УДК 004.657

В. С. Шевский, Ю. А. Шичкина
Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В. И. Ульянова (Ленина)

Анализ способов параллельного выполнения запросов к реляционным базам данных

Предлагается подход к выполнению запроса, заключающийся в изменении исходного запроса и его выполнении с использованием библиотеки параллельных вычислений. Для изменения исходного запроса предлагается применять информационные графы. Для преобразования информационного графа предлагается использовать методы оптимизации по ширине и высоте, изначально предназначенные для классических алгоритмов, но адаптированные под запросы. Также приводятся результаты тестирования запросов различной сложности по структуре. Показывается преимущество автоматического распараллеливания перед распараллеливанием вручную. Рассматриваются варианты распараллеливания с учетом параллелизма по задачам (при дефрагментации запроса на подзапросы) и параллелизма по данным (при дефрагментации таблиц реляционной базы данных). Тестирование проводилось на базе данных из 5 таблиц и сложном запросе из 29 подзапросов. Результаты показывают, что подход, основанный на информационных графах, может быть применен при оптимизации запросов к реляционным базам данных.

База данных, система управления базой данных, запрос, многопоточное программирование

Огромное количество программных проектов от самых различных организаций активно используют системы управления базами данных (СУБД). Разнообразие СУБД сегодня велико и способно удовлетворить самые разные потребности в области обработки данных.

По сведениям DB-Engines, организации по сбору и представлению информации о СУБД различных парадигм (SQL, NoSQL, NewSQL), публикующей ежемесячные результаты исследований популярности СУБД, несмотря на рост востребованности NoSQL- и NewSQL-баз данных, первенство по применению по-прежнему удерживают реляционные базы данных [1].

Реляционные базы данных успешно справляются со своими главными задачами, а именно:

– хранение новых записей;

– считывание данных;

– поиск данных;

– защита данных от несанкционированного использования.

Однако с ростом объема данных многие реляционные базы данных начинают сталкиваться с проблемой низкой скорости считывания данных.

Глобальным решением стало изменение принципа хранения и считывания данных, которое породило новое поколение СУБД – NoSQL. Данный принцип предполагает хранение данных в виде совокупности файлов, например в формате JSON. Такой подход позволил значительно ускорить обработку запросов даже при условии высокой нагрузки сервера. Однако NoSQL-системы имеют также и существенные недостатки. Во-первых, ориентируясь на скорость обработки запросов, они частично отказались от требования ACID. Это означает, что NoSQL-базы данных не могут обеспечить такую же надежность работы, как в реляционных БД. Во-вторых, они имеют свой язык запросов. И хотя он схож по синтаксису с SQL, его введение имеет несколько недостатков, среди которых:

– разнообразие языков для каждого типа БД, что неизбежно заставляет пользователя текущей системы NoSQL учить новый язык;

– отсутствие поддержки оператора join;

– в целом небогатый объем возможностей языков по сравнению с SQL;

– отсутствие таких инструментов, как хранимые процедуры и т. п.

Кроме SQL и NoSQL существует еще целый мир NewSQL. Это базы данных, которые взяли новые подходы распределенных систем от NoSQL и оставили реляционную модель представления данных и язык запросов SQL. Эти БД возникли

буквально за последнее десятилетие, но уже интенсивно борются за пользователей, оттесняя на рынке и старые добрые SQL БД, и чуть менее старые NoSQL. Важно то, что рассматривая класс NewSQL, все равно приходится рассматривать реляционные модели. А значит, проблема оптимизации запросов и поиска новых методов ускорения их выполнения с использованием вычислительных систем, допускающих параллельные вычисления, актуальна для NewSQL в той же мере, что и для реляционных СУБД.

В данной статье представлены результаты оптимизации запросов к реляционной базе данных по времени выполнения за счет поиска оптимального плана их реализации на вычислительных системах, допускающих параллельные вычисления. В качестве примеров рассмотрены различные SQL-запросы и представлены результаты исследования поведения базы данных при запуске их последовательных и распараллеленных планов выполнения.

Существующие исследования в области оптимизации доступа к данным. В связи с тем, что своевременность и качество обработки информации – это своего рода ключ к успеху любой организации, а объемы информации имеют тенденцию к непрерывному массовому увеличению, то естественно, что по различным аспектам управления потоками данных проводилось очень много исследований еще со времен появления первого компьютера. Все эти исследования можно разделить на 2 основных направления:

– аппаратная оптимизация хранения данных;

– оптимизация алгоритмов обработки и доступа к данным.

Рассмотрим каждое из них.

