chaco // Encyclopedia of Parallel Computing / Ed. by D.A. Padua. Springer. 2011. P. 248–249. URL: https://doi.org/10.1007/978-0-387-09766-4_310.

105. Hidri M.S., Zoghlami M.A., Ayed R.B. Speeding up the large-scale consensus fuzzy clustering for handling Big Data // Fuzzy Sets and Systems. 2018. Vol. 348. P. 50–74. URL: https://doi.org/10.1016/j.fss.2017.11.003.

106. Huang S., Dai G., Sun Y. et al. DTW-based subsequence similarity search on AMD heterogeneous computing platform // Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing, HPCC/EUC 2013, Zhangjiajie, China, November 13–15, 2013. 2013. P. 1054–1063. URL: https://doi.org/10.1109/HPCC.and.EUC.2013.149.

107. Huang T., Zhu Y., Mao Y. et al. Parallel discord discovery // Proceedings of the Advances in Knowledge Discovery and Data Mining – 20th Pacific-Asia Conference, PAKDD 2016, Auckland, New Zealand, April 19–22, 2016, Part II. 2016. P. 233–244. URL: https://doi.org/10.1007/978-3-319-31750-2_19.

108. Huang Z. Extensions to the k-Means algorithm for clustering large data sets with categorical values // Data Min. Knowl. Discov. 1998. Vol. 2, No. 3. P. 283–304. URL: https://doi.org/10.1023/A:1009769707641.

109. Imielinski T., Virmani A. MSQL: A query language for database mining // Data Min. Knowl. Discov. 1999. Vol. 3, No. 4. P. 373–408. URL: https://doi.org/10.1023/A:1009816913055.

110. Jaros M., Strakos P., Karásek T. et al. Implementation of K-means segmentation algorithm on Intel Xeon Phi and GPU: Application in medical

imaging // Advances in Engineering Software. 2017. Vol. 103. P. 21–28. URL: https://doi.org/10.1016/j.advengsoft.2016.05.008.

111. Jeffers J., Reinders J. Intel Xeon Phi coprocessor high performance programming. Morgan Kaufmann Publishers Inc., 2013. URL: https://doi.org/10.1016/C2011-0-06997-1.

112. Kahle J.A., Day M.N., Hofstee H.P. et al. Introduction to the Cell multiprocessor // IBM Journal of Research and Development. 2005. Vol. 49, No. 4-5. P. 589–604. URL: https://doi.org/10.1147/rd.494.0589.

113. Karypis G. METIS and ParMETIS // Encyclopedia of Parallel Computing / Ed. by D.A. Padua. Springer. 2011. P. 1117–1124. URL: https://doi.org/10.1007/978-0-387-09766-4_500.

114. Karypis G., Kumar V. Analysis of Multilevel Graph Partitioning // Proceedings Supercomputing '95, San Diego, CA, USA, December 4–8, 1995. 1995. P. 29. URL: https://doi.org/10.1145/224170.224229.

115. Kaufman L., Rousseeuw P.J. Finding groups in data: an introduction to cluster analysis. John Wiley, 1990. URL: https://doi.org/10.1002/9780470316801.

116. Keogh E., Lin J., Fu A. HOT SAX: efficiently finding the most unusual time series subsequence // Proceedings of the 5th IEEE International Conference on Data Mining, ICDM'05, Houston, Texas, November 27–30, 2005. 2005. P. 8. URL: https://doi.org/10.1109/ICDM.2005.79.

117. Keogh E.J., Lin J., Lee S., Herle H.V. Finding the most unusual time series subsequence: algorithms and applications // Knowl. Inf. Syst. 2007. Vol. 11, No. 1. P. 1–27. URL: https://doi.org/10.1007/s10115-006-0034-6.

118. Kernighan B.W., Lin S. An Efficient Heuristic Procedure for Partitioning Graphs // The Bell system technical journal. 1970. Vol. 49, No. 1. P. 291–307.

119. Kim J., Hwang I., Kim Y., Moon B.R. Genetic approaches for graph partitioning: a survey // 13th Annual Genetic and Evolutionary Computation Conference, GECCO 2011, Proceedings, Dublin, Ireland, July 12-16, 2011. 2011. P. 473–480. URL: https://doi.org/10.1145/2001576.2001642.

120. Kim S., Park S., Chu W.W. An index-based approach for similarity search supporting time warping in large sequence databases // Proceedings of the 17th International Conference on Data Engineering, April 2–6, 2001, Heidelberg, Germany. 2001. P. 607–614. URL: https://doi.org/10.1109/ICDE.2001.914875.

121. Kim S., Yoon J., Park S., Kim T. Shape-based retrieval of similar subsequences in time-series databases // Proceedings of the 2002 ACM Symposium on Applied Computing (SAC), March 10–14, 2002, Madrid, Spain. 2002. P. 438–445. URL: https://doi.org/10.1145/508791.508874.

122. Knuth D.E. The Art of Computer Programming, Volume 4, Fascicle 3: Generating All Combinations and Partitions. Addison-Wesley Professional, 2005.

123. Kohlhoff K.J., Sosnick M.H., Hsu W.T. et al. CAMPAIGN: an open-source library of GPU-accelerated data clustering algorithms // Bioinformatics. 2011. Vol. 27, No. 16. P. 2321–2322. URL: https://doi.org/10.1093/bioinformatics/btr386.

124. Kostenetskii P.S., Sokolinsky L.B. Simulation of hierarchical multiprocessor database systems // Programming and Computer Software. 2013. Vol. 39, No. 1. P. 10–24. URL: https://doi.org/10.1134/S0361768813010040.

125. Kraeva Y., Zymbler M. An efficient subsequence similarity search on modern Intel many-core processors for data intensive applications // Аналитика и управление данными в областях с интенсивным использованием данных: Сборник научных трудов XX Международной конференции DAMDID/RCDL'2018 (Москва, 9–12 октября 2018 г.). 2018. P. 116–124.

