САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Математико-механический факультет

Кафедра системного программирования

Ерохин Георгий Алексеевич

Исследование масштабируемости многомерного индекса на основе R-дерева

Дипломная работа

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н., профессор Терехов А.Н.

Научный руководитель:
ассистент Чернышев Г.А.

Рецензент:
д. ф.-м. н., профессор Новиков Б.А.

Санкт-Петербург
2014

SAINT-PETERSBURG STATE UNIVERSITY
Mathematics & Mechanics Faculty

Software Engineering Chair

Georgii Erokhin

Investigation of R-tree based multithreaded multidimensional index scalability

Graduation Thesis

Admitted for defence.
Head of the chair:
professor Andrey Terekhov

Scientific advisor:
assistant professor George Chernishev

Reviewer:
professor Boris Novikov

Saint-Petersburg
2014

Оглавление

1. Введение

1.1. Многомерное индексирование

При поиске среди большого количества данных часто используются приемы