Аппаратная оптимизация хранения данных. Так как статья посвящена алгоритмическим методам ускорения процесса обработки данных, не будем подробно останавливаться на аппаратных методах хранения данных. Отметим только, что эта область также непрерывно развивается. Так перспективным направлением является развитие квантовых компьютеров, которые повышают скорость вычисления алгоритмов, запускаемых на классическом компьютере [2]. В [3] авторы предлагают гибридную архитектуру вычислительной системы для хранения неструктурированных данных. Новая система индексирования таблиц БД, которая используется на гибридных дисках, предлагается в [4]. Основная идея, описываемая в [4], – это уменьшение случайных записей в память (запись в случайные блоки памяти). Чтобы добиться этого, авторами было использовано асимметричное свойство ввода-вывода SSD. Итогом работы авторов стала разработанная БД, которая получила название HybridDB tree. Приведенные результаты показали, что производительность базы выше, чем при использовании алгоритма B+-tree на жестком диске SSD/HDD без оптимизации. Возможности использования SSD и RAM в качестве основного источника хранения данных для современных систем также рассматриваются в [5].

Оптимизация алгоритмов обработки и доступа к данным. Авторы [6] предлагают масштабируемый метод в стиле MapReduce, называемый ICBL, который сокращает время выполнения запросов в базе данных neo4j.

Решению проблемы непрерывной обработки запросов в распределенных средах посвящены исследования в [7]. Они представляют 2 алгоритма, которые посредством анализа данных позволяют уменьшить объем данных для обработки и тем самым снизить затраты на связь между объектами базы данных и сами вычисления.

Появление технологий мобильных вычислений обеспечивает возможность доступа к информации в любое время и в любом месте. Однако поскольку мобильные вычислительные среды обладают такими неотъемлемыми характеристиками, как энергопотребление, объемы хранимых данных, стоимость связи и пропускная способность, то для мобильных вычислений эффективная обработка запросов и минимальное время отклика запросов представляют особенно большой интерес. В [8] представлены способы оптимизации запросов в мобильных базах данных по двум основным категориям: стратегия обработки запросов и стратегия управления кешированием. Авторы статьи описывают несколько методов улучшения производительности запросов посредством передачи данных конечным пользователям.

Модель для преобразования SQL-запросов в структуру дерева запросов, предназначенную для выполнения в многопроцессорной среде, поддерживающей конвейер, предлагают авторы [9]. На последнем этапе представления запроса без каких-либо дополнительных затрат ими выполняются некоторые элементарные шаги по оптимизации запроса, и поэтому последующая оптимизация может быть сфокусирована на более сложных шагах.

Все конструкции SQL еще с 90-х гг. пытались представлять с помощью графов, например операторных графов [10], графов, в которых узлами являются операторы реляционной алгебры [11], и др. [12], [13].

Для представления подзапросов с помощью графов применяются различные подходы. В System R [9] блок запросов представляет собой узел дерева и рассматривается как единица, для которой выбран оптимизированный план выполнения. Однако внутренний блок должен быть проанализирован до анализа внешнего блока, содержащего внутренний блок как подзапрос. Показано, что это приводит к низкой производительности [9].

Широко признанным эффективным инструментом для крупномасштабного анализа данных является система MapReduce. Эта система обеспечивает высокую производительность за счет использования параллелизма между обрабатывающими узлами, обеспечивая простой интерфейс для приложений верхнего уровня. Некоторые поставщики улучшили свои системы хранилищ данных, интегрировав MapReduce в свои системы. Однако существующие системы обработки запросов на основе MapReduce, такие, как Hive, не дают хороших результатов для традиционных систем баз данных. В [14] предлагается схема оптимизации запросов на основе MapReduce. В частности, авторы внедряют в Hive оптимизатор запросов, который предназначен для создания эффективного плана запросов на основе модели затрат.

В последнее время интенсивные исследования проводятся по части оптимизации запросов к графическим базам данных [15], [16].

Таким образом, несмотря на разнообразие и количество проводимых исследований в области усовершенствования процесса получения данных из различных баз данных и создание оптимизаторов запросов, в большинстве реляционных СУБД на текущий момент методик и программного обеспечения по эквивалентному преобразованию запросов к форме, наиболее эффективно реализуемой на вычислительных системах, допускающих параллельное исполнение запросов, очень мало и они далеки от совершенства.

Постановка задачи. Входными данными является запрос, записанный на SQL и выполняемый последовательно (назовем его *исходный* запрос). Выходными данными является план выполнения исходного запроса на параллельной вычислительной системе.

Задача заключается в разработке метода, который бы позволял проанализировать исходный запрос и построить план его выполнения на параллельной вычислительной системе.

В данной статье рассмотрены планы выполнения запросов, которые подготавливались вручную. Однако в дальнейшем предполагается написание и использование алгоритма автоматического создания плана выполнения запроса.