126. Kraeva Y., Zymbler M.L. An efficient subsequence similarity search on modern Intel many-core processors for data intensive applications // Selected Papers of the XX International Conference on Data Analytics and Management in Data Intensive Domains (DAMDID/RCDL 2018), Moscow, Russia, October 9-12, 2018. 2018. P. 143–151. URL: http://ceur-ws.org/Vol-2277/paper26.pdf.

127. Kraeva Y., Zymbler M.L. Scalable algorithm for subsequence similarity search in very large time series data on cluster of Phi KNL // 20th International Conference on Data Analytics and Management in Data Intensive Domains, DAMDID/RCDL 2018, Moscow, Russia, October 9–12, 2018, Revised Selected Papers. Communications in Computer and Information Science. 2019. P. 149–164. URL: https://doi.org/10.1007/978-3-030-23584-0_9.

128. Krause C., Johannsen D., Deeb R. et al. An SQL-based query language and engine for graph pattern matching // Graph Transformation - 9th International Conference, ICGT 2016, in Memory of Hartmut Ehrig, Held as Part of STAF 2016, Vienna, Austria, July 5–6, 2016, Proceedings. 2016. P. 153–169. URL: https://doi.org/10.1007/978-3-319-40530-8_10.

129. Kumar P., Bhatt P., Choudhury R. Bitwise Dynamic Itemset Counting Algorithm // Proceedings of the ICCIC 2015, IEEE International Conference on Computational Intelligence and Computing Research, December 10–12, 2015, Madurai, India. 2015. P. 1–4. URL: https://doi.org/10.1109/ICCIC.2015.7435752.

130. Kumar V., Grama A., Gupta A., Karypis G. Introduction to parallel computing. Benjamin/Cummings, 1994.

131. Kurte K.R., Durbha S.S. High resolution disaster data clustering using Graphics Processing Units // 2013 IEEE International Geoscience and Remote Sensing Symposium, IGARSS 2013, Melbourne, Australia, July 21–26, 2013. 2013. P. 1696–1699. URL: https://doi.org/10.1109/IGARSS.2013.6723121.

132. Lee S., Liao W., Agrawal A. et al. Evaluation of K-means data clustering algorithm on Intel Xeon Phi // 2016 IEEE International Conference on Big Data, BigData 2016, Washington DC, USA, December 5–8, 2016. 2016. P. 2251–2260. URL: https://doi.org/10.1109/BigData.2016.7840856.

133. Lepikhov A.V., Sokolinsky L.B. Query processing in a DBMS for cluster systems // Programming and Computer Software. 2010. Vol. 36, No. 4. P. 205–215. URL: https://doi.org/10.1134/S0361768810040031.

134. Lepinioti K., McKearney S. Integrating Cobweb with a relational database // Proceedings of the International MultiConference of Engineers and Computer Scientists 2007, IMECS 2007, March 21–23, 2007, Hong Kong, China. 2007. P. 868–873.

135. Li J., Fu A.W., Fahey P. Efficient discovery of risk patterns in medical data // Artificial Intelligence in Medicine. 2009. Vol. 45, No. 1. P. 77–89. URL: https://doi.org/10.1016/j.artmed.2008.07.008.

136. Lichman M. UCI Machine Learning Repository. Individual household electric power consumption dataset. Irvine, CA: University of California, School of Information and Computer Science. 2013. URL: http://archive.ics.uci.edu/ml/datasets/individual+household+electric+power+consumption.

137. Lim S., Park H., Kim S. Using multiple indexes for efficient subsequence matching in time-series databases // Database Systems for Advanced Applications, 11th International Conference, DASFAA 2006, Singapore, April 12–15, 2006, Proceedings. 2006. P. 65–79. URL: https://doi.org/10.1007/11733836_7.

138. Lim S., Park H., Kim S. Using multiple indexes for efficient subsequence matching in time-series databases // Inf. Sci. 2007. Vol. 177, No. 24. P. 5691–5706. URL: https://doi.org/10.1016/j.ins.2007.07.004.

139. Lima A.A.B., Furtado C., Valduriez P., Mattoso M. Parallel OLAP query processing in database clusters with data replication // Distributed and Parallel Databases. 2009. Vol. 25, No. 1–2. P. 97–123. URL: https://doi.org/10.1007/s10619-009-7037-8.

140. Lin J., Keogh E.J., Lonardi S., Chiu B.Y. A symbolic representation of time series, with implications for streaming algorithms // Proceedings of the 8th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery, DMKD 2003, San Diego, California, USA, June 13, 2003. 2003. P. 2–11. URL: https://doi.org/10.1145/882082.882086.

141. Lischner R. STL – pocket reference: containers, iterators, and algorithms. O'Reilly, 2003. URL: http://www.oreilly.de/catalog/stlpr/.

142. Liu G., Lu H., Lou W. et al. Efficient mining of frequent patterns using ascending frequency ordered prefix-tree // Data Min. Knowl. Discov. 2004. Vol. 9, No. 3. P. 249–274. URL: https://doi.org/10.1023/B:DAMI.0000041128.59011.53.

143. Liu J., Pan Y., Wang K., Han J. Mining frequent item sets by opportunistic projection // Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, July 23–26, 2002, Edmonton, Alberta, Canada. 2002. P. 229–238. URL: https://doi.org/10.1145/775047.775081.

144. Lloyd S.P. Least squares quantization in PCM // IEEE Transactions on Information Theory. 1982. Vol. 28, No. 2. P. 129–136. URL: https://doi.org/10.1109/TIT.1982.1056489.

145. Ludwig S.A. MapReduce-based fuzzy c-means clustering algorithm: implementation and scalability // Int. J. Machine Learning & Cybernetics. 2015. Vol. 6, No. 6. P. 923–934. URL: https://doi.org/10.1007/s13042-015-0367-0.

146. Macqueen J. Some methods of classification and analysis of multivariate observations // Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability. 1967. P. 281–297.

147. Mahajan D., Kim J.K., Sacks J. et al. In-RDBMS Hardware Acceleration of Advanced Analytics // PVLDB. 2018. Vol. 11, No. 11. P. 1317–1331. URL: http://www.vldb.org/pvldb/vol11/p1317-mahajan.pdf.

148. Malerba D., Appice A., Ceci M. A Data Mining query language for Knowledge Discovery in a Geographical Information System // Database Support for Data Mining Applications: Discovering Knowledge with Inductive Queries. 2004. P. 95–116. URL: https://doi.org/10.1007/978-3-540-44497-8_5.

149. Martino F.D., Pedrycz W., Sessa S. Spatiotemporal extended fuzzy C-means clustering algorithm for hotspots detection and prediction // Fuzzy Sets and Systems. 2018. Vol. 340. P. 109–126. URL: https://doi.org/10.1016/j.fss.2017.11.011.

150. Mattson T. S08 - Introduction to OpenMP // Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing, November 11–17, 2006, Tampa, FL, USA. 2006. P. 209. URL: https://doi.org/10.1145/1188455.1188673.

151. Matusevich D.S., Ordonez C. A clustering algorithm merging MCMC and EM methods using SQL Queries // Proceedings of the 3rd Internation-

al Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications, BigMine 2014, New York City, USA, August 24, 2014. 2014. P. 61–76. URL: http://jmlr.org/proceedings/papers/v36/matusevich14.html.

152. Mazandarani F.N., Mohebbi M. Wide complex tachycardia discrimination using dynamic time warping of ECG beats // Computer Methods and Programs in Biomedicine. 2018. Vol. 164. P. 238–249. URL: https://doi.org/10.1016/j.cmpb.2018.04.009.

153. Meek C., Thiesson B., Heckerman D. The learning-curve sampling method applied to model-based clustering // Journal of Machine Learning Research. 2002. Vol. 2. P. 397–418. URL: http://www.jmlr.org/papers/v2/meek02a.html.

154. Mehta M., DeWitt D.J. Data placement in shared-nothing parallel database systems // VLDB J. 1997. Vol. 6, No. 1. P. 53–72. URL: https://doi.org/10.1007/s007780050033.

155. Melnykov V., Chen W.C., Maitra R. MixSim: An R package for simulating data to study performance of clustering algorithms // Journal of Statistical Software, Articles. 2012. Vol. 51, No. 12. P. 1–25. URL: https://doi.org/10.18637/jss.v051.i12.

156. Mennis J., Guo D. Spatial data mining and geographic knowledge discovery - An introduction // Computers, Environment and Urban Systems. 2009. Vol. 33, No. 6. P. 403–408. URL: https://doi.org/10.1016/j.compenvurbsys.2009.11.001.

157. Meo R., Psaila G., Ceri S. A new SQL-like operator for mining association rules // Proceedings of 22th International Conference on Very Large Data Bases, VLDB'96, September 3–6, 1996, Mumbai (Bombay), India. 1996. P. 122–133. URL: http://www.vldb.org/conf/1996/P122.PDF.

158. Miniakhmetov R., Movchan A., Zymbler M. Accelerating time series subsequence matching on the Intel Xeon Phi many-core coprocessor // 38th International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2015, Opatija, Croatia, May 25–29, 2015. 2015. P. 1399–1404. URL: https://doi.org/10.1109/MIPRO.2015.7160493.

159. Mohammadi M., Al-Fuqaha A.I. Enabling cognitive Smart Cities using Big Data and Machine Learning: approaches and challenges // IEEE Communications Magazine. 2018. Vol. 56, No. 2. P. 94–101. URL: https://doi.org/10.1109/MCOM.2018.1700298.

160. Moon Y., Whang K., Han W. General Match: a subsequence matching method in time-series databases based on generalized windows // Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3–6, 2002. 2002. P. 382–393. URL: https://doi.org/10.1145/564691.564735.

161. Moon Y., Whang K., Loh W. Duality-based subsequence matching in time-Series databases // Proceedings of the 17th International Conference on Data Engineering, April 2–6, 2001, Heidelberg, Germany. 2001. P. 263–272. URL: https://doi.org/10.1109/ICDE.2001.914837.

162. Movchan A., Zymbler M. Parallel algorithm for local-best-match time series subsequence similarity search on the Intel MIC architecture // Procedia Computer Science. 2015. Vol. 66. P. 63–72. URL: https://doi.org/10.1016/j.procs.2015.11.009.

163. Movchan A.V., Zymbler M.L. Time series subsequence similarity search under Dynamic Time Warping distance on the Intel many-core accelerators // Similarity Search and Applications - 8th International Conference, SISAP 2015, Glasgow, UK, October 12–14, 2015, Proceedings. 2015. P. 295–306. URL: https://doi.org/10.1007/978-3-319-25087-8_28.

164. Musselman A. Apache Mahout // Encyclopedia of Big Data Technologies / Ed. by Sakr S., Zomaya A.Y. Springer. 2019. URL: https://doi.org/10.1007/978-3-319-63962-8_144-1.

165. Nambiar R.O., Poess M., Masland A. et al. TPC benchmark roadmap 2012 // TPCTC. 2012. P. 1–20. URL: https://doi.org/10.1007/978-3-642-36727-4_1.

166. Narayanan R., Özisikyilmaz B., Zambreno J. et al. MineBench: A benchmark suite for Data Mining workloads // Proceedings of the 2006 IEEE International Symposium on Workload Characterization, IISWC 2006, October 25–27, 2006, San Jose, California, USA. 2006. P. 182–188. URL: https://doi.org/10.1109/IISWC.2006.302743.

167. Neumann T. Query optimization (in relational databases) // Encyclopedia of Database Systems / Ed. by L. Liu, M.T. Özsu. Springer. 2009. P. 2273–2278. URL: https://doi.org/10.1007/978-0-387-39940-9_293.