Пусть после выполнения исходного запроса получается множество данных S_0. Тогда после успешного выполнения параллельного плана запроса должны быть получены непересекающиеся множества данных S_1–S_n (где S_n – это множество параллельных ветвей плана), при объединении которых получаем исходное множество: $S_0 = S_1 \cup \cup \ldots \cup S_n$ (рис. 1).

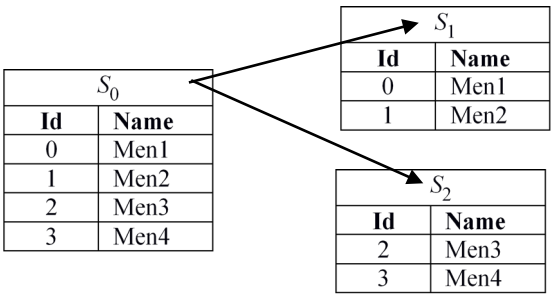


Рис. 1

Подзапрос каждой параллельной ветви плана должен считывать свою область базы данных, благодаря чему исключается сценарий блокировки данных на чтение в системах, которые поддерживают эту функцию.

Способы организации параллельного выполнения запросов. Главная цель исследования – установить, будет ли наблюдаться прирост производительности при параллельном запуске нескольких подзапросов исходного запроса по сравнению с последовательным запуском исходного запроса. В качестве тестируемой базы данных была выбрана MySQL-база данных, имеющая следующую архитектуру (рис. 2).

Таблицы работают на подсистеме низкого уровня InnoDB.

Объем таблиц следующий:

– Table1: 565825;
– Table2: 565819;
– Table3: 318735;
– Table4: 565818;
– Table5: 318735.

Технические характеристики процессора:

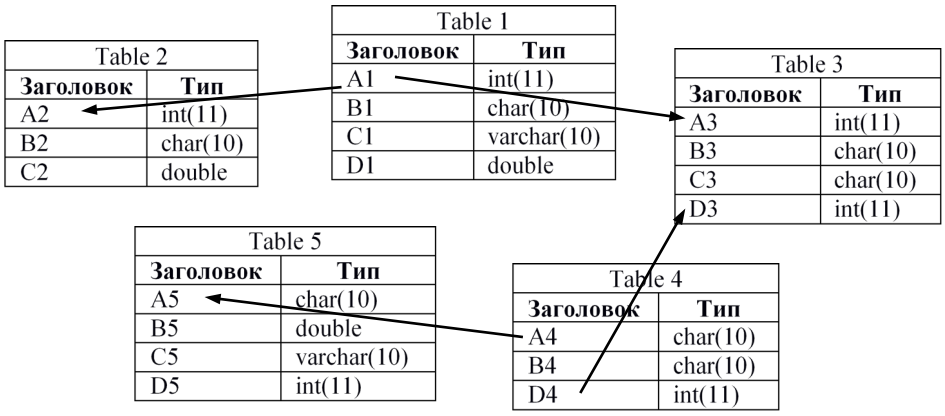


Рис. 2

– intel core i5 4210U;

– количество ядер – 2, количество потоков– 4;

– частота процессора по умолчанию – 1.7 ГГц, в turbo bust – 2.4 Ггц;

– кешы – 2x L1 32 Кбайт, L2 256 Кбайт, L3 3072 Кбайт;

– ОЗУ – 8058.788 Мбайт;

– OS – ubuntu 16.04.

Для проведения тестирования была написана программа на языке java. Взаимодействие с базой данных организовано было с помощью вызовов программных функций драйвера jdbc. Распараллеливание запросов осуществлялось с помощью базовых потоков, представленных в java как объекты класса Thread, который входит в состав java SE.

Тестирование проводилось по двум направлениям:

– тестирование запросов, состоящих из одного или нескольких конструкций Select, но направленных на значительный объем выбираемых данных из базы;

– тестирование больших SQL-запросов, составленных из некоторого множества взаимосвязанных подзапросов.

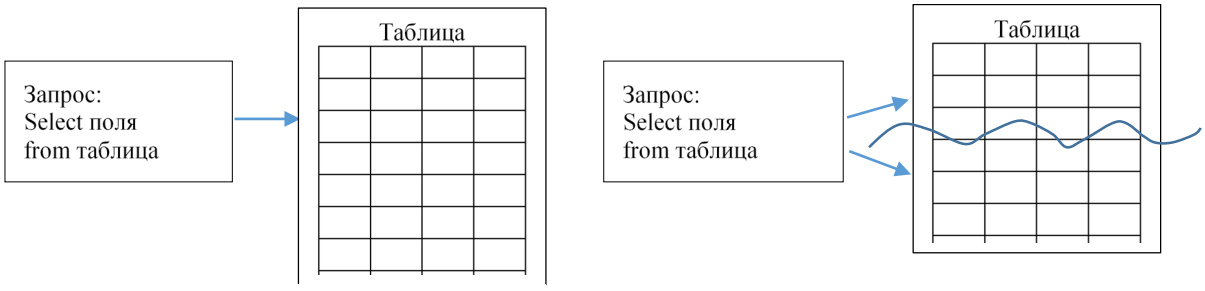
Результаты тестирования приведены далее.