168. Ng R.T., Han J. Efficient and effective clustering methods for spatial Data Mining // VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12–15, 1994, Santiago de Chile, Chile. 1994. P. 144–155. URL: http://www.vldb.org/conf/1994/P144.PDF.

169. Oliveira M.D.B., Gama J. An overview of social network analysis // Wiley Interdisc. Rew.: Data Mining and Knowledge Discovery. 2012. Vol. 2, No. 2. P. 99–115. URL: https://doi.org/10.1002/widm.1048.

170. O'Neil E.J., O'Neil P.E., Weikum G. The LRU-K page replacement algorithm for database disk buffering // Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26–28, 1993. 1993. P. 297–306. URL: https://doi.org/10.1145/170035.170081.

171. Ordonez C. Programming the k-means clustering algorithm in SQL // Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Seattle, Washington, USA, August 22–25, 2004. 2004. P. 823–828. URL: https://doi.org/10.1145/1014052.1016921.

172. Ordonez C. Integrating k-means clustering with a relational DBMS using SQL // IEEE Trans. Knowl. Data Eng. 2006. Vol. 18, No. 2. P. 188–201. URL: https://doi.org/10.1109/TKDE.2006.31.

173. Ordonez C. Statistical model computation with UDFs // IEEE Trans. Knowl. Data Eng. 2010. Vol. 22, No. 12. P. 1752–1765. URL: https://doi.org/10.1109/TKDE.2010.44.

174. Ordonez C. Can we analyze big data inside a DBMS? // Proceedings of the sixteenth international workshop on Data warehousing and OLAP, DOLAP 2013, San Francisco, CA, USA, October 28, 2013. 2013. P. 85–92. URL: https://doi.org/10.1145/2513190.2513198.

175. Ordonez C., Cereghini P. SQLEM: fast clustering in SQL using the EM algorithm // Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16–18, 2000, Dallas, Texas, USA. 2000. P. 559–570. URL: https://doi.org/10.1145/342009.335468.

176. Ordonez C., Chen Z. Horizontal aggregations in SQL to prepare data sets for Data Mining analysis // IEEE Trans. Knowl. Data Eng. 2012. Vol. 24, No. 4. P. 678–691. URL: https://doi.org/10.1109/TKDE.2011.16.

177. Ordonez C., Ezquerra N.F., Santana C.A. Constraining and summarizing association rules in medical data // Knowl. Inf. Syst. 2006. Vol. 9, No. 3. P. 1–2. URL: https://doi.org/10.1007/s10115-005-0226-5.

178. Ordonez C., Garcia-Alvarado C. A data mining system based on SQL queries and UDFs for relational databases // Proceedings of the 20th ACM Conference on Information and Knowledge Management, CIKM

2011, Glasgow, United Kingdom, October 24–28, 2011. 2011. P. 2521–2524. URL: https://doi.org/10.1145/2063576.2064008.

179. Ordonez C., Garcia-Alvarado C., Baladandayuthapani V. Bayesian variable selection in linear regression in one pass for large datasets // TKDD. 2014. Vol. 9, No. 1. P. 3:1–3:14. URL: https://doi.org/10.1145/2629617.

180. Ordonez C., García-García J. Database systems research on data mining // Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6–10, 2010. 2010. P. 1253–1254. URL: https://doi.org/10.1145/1807167.1807335.

181. Ordonez C., García-García J., Garcia-Alvarado C. et al. Data mining algorithms as a service in the cloud exploiting relational database systems // Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22–27, 2013. 2013. P. 1001–1004. URL: https://doi.org/10.1145/2463676.2465240.

182. Ordonez C., Mohanam N., Garcia-Alvarado C. PCA for large data sets with parallel data summarization // Distributed and Parallel Databases. 2014. Vol. 32, No. 3. P. 377–403. URL: https://doi.org/10.1007/s10619-013-7134-6.

183. Ordonez C., Santana C.A., de Braal L. Discovering Interesting Association Rules in Medical Data // 2000 ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, Dallas, Texas, USA, May 14, 2000. 2000. P. 78–85.

184. Owens J. GPU architecture overview // International Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 2007, San

Diego, California, USA, August 5–9, 2007, Courses. 2007. P. 2. URL: https://dx.doi.org/10.1145/1281500.1281643.

185. 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. 1992. P. 115–137. URL: https://doi.org/10.1007/3-540-55693-1_35.

186. 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. 2013. P. 153–164. URL: https://doi.org/10.1007/978-3-642-40285-2_15.

187. Pan C., Zymbler M. Very large graph partitioning by means of parallel DBMS // Advances in Databases and Information Systems - 17th East European Conference, ADBIS 2013, Genoa, Italy, September 1–4, 2013. Proceedings. 2013. P. 388–399. URL: https://doi.org/10.1007/978-3-642-40683-6_29.

188. Pan C., Zymbler M. Encapsulation of partitioned parallelism into open-source database management systems // Programming and Computer Software. 2015. Vol. 41, No. 6. P. 350–360. URL: https://doi.org/10.1134/S0361768815060067.

189. Paranjape-Voditel P., Deshpande U. A DIC-based Distributed Algorithm for Frequent Itemset Generation // JSW. 2011. Vol. 6, No. 2. P. 306–313. URL: https://doi.org/10.4304/jsw.6.2.306-313.

190. Park J.S., Chen M., Yu P.S. An effective hash based algorithm for mining association rules // Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22–25,

1995. 1995. P. 175–186. URL: https://doi.org/10.1145/223784.223813.

191. Park S., Chu W.W., Yoon J., Hsu C. Efficient Searches for Similar Subsequences of Different Lengths in Sequence Databases // ICDE. 2000. P. 23–32. URL: https://doi.org/10.1109/ICDE.2000.839384.

192. Pastukhov R., Korshunov A., Turdakov D.Y., Kuznetsov S.D. Improving quality of graph partitioning using multi-level optimization // Programming and Computer Software. 2015. Vol. 41, No. 5. P. 302–306. URL: https://doi.org/10.1134/S0361768815050096.

193. Pattanaprateep O., McEvoy M., Attia J., Thakkinstian A. Evaluation of rational nonsteroidal anti-inflammatory drugs and gastro-protective agents use; association rule data mining using outpatient prescription patterns // BMC Med. Inf. & Decision Making. 2017. Vol. 17, No. 1. P. 96:1–96:7. URL: https://doi.org/10.1186/s12911-017-0496-3.

194. Paulson L.D. Open source databases move into the marketplace // IEEE Computer. 2004. Vol. 37, No. 7. P. 13–15. URL: https://doi.org/10.1109/MC.2004.62.

195. Pearson K. The problem of the random walk // Nature. 1905. Vol. 72, No. 1865. P. 294. URL: https://doi.org/10.1038/072342a0.

196. Pezoa F., Reutter J.L., Suárez F. et al. Foundations of JSON Schema // Proceedings of the 25th International Conference on World Wide Web, WWW 2016, Montreal, Canada, April 11–15, 2016. 2016. P. 263–273. URL: https://doi.org/10.1145/2872427.2883029.

197. Piatetsky-Shapiro G. Knowledge Discovery in real databases: A report on the IJCAI-89 Workshop // AI Magazine. 1991. Vol. 11, No. 5. P. 68–70. URL: http://www.aaai.org/ojs/index.php/aimagazine/article/view/873.

198. Pothen A., Simon H.D., Liou K.P. Partitioning sparse matrices with eigenvectors of graphs // SIAM Journal Matrix Analytics Applications. 1990. May. Vol. 11, No. 3. P. 430–452.

199. Raghavanand U.N., Réka A., Kumara S. Near linear time algorithm to detect community structures in large-scale networks // Phys. Rev. E. 2007. Vol. 76. P. 036106. URL: https://doi.org/10.1103/PhysRevE.76.036106.

200. Rakthanmanon T., Campana B.J.L., Mueen A. et al. Searching and mining trillions of time series subsequences under Dynamic Time Warping // The 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD'12, Beijing, China, August 12–16, 2012. 2012. P. 262–270. URL: https://doi.org/10.1145/2339530.2339576.

201. Rakthanmanon T., Campana B.J.L., Mueen A. et al. Addressing Big Data time series: mining trillions of time series subsequences under Dynamic Time Warping // TKDD. 2013. Vol. 7, No. 3. P. 10. URL: https://doi.org/10.1145/2500489.

202. Ratanamahatana C.A., Keogh E.J. Three myths about Dynamic Time Warping Data Mining // Proceedings of the 2005 SIAM International Conference on Data Mining, SDM 2005, Newport Beach, CA, USA, April 21–23, 2005. 2005. P. 506–510. URL: https://doi.org/10.1137/1.9781611972757.50.

203. Rebbapragada U., Protopapas P., Brodley C.E., Alcock C. Finding anomalous periodic time series // Machine Learning. 2009. Vol. 74, No. 3. P. 281–313. URL: https://doi.org/10.1007/s10994-008-5093-3.

204. Rechkalov T., Zymbler M. An approach to data mining inside PostgreSQL based on parallel implementation of UDFs // Selected Papers of the XIX International Conference on Data Analytics and Management in Data

Intensive Domains (DAMDID/RCDL 2017), Moscow, Russia, October 9–13, 2017. Vol. 2022. 2017. P. 114–121. URL: http://ceur-ws.org/Vol-2022/paper20.pdf.

205. Rechkalov T., Zymbler M. A study of Euclidean distance matrix computation on Intel many-core processors // 12th International Conference, PCT 2018, Rostov-on-Don, Russia, April 2–6, 2018, Revised Selected Papers. Communications in Computer and Information Science. Vol. 910. 2018. P. 200–215. URL: https://doi.org/10.1007/978-3-319-99673-8_15.

206. Rechkalov T., Zymbler M.L. An approach to Data Mining inside PostgreSQL based on parallel implementation of UDFs // Selected Papers of the XIX International Conference on Data Analytics and Management in Data Intensive Domains (DAMDID/RCDL 2017), Moscow, Russia, October 9–13, 2017. 2017. P. 114–121. URL: http://ceur-ws.org/Vol-2022/paper20.pdf.

207. Rechkalov T.V., Zymbler M.L. Accelerating medoids-based clustering with the Intel Many Integrated Core architecture // Proceedings of the AICT 2015, 9th IEEE International Conference on Application of Information and Communication Technologies, Rostov-on-Don, Russia, October 14–16, 2015. 2015. P. 413–417. URL: https://doi.org/10.1109/ICAICT.2015.7338591.

208. Rechkalov T.V., Zymbler M.L. Integrating DBMS and parallel Data Mining algorithms for modern many-core processors // Data Analytics and Management in Data Intensive Domains - XIX International Conference, DAMDID/RCDL 2017, Moscow, Russia, October 10–13, 2017, Revised Selected Papers. 2017. P. 230–245. URL: https://doi.org/10.1007/978-3-319-96553-6_17.

209. Saad Y. SPARSKIT: A basic tool kit for sparse matrix computations. Technical report RIACS-90-20. NASA Ames Research Center, Mof-

fett Field, CA, 1990. Дата обращения: 05.08.2019. URL: https://www-users.cs.umn.edu/~saad/PDF/RIACS-90-20.pdf.

210. Sakoe H., Chiba S. Dynamic programming algorithm optimization for spoken word recognition // Readings in Speech Recognition / Ed. by A. Waibel, K.-F. Lee. 1990. P. 159–165. URL: http://dl.acm.org/citation.cfm?id=108235.108244.

211. Sakurai Y., Faloutsos C., Yamamuro M. Stream monitoring under the time warping distance // Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15–20, 2007. 2007. P. 1046–1055. URL: https://doi.org/10.1109/ICDE.2007.368963.

212. Sarawagi S., Thomas S., Agrawal R. Integrating mining with relational database systems: alternatives and implications // SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2–4, 1998, Seattle, Washington, USA. 1998. P. 343–354. URL: https://doi.org/10.1145/276304.276335.