Параллельный запуск запросов, состоящих из одного или нескольких конструкций Select. Пусть дана совокупность SQL-запросов $Q = \{Q_1, Q_2, ..., Q_n\}$. Все запросы Q_i ($i = 1, 2, \ldots, n$) состоят из одной конструкции Select и обращаются к одной таблице. Одна из схем параллельного запуска таких запросов заключается в следующем:

1. Каждому из запросов Q_i поставить в соответствие пару запросов SQ_{i1}, SQ_{i2}. В результате будет получено множество подзапросов $S = \{S_{11}, S_{12}, S_{21}, S_{22}, \ldots, S_{n1}, S_{n2}\}$. Всего запросов во множестве S будет $m = n \cdot 2$.

2. Провести последовательное выполнение пар подзапросов S.

Подзапросы S_{ij} ($i = 1, 2, \ldots, n$; $j = 1, 2$) получаются из запроса Q_i ($i = 1, 2, \ldots, n$) введением условия Where в запросах Q_i, в котором указывается, по какому принципу ведется условное разделение таблицы на несколько частей для обработки. Схематично это изображено на рис. 3, *б*. В данном случае в условии прописывается принцип, по которому все записи таблицы можно однозначно разбить на две примерно одинаковые по объему части. Например: если исходный запрос имеет вид SELECT * FROM TABLE и в таблице есть первичный ключ A типа счетчика, то этот запрос может быть разбит на 2 следующих подзапроса:



а *б*

Рис. 3

1. SELECT * FROM TABLE WHERE A < = Count / 2;

2. SELECT * FROM TABLE WHERE A > Count / 2,

где Count – количество записей в таблице Table; A – ключевое поле-счетчик таблицы Table.

Для всех методов в программе на языке java каждый из полученных подзапросов в коде инициализируется в отдельном потоке (объекте Thread). После формирования потоков происходит их запуск в цикле.

Параллельный запуск сложных запросов, состоящих из множества подзапросов. Пусть дан SQL-запрос Q, состоящий из множества подзапросов $S = \{S_1, S_2, \ldots, S_n\}$, где n – число подзапросов. Все подзапросы S_i ($i = 1, 2, \ldots, n$) состоят из одной конструкции Select. Число таблиц, к которым обращается подзапрос S, в данной контексте не важно. Одна из схем параллельного запуска таких запросов заключается в следующем:

1. Разбить запрос Q на совокупность подзапросов $S' = \{S_{1'}, S_{2'}, ..., S_{m'}\}$, $m \le n$, m – число подзапросов S', каждый из которых может быть либо равен некоторому запросу S_i ($i = 1, 2, \ldots, n$), или содержит S_i ($i = 1, 2, \ldots, n$) в качестве подзапроса.

2. Составить расписание выполнения подзапросов S' на заданном числе вычислительных узлов.

При параллельном выполнении сложного запроса Q, как и в случае простого запроса, каждый подзапрос исполняется в своем потоке Thread в целевом приложении на языке Java.

Примечания:

1. Оба рассмотренных варианта параллельного выполнения простого и сложного запросов подразумевают наличие точек синхронизации и агрегирования данных.

2. Оба варианта могут быть объединены для достижения наибольшей скорости выполнения запросов.

Тестирование. ***Тестирование запросов, состоящих из одного или нескольких конструкций Select.*** Для тестирования схемы распараллеливания простых запросов по данным были выбраны следующие запросы с учетом схемы базы данных на рис. 2:

1) select * from Table2 where C2 > = any (select B5 from Table5);

2) select * from Table2 where B2 not in (select B3 from Table3);

3) select Table2.A2, Table2.B2, Table2.C2 from Table2, Q5 where Table2.A2 = Q5.A2 and Table2.B2 = Q5.B2;

4) select Table2.A2, Table2.B2, Table2.C2 from Table2, (select W1.A2, W1.B2 from (select A2, B2 from Table2 where C2 > = all (select B5 from Table5)) as W1, (select A2, B2 from Table2 where B2 not in (select B3 from Table3)) as W2 where W1.A2 = W2.A2 and W1.B2 = W2.B2) as W where Table2.A2 = W.A2 and Table2.B2 = W.B2;

5) select * from Table2 where C2 > = all (select B5 from Table5) and B2 not in (select B3 from Table3);

6) select * from Table2.

По результатам упрощенного тестирования построена диаграмма (рис. 4), где по оси Oy показано время выполнения запроса в наносекундах.