213. Sart D., Mueen A., Najjar W.A. et al. Accelerating Dynamic Time Warping subsequence search with GPUs and FPGAs // ICDM 2010, The 10th IEEE International Conference on Data Mining, Sydney, Australia, December 14–17, 2010. 2010. P. 1001–1006. URL: https://doi.org/10.1109/ICDM.2010.21.

214. Savasere A., Omiecinski E., Navathe S.B. An efficient algorithm for mining association rules in large databases // VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11–15, 1995, Zurich, Switzerland. 1995. P. 432–444. URL: http://www.vldb.org/conf/1995/P432.PDF.

215. Schlegel B., Karnagel T., Kiefer T., Lehner W. Scalable frequent itemset mining on many-core processors // Proceedings of the Ninth International

Workshop on Data Management on New Hardware, DaMoN 2013, New York, NY, USA, June 24, 2013. 2013. P. 3. URL: https://doi.org/10.1145/2485278.2485281.

216. Shabib A., Narang A., Niddodi C.P. et al. Parallelization of searching and mining time series data using Dynamic Time Warping // 2015 International Conference on Advances in Computing, Communications and Informatics, ICACCI 2015, Kochi, India, August 10–13, 2015. 2015. P. 343–348. URL: https://doi.org/10.1109/ICACCI.2015.7275633.

217. Shanahan J.G., Dai L. Large scale distributed data science using Apache Spark // Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Sydney, NSW, Australia, August 10–13, 2015. 2015. P. 2323–2324. URL: https://doi.org/10.1145/2783258.2789993.

218. Shanmugapriya S., Valarmathi A. Efficient fuzzy c-means based multilevel image segmentation for brain tumor detection in MR images // Design Autom. for Emb. Sys. 2018. Vol. 22, No. 1-2. P. 81–93. URL: https://doi.org/10.1007/s10617-017-9200-1.

219. Sharanyan S., Arvind K., Rajeev G. Implementing the Dynamic Time Warping algorithm in multithreaded environments for real time and unsupervised pattern discovery // ICCCT 2011, The 2nd IEEE International Conference on Computer and Communication Technology, Allahabad, India, September 15–17, 2011. 2011. P. 394–398. URL: https://doi.org/10.1109/ICCCT.2011.6075111.

220. Silberschatz A., Stonebraker M., Ullman J.D. Database systems: achievements and opportunities // Commun. ACM. 1991. Vol. 34, No. 10. P. 110–120. URL: https://doi.org/10.1145/125223.125272.

221. Silva Y.N., Aref W.G., Ali M.H. Similarity Group-By // Proceedings of the 25th International Conference on Data Engineering, ICDE 2009,

March 29, 2009 – April 2, 2009, Shanghai, China. 2009. P. 904–915. URL: https://doi.org/10.1109/ICDE.2009.113.

222. Sokolinsky L.B. Organization of parallel query processing in multiprocessor database machines with hierarchical architecture // Programming and Computer Software. 2001. Vol. 27, No. 6. P. 297–308. URL: https://doi.org/10.1023/A:1012706401123.

223. Sokolinsky L.B. LFU-K: An effective buffer management replacement algorithm // Database Systems for Advances Applications, 9th International Conference, DASFAA 2004, Jeju Island, Korea, March 17–19, 2004, Proceedings. 2004. P. 670–681. URL: https://doi.org/10.1007/978-3-540-24571-1_60.

224. Stefano C.D., Maniaci M., Fontanella F., di Freca A.S. Reliable writer identification in medieval manuscripts through page layout features: The "Avila" Bible case // Eng. Appl. of AI. 2018. Vol. 72. P. 99–110. URL: https://doi.org/10.1016/j.engappai.2018.03.023.

225. Stonebraker M., Kemnitz G. The Postgres next generation database management system // Commun. ACM. 1991. Vol. 34, No. 10. P. 78–92. URL: https://doi.org/10.1145/125223.125262.

226. Stonebraker M., Madden S., Dubey P. Intel "Big Data" Science and Technology Center vision and execution plan // SIGMOD Record. 2013. Vol. 42, No. 1. P. 44–49. URL: https://doi.org/10.1145/2481528.2481537.

227. Strohmaier E., Meuer H.W., Dongarra J., Simon H.D. The TOP500 list and progress in high-performance computing // IEEE Computer. 2015. Vol. 48, No. 11. P. 42–49. URL: https://doi.org/10.1109/MC.2015.338.

228. Sui X., Nguyen D., Burtscher M., Pingali K. Parallel graph partitioning on multicore architectures // Languages and Compilers for Parallel Com-

puting - 23rd International Workshop, LCPC 2010, Houston, TX, USA, October 7–9, 2010. Revised Selected Papers. 2010. P. 246–260. URL: https://doi.org/10.1007/978-3-642-19595-2_17.

229. Sun P., Huang Y., Zhang C. Cluster-By: An efficient clustering operator in emergency Management Database Systems // Proceedings of the Web-Age Information Management International Workshops, WAIM 2013 : HardBD, MDSP, BigEM, TMSN, LQPM, BDMS, Beidaihe, China, June 14–16, 2013. 2013. P. 152–164. URL: https://doi.org/10.1007/978-3-642-39527-7_17.

230. Takahashi N., Yoshihisa T., Sakurai Y., Kanazawa M. A parallelized data stream processing system using Dynamic Time Warping distance // 2009 International Conference on Complex, Intelligent and Software Intensive Systems, CISIS 2009, Fukuoka, Japan, March 16–19, 2009. 2009. P. 1100–1105. URL: https://doi.org/10.1109/CISIS.2009.77.

231. Tang L., Liu H. Graph mining applications to social network analysis // Managing and mining graph data / Ed. by C.C. Aggarwal, H. Wang. Springer, 2010. Vol. 40 of Advances in Database Systems. P. 487–513. URL: https://doi.org/10.1007/978-1-4419-6045-0_16.