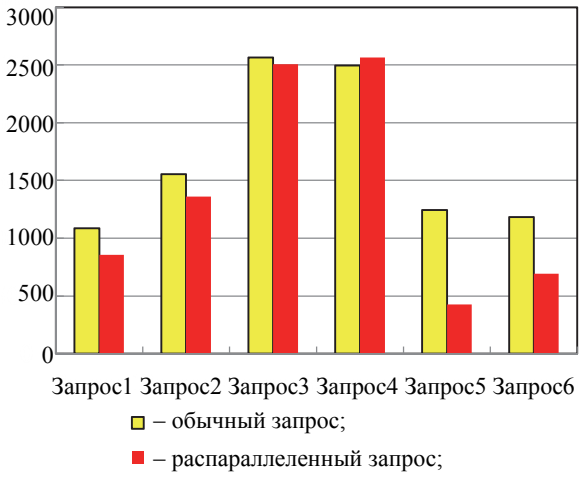


Рис. 4

Видно, что распараллеленный запрос оказался эффективнее для всех запросов, кроме 4-го. Запрос 4 выполнился медленнее обычного запроса 4, так как подзапросы S_{41} и S_{42}, ссылаясь на общие участки памяти, создавали конфликты.

Тестирование сложных запросов, состоящих из множества подзапросов. При тестировании схемы распараллеливания сложных запросов за основу был взят запрос

$$Q = S_1 \cup S_2 \cup \ldots \cup S_n,$$

где $n = 29$, $S = \{S_1, S_2, \ldots, S_n\}$ – множество подзапросов запроса Q:

S_1: Select a5 From Table5 Where f5> = value;

S_2: Select a4 From Table4 Where a4<>b4;

S_3: Select * From Table4, Table5 Where a4 = a5 and a5 in S_1 and a4 in S_2;

S_4: Select a5 From Table5 Where value1 < = b5 and b5< = value2;

S_5: Select a4 From Table4 Where d4> = (Select avg(f5) From Table5);

S'_5 : Select avg(f5) From Table5;

S_6: Select * From Table4, Table5 Where a4 = a5 and a4 in S_5 and a5 in S_4;

S_7: Select * From Table1 Where c1 like '*a*';

S_8: Select * From Table2, S_7 Where a1 = $a2$ and c2> = (Select min(d1) From Table1);

S_9: Select min(d1) From Table1;

S_{10}: Select * From Table3, S_8 Where a1 = a3;

S_{11}: Select * From Table1 Where c1 like '*v*';

S_{12}: Select * From Table3 Where c3 like '*v*';

S_{13}: Select a1 as a13 From S_{11}, S_{12} Where a1 = a3;

S_{14}: Select * From S_{13};

S_{15}: Select * From S_{10} Where a1 not in S_{14};

S_{16}: Select a3 as a17 From Table3, Table4 Where d3 = d4 and a3< = d3;

S'_{16} : Select a17 From S_{16};

S_{17}: Select a5 From S_6;

S_{18}: Select * From S_3 Where a5 not in S_{17};

S_{19}: S_{20} Union S_{21};

S_{20}: Select avg(d1) as d11 From Table1;

S_{21}: Select avg(c2) as d11 From Table2;

S_{22}: Select * From S_{15} Where (d1 + c2)>= all(S_{23});

S_{23}: Select max(d11) From S_{19};

S_{24}: Select * From S_{15} Where a1 in S_{25} and a1 in S'_{16} ;

S_{25}: Select a17 From S_{10};

S_{26}: Select * From S_{24}, S_{18} Where d4 = d5;

S_{27}: Select avg(a17)*2 as a18 From S_{17};

S_{28}: Select * From S_{22} Where a3 < = S_{27};

S_{29}: Select a4, b4, d4, a5, b5, e5, f5, S_{26}.a3, S_{26}.b3, S_{26}.c3, S_{26}.d3, S_{26}.a2, S_{26}.b2, S_{26}.c2, S_{26}.a1, S_{26}.b1, S_{26}.c1, S_{26}.d1 From S_{26}, S_{28} Where S_{28}.a3 = S_{26}.a3.

Для распараллеливания запроса Q было выбрано расписание:

1. Выделение множества подзапросов $S = \{S_1, S_2, \ldots, S_n\}$.

2. Запуск каждого подзапроса S_i в своем потоке Thread.

3. Сбор результатов.

Сложность заключается в том, что вручную проанализировать такое число запросов невозможно.

Автоматическое распараллеливание запросов. Выходом из этой проблемы является применение теории графов. Поставим в соответствие запросу Q граф зависимостей по данным между подзапросами S_i (i = 1, 2, …, n) (рис. 5). Символом T обозначены таблицы (Table).

Применим к этому графу метод распараллеливания на основе списков смежности, описанный в [17].

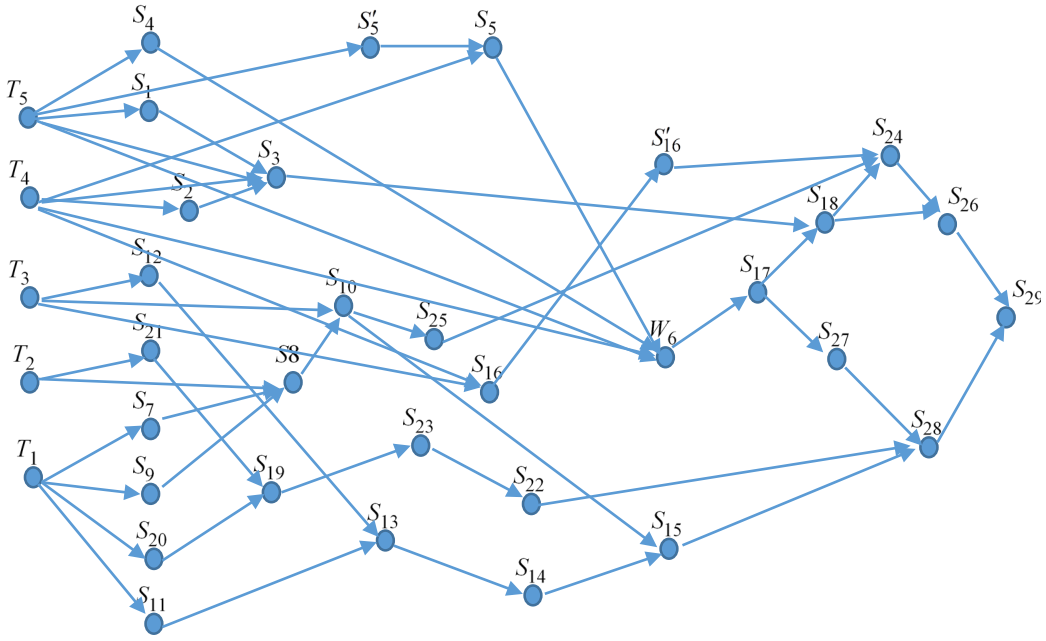


Рис. 5

В результате получим следующее расписание для подзапросов S_i ($i = 1, 2, \ldots, n$):

t_0: $S_{16}, S_1, S_2, S_4, S'_5, S_{20}, S_7, S_9, S_{21}, S_{12}, S_{11}$
t_1: $S'_{16}, S_8, S_{19}, S_{13}, S_3, S_5$
t_2: $S_{10}, S_{14}, S_{23}, S_6$
t_3: S_{25}, S_{15}, S_{17}
t_4: $S_{22}, S_{24}, S_{18}, S_{27}$
t_5: S_{26}, S_{28}
t_6: S_{29}

После каждого этапа параллельного выполнения t_i ($i = 0, 1, \ldots, 5$), кроме последнего, необходимо применение функций объединения результатов.

По результатам тестирования сложного запроса Q построена диаграмма (рис. 6), где по оси Oy показано время выполнения запроса в наносекундах.

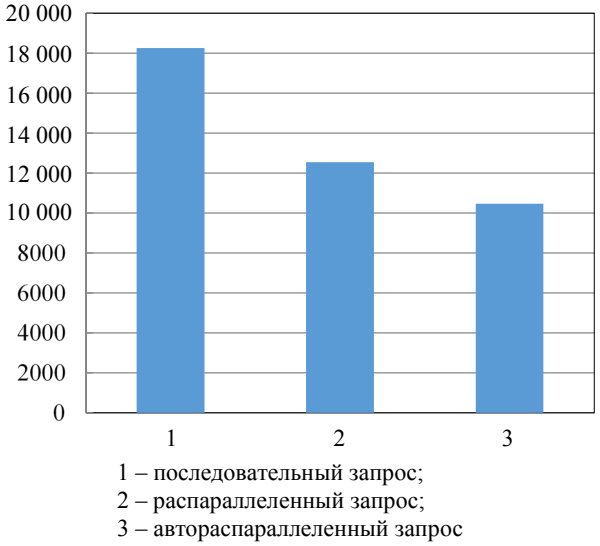


Рис. 6

Видно, что распараллеленный запрос оказался эффективнее последовательного. То, что запрос, распараллеленный с применением теории графов и методов параллельных вычислений, оказался более эффективным, – для данного эксперимента случайность. В теории параллельных вычислений есть другие методы, позволяющие оптимизировать параллельные алгоритмы по различным параметрам (времени, плотности вычислений, объему межпроцессорных передач, объему вычислительных ресурсов). И применение этих методов даст более эффективный результат по скорости выполнения запроса.