232. Tang Z., Maclennan J., Kim P.P. Building Data Mining solutions with OLE DB for DM and XML for analysis // SIGMOD Record. 2005. Vol. 34, No. 2. P. 80–85. URL: https://doi.org/10.1145/1083784.1083805.

233. Tarango J., Keogh E.J., Brisk P. Instruction set extensions for Dynamic Time Warping // Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2013, Montreal, QC, Canada, September 29 – October 4, 2013. 2013. P. 18:1–18:10. URL: https://doi.org/10.1109/CODES-ISSS.2013.6659005.

234. Trifunovic A., Knottenbelt W.J. Towards a parallel disk-based algorithm for multilevel k-way hypergraph partitioning // 18th International Parallel and Distributed Processing Symposium (IPDPS 2004), Proceedings, 26–30 April 2004, Santa Fe, New Mexico, USA. 2004. URL: https://doi.org/10.1109/IPDPS.2004.1303286.

235. Turner V., Gantz J., Reinsel D., et al. The Digital Universe of opportunities: rich data and the increasing value of the Internet of Things. 2014. URL: http://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm.

236. Tuzcu V., Nas S. Dynamic time warping as a novel tool in pattern recognition of ECG changes in heart rhythm disturbances // Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, Waikoloa, Hawaii, USA, October 10–12, 2005. 2005. P. 182–186. URL: https://doi.org/10.1109/ICSMC.2005.1571142.

237. Waas F.M. Beyond Conventional Data Warehousing - Massively Parallel Data Processing with Greenplum Database // Business Intelligence for the Real-Time Enterprise - Second International Workshop, BIRTE 2008, Auckland, New Zealand, August 24, 2008, Revised Selected Papers. 2008. P. 89–96. URL: https://doi.org/10.1007/978-3-642-03422-0_7.

238. Wale N., Ning X., Karypis G. Trends in chemical graph Data Mining // Managing and Mining Graph Data / Ed. by C.C. Aggarwal, H. Wang. Springer, 2010. Vol. 40 of Advances in Database Systems. P. 581–606. URL: https://doi.org/10.1007/978-1-4419-6045-0_19.

239. Wang H., Zaniolo C., Luo C. ATLAS: A small but complete SQL extension for Data Mining and data streams // VLDB. 2003. P. 1113–1116. URL: http://www.vldb.org/conf/2003/papers/S37P01.pdf.

240. Wang W., Yang J., Muntz R.R. STING: A statistical information grid approach to spatial Data Mining // VLDB'97, Proceedings of 23rd In-

ternational Conference on Very Large Data Bases, August 25–29, 1997, Athens, Greece. 1997. P. 186–195. URL: http://www.vldb.org/conf/1997/P186.PDF.

241. Wang Z., Huang S., Wang L. et al. Accelerating subsequence similarity search based on Dynamic Time Warping distance with FPGA // The 2013 ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA '13, Monterey, CA, USA, February 11–13, 2013. 2013. P. 53–62. URL: https://doi.org/10.1145/2435264.2435277.

242. Wasserman S., Faust K. Social Network Analysis: Methods and Applications. Cambridge University Press, 1994. URL: https://doi.org/10.1017/CBO9780511815478.

243. Watts D.J., Strogatz S.H. Collective dynamics of "small-world" networks // Nature. 1998. Vol. 393, No. 6684. P. 440–442. URL: https://dx.doi.org/10.1038/30918.

244. Widenius U.M., Axmark D. MySQL reference manual - documentation from the source. O'Reilly, 2002. URL: http://www.oreilly.de/catalog/mysqlref/.

245. Woodbridge D.M., Wilson A.T., Rintoul M.D., Goldstein R.H. Time series discord detection in medical data using a parallel relational database // 2015 IEEE International Conference on Bioinformatics and Biomedicine, BIBM 2015, Washington, DC, USA, November 9–12, 2015. 2015. P. 1420–1426. URL: https://doi.org/10.1109/BIBM.2015.7359885.

246. Wu F., Wu Q., Tan Y. et al. A Vectorized K-Means Algorithm for Intel Many Integrated Core Architecture // Advanced Parallel Processing Technologies - 10th International Symposium, APPT 2013, Stockholm, Sweden, August 27–28, 2013, Revised Selected Papers. 2013. P. 277–294. URL: https://doi.org/10.1007/978-3-642-45293-2_21.

247. Wu X., Kumar V., Quinlan J.R. et al. Top 10 algorithms in Data Mining // Knowledge and Information Systems. 2008. Vol. 14, No. 1. P. 1–37. URL: https://doi.org/10.1007/s10115-007-0114-2.

248. Wu Y., Zhu Y., Huang T. et al. Distributed discord discovery: Spark based anomaly detection in time series // Proceedings of the 17th IEEE International Conference on High Performance Computing and Communications, HPCC 2015, 7th IEEE International Symposium on Cyberspace Safety and Security, CSS 2015, and 12th IEEE International Conference on Embedded Software and Systems, ICESS 2015, New York, NY, USA, August 24–26, 2015. 2015. P. 154–159. URL: https://doi.org/10.1109/HPCC-CSS-ICESS.2015.228.

249. Xhafa F., Bogza A., Caballé S., Barolli L. Apache Mahout's k-Means vs Fuzzy k-Means performance evaluation // 2016 International Conference on Intelligent Networking and Collaborative Systems, INCoS 2016, Ostrawva, Czech Republic, September 7–9, 2016. 2016. P. 110–116. URL: https://doi.org/10.1109/INCoS.2016.103.

250. Xi X., Keogh E.J., Shelton C.R. et al. Fast time series classification using numerosity reduction // Proceedings of the 23rd International Conference on Machine Learning, ICML 2006, Pittsburgh, Pennsylvania, USA, June 25–29, 2006. 2006. P. 1033–1040. URL: https://doi.org/10.1145/1143844.1143974.