Результаты. Тестирование методов распараллеливания запросов к базам данных в результате большинства тестов показало положительный результат. Анализ полученных в результате тестирования показателей скорости выполнения запросов показал:

1. Необходимость применения для распараллеливания запросов к базам данных двух подходов, известных в теории параллельных вычислений как параллелизм по данным и параллелизм по задачам.

2. Возможность применения для распараллеливания запросов к базам данных методов из теории параллельных вычислений. Более того, большинство этих методов применять к запросам в базах данных намного проще, чем к произвольным алгоритмам в классическом программировании. Связано это в первую очередь с меньшим числом конструкций SQL по сравнению с классическими языками программирования; во вторую – с меньшим размером кода запроса; в третью – с меньшей сложностью структуры запроса; в четвертую – с более точной оценкой временной трудоемкости операций SQL за счет наличия механизма сбора статистики в большинстве современных СУБД.

3. Необходимость модификации (или адаптации) методов теории параллельных вычислений для непосредственного применения в запросах к базам данных. Если посмотреть на последний тест (рис. 6), то видно, что для полностью автоматического построения расписания и автоматического преобразования кода запроса под это расписание необходимо в граф добавить дополнительные вершины, которые будут соответствовать точкам объединения результатов после очередного выполнения серии параллельных подзапросов. А это в свою очередь добавит ребер и усложнит граф.

4. При применении первого подхода (параллелизм по данным) необходимо учитывать, что не все операции Select такой подход допускают. Так, очевидно, что если оператор Select имеет вид

Select * from table where *поле* > *значение*;

то применить первый подход возможно так:

Create view V1 as select * from table where *поле* > *значение* and *ключ* < = count/2;

Create view V2 as select * from table where *поле* > *значение* and *ключ* > count/2;

Тогда для объединения результата понадобится команда

Select * from V1 union select * from V2.

Возможно применение подхода к распараллеливанию по данным и при наличии в операторе Select функции агрегирования count, min, sum, max.

Для выбора значений и объединения результатов в этом случае понадобятся команды:

Create view V1 as select max (*поле*) as M from table where *поле* > *значение* and *ключ* < = count/2;

Create view V2 as select max (*поле*) as M from table where *поле* > *значение* and *ключ* > count/2;

Select max (M) from V1 union select M from V2.

Однако применение подхода к распараллеливанию по данным невозможно при наличии в конструкции Select, например, функции агрегирования avg.

Все эти ситуации должны анализироваться при распараллеливании запроса. В итоге как вариант в граф зависимостей подзапросов может быть добавлен еще один параметр или тип вершин, показывающий делимость или неделимость таблиц при выполнении операций, соответствующих вершинам.

Следует также учитывать, что при совмещении этих двух подходов граф запроса значительно вырастет по числу вершин и ребер. Однако это оправданно, так как дает исследователю механизм, позволяющий оценить существующие запросы и оптимизировать их в случае необходимости.

Исследование выполнено при финансовой поддержке РФФИ в рамках научного проекта № 18-57-34001.

СПИСОК ЛИТЕРАТУРЫ

1. Knowledge Base of Relational and NoSQL Database Management Systems. URL: https://db-engines.com/en/ranking (дата доступа 24.09.2018).

2. A NASA perspective on quantum computing: Opportunities and challenges / R. Biswas, Z. Jiang, K. Kechezhi, S. Knysh, S. Mandra, B. O'Gorman, A. Perdomo-Ortiz, A. Petukhov, J. Realpe-Gomez, E. Rieffel, D. Venturelli, F. Vasko, Z. Wang // Parallel Computing. 2017. Vol. 64. P. 81–98.

3. Hybrid storage architecture and efficient MapReduce processing for unstructured data / Weiming Lu, Yaoguang Wang, Jingyuan Juang, Jian Liu, Yapeng Shen, Baogang Wei // Parallel Computing. 2017. Vol. 69. P. 63–77.

4. Peiquan Jin, Puyuan Yang, Lihua Yue. Optimizing B+-tree for hybrid storage systems // Distributed and Parallel Databases. 2015. Vol. 33. P. 449–475.

5. Qiong Luo, Jens Teubner. Special issue on data management on modern hardware // Distributed and Parallel Databases. 2015. Vol. 33. P. 415–416.

6. Abdurrahman Yasar, Bugra Gedik, Hakan Ferhatosmanoglu. Distributed block formation and layout for disk-based management of large-scale graphs // Distributed and Parallel Databases. 2017. Vol. 35, № 1. P. 23–53.

7. Daichi Amagata, Takashiro Hara, Shojiro Nishio. Sliding window top-k dominating query processing over distributed data streams // Distributed and Parallel Databases. 2016. Vol. 34, № 4. P. 535–566.

8. Agustinus Borgy Waluyo, Bala Srinivasan, and David Taniar. Research in Mobile Database Query Optimization and Processing // Mobile Information Systems. 2005. Vol. 1, № 4. P. 225–252.

9. Spiliopoulou M., Hatzopoulos M. Ttanslation of SQL queries into a graph structure: query transformations and pre-optimization issues in a pipeline multiprocessor environment // Informafion Sysfems. 1992. Vol. 17, № 2. P. 161–170.

10. Yao S. B. Optimization of query evaluation algorithms // ACM Trans. Database Syst. 1979. № 4 (2). P. 133–155.

11. Mikkilineni K. P., Su S. Y. W. An evaluation of relational join algorithms in a pipelined query processing environment // IEEE Trans. Software Engng. 1988. № 14(6). P. 838–848.

12. Jarke M., Koch J. Query optimization in database systems // ACM Comput. Suru. 1984. № 16 (2). P. 111–152.

13. Smith J. M., Chang P. Y. T. Optimizing the performance of a relational algebra database interface // Commun ACM. 1975. № 18 (10). P. 568–579.

14. Sai Wu, Feng Li, Sharad Mehrotra, Beng Chin Ooi, Query Optimization for Massively Parallel Data Processing // Proc. of the 2011 SoCC conf., NY, USA, Oct. 2011. P. 12.

15. Shnaiderman L., Shmueli O. A Parallel Tree Pattern Query Processing Algorithm for Graph Databases using a GPGPU, Workshop Proceedings of the EDBT/ICDT // 2015 Joint Conf., on CEUR-WS.org. Brussels, March 27, 2015. URL: http://ceur-ws.org/Vol-1330/paper-24.pdf (дата обращения 24.09.2018).

16. Dex: Scalable high-performance graph database. http://www.sparsity-technologies.com/dex (дата обращения 24.09.2018).

17. Shichkina Y. A., Kupriyanov M. S. Applying the list method to the transformation of parallel algorithms into account temporal characteristics of operations // Proc. of the 19th Intern. Conf. on Soft Computing and Measurements. St. Petersburg: Elsevier, 2016. P. 292–295.

V. S. Shevsky, Yu. A. Shichkina
Saint Petersburg Electrotechnical University «LETI»

ANALYSIS OF PARALLEL QUERY EXECUTION METHODS FOR RELATIONAL DATABASES

In the modern world there is an increase in the scope of data used in various areas of the information society. Large amounts of data create a potential problem for their processing. The way to solve this problem is the use of algorithms to optimize the reading and writing of data implemented as software, as well as the introduction of devices that represent a platform for rapid access and transfer of data and that are implemented as hardware. The present article presents the results of a research carried out by the authors in this field. The authors propose an approach to the execution of a query, which involves changing the initial query and its execution using a parallel computation library. The presented results show that the authors' assumptions are correct and the approach can be applied when optimizing queries to relational databases.

Database, database managing system, multithread programming

УДК 630.945.31

С. В. Воробьев, О. П. Кормилицын
Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В. И. Ульянова (Ленина)

Современные технологии проектирования, производства и конструкторская подготовка студентов в техническом вузе

Рассматриваются вопросы конструкторской подготовки студентов в рамках дисциплин, преподаваемых кафедрой. Представлены задачи, которые необходимо решать при проектировании технических объектов. Достаточно подробно изложены существующие на кафедре формы и методы обучения студентов моделированию конструкций и анализу их напряженно-деформированного состояния при различных внешних воздействиях. Приводятся задачи, которые студенты решают в процессе курсового проектирования. Обращается внимание на теоретические и практические знания, которые должен иметь студент при решении задач курсового проектирования и выполнении самостоятельных работ. Предложены пути усиления подготовки молодых специалистов в области проектно-конструкторских работ. Учебные программы дисциплин строятся с учетом специфики направления подготовки молодых специалистов. В качестве примера приведены задания к самостоятельным работам студентов и примеры курсового проектирования.

Конструирование, механика, напряженно-деформированное состояние, кинематические параметры, прочность, компьютерное моделирование

Конструкторская подготовка студентов в инженерных высших учебных заведениях всегда имела большое значение, а сейчас, когда стремительно развиваются новые технологии, современные производства, открываются совершенно новые направления в науке и технике, эта задача становится особенно важной. Одним из основных этапов проектно-конструкторских работ в настоящее время ввиду широкого использования средств вычислительной техники, автоматизированных программных систем является процесс моделирования. Он состоит в выделении свойств объекта, определении особенностей взаимодействия с другими объектами, отражении особенностей функционирования при различных внешних воздействиях и логическом анализе собранной информации. Моделирование – метод исследования явлений, процессов и систем, основанный на построении и исследовании их моделей. Исследования в области технических наук включают