251. Xie J., Kelley S., Szymanski B.K. Overlapping community detection in networks: The state-of-the-art and comparative study // ACM Comput. Surv. 2013. Vol. 45, No. 4. P. 43:1–43:35. URL: https://doi.org/10.1145/2501654.2501657.

252. Yankov D., Keogh E.J., Rebbapragada U. Disk Aware Discord Discovery: Finding Unusual Time Series in Terabyte Sized Datasets // Proceedings of the 7th IEEE International Conference on Data Mining (ICDM 2007),

October 28–31, 2007, Omaha, Nebraska, USA. 2007. P. 381–390. URL: https://doi.org/10.1109/ICDM.2007.61.

253. Yankov D., Keogh E.J., Rebbapragada U. Disk aware discord discovery: finding unusual time series in terabyte sized datasets // Knowl. Inf. Syst. 2008. Vol. 17, No. 2. P. 241–262. URL: https://doi.org/10.1007/s10115-008-0131-9.

254. Zaharia M., Chowdhury M., Franklin M.J. et al. Spark: Cluster Computing with Working Sets // 2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010. 2010. URL: https://www.usenix.org/legacy/event/hotcloud10/tech/full_papers/Zaharia.pdf.

255. Zaki M.J. Scalable algorithms for association mining // IEEE Trans. Knowl. Data Eng. 2000. Vol. 12, No. 3. P. 372–390. URL: https://doi.org/10.1109/69.846291.

256. Zeng Z., Wu B., Wang H. A parallel graph partitioning algorithm to speed up the large-scale distributed graph mining // Proceedings of the 1st International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications, BigMine 2012, Beijing, China, August 12, 2012. 2012. P. 61–68. URL: https://doi.org/10.1145/2351316.2351325.

257. Zhang M. Application of Data Mining technology in digital library // JCP. 2011. Vol. 6, No. 4. P. 761–768. URL: https://doi.org/10.4304/jcp.6.4.761-768.

258. Zhang Y., Adl K., Glass J.R. Fast spoken query detection using lower-bound Dynamic Time Warping on Graphical Processing Units // 2012 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2012, Kyoto, Japan, March 25–30, 2012. 2012. P. 5173–5176. URL: https://doi.org/10.1109/ICASSP.2012.6289085.

259. Zheng Z., Kohavi R., Mason L. Real world performance of association rule algorithms // Proceedings of the 7th ACM SIGKDD international conference on Knowledge discovery and data mining, San Francisco, CA, USA, August 26–29, 2001. 2001. P. 401–406. URL: https://doi.org/10.1145/502512.502572.

260. Zhou J. Hash join // Encyclopedia of Database Systems / Ed. by L. Liu, M.T. Özsu. Springer. 2009. P. 1288–1289. URL: https://doi.org/10.1007/978-0-387-39940-9_869.

261. Zhou J. Nested loop join // Encyclopedia of Database Systems / Ed. by L. Liu, M.T. Özsu. Springer. 2009. P. 1895. URL: https://doi.org/10.1007/978-0-387-39940-9_868.

262. Zhou J. Sort-merge join // Encyclopedia of Database Systems / Ed. by L. Liu, M.T. Özsu. Springer. 2009. P. 2673–2674. URL: https://doi.org/10.1007/978-0-387-39940-9_867.

263. Zou J., Chen L., Chen C.L.P. Ensemble fuzzy c-means clustering algorithms based on KL-Divergence for medical image segmentation // 2013 IEEE International Conference on Bioinformatics and Biomedicine, Shanghai, China, December 18–21, 2013. 2013. P. 291–296. URL: https://doi.org/10.1109/BIBM.2013.6732505.

264. Zymbler M. Best-match time series subsequence search on the Intel Many Integrated Core architecture // Advances in Databases and Information Systems - 19th East European Conference, ADBIS 2015, Poitiers, France, September 8–11, 2015, Proceedings. 2015. P. 275–286. URL: https://doi.org/10.1007/978-3-319-23135-8_19.

265. Zymbler M. Accelerating dynamic itemset counting on Intel many-core systems // 40th International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2017,

Opatija, Croatia, May 22–26, 2017. 2017. P. 1343–1348. URL: https://doi.org/10.23919/MIPRO.2017.7973631.

266. Zymbler M. Parallel algorithm for frequent itemset mining on Intel many-core systems // CIT. Journal of Computing and Information Technology. 2019. Vol. 26, No. 4. P. 209–221. URL: https://doi.org/10.20532/cit.2018.1004382.

267. Перечень оборудования Центра коллективного пользования Сибирского Суперкомпьютерного Центра ИВМиМГ СО РАН. Дата обращения: 03.11.2018. URL: http://www.sscc.icmmg.nsc.ru/hardware.html.

268. Advances in Knowledge Discovery and Data Mining, Ed. by U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, R. Uthurusamy. AAAI/MIT Press, 1996.

269. IBM Quest Synthetic Data Generator. Дата обращения: 03.11.2018. URL: https://ibmquestdatagen.sourceforge.io/.

270. Intel Math Kernel Library 2018 Release Notes. Дата обращения: 03.11.2018. URL: https://software.intel.com/en-us/articles/intel-math-kernel-library-intel-mkl-2018-release-notes.

271. KDD Cup 1999 Data. Дата обращения: 01.07.2019. URL: http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html.

272. Managing and Mining Graph Data, Ed. by C.C. Aggarwal, H. Wang. Springer, 2010. Vol. 40 of Advances in Database Systems. URL: https://doi.org/10.1007/978-1-4419-6045-0.

273. Proceedings of the 11th International Joint Conference on Artificial Intelligence. Detroit, MI, USA, August 1989 / Ed. by N.S. Sridharan. Morgan Kaufmann, 1989. URL: http://ijcai.org/proceedings/1989-1.

274. Threadpool Library for C++ ver. 0.2.5. Дата обращения: 03.11.2018. URL: http://threadpool.sourceforge.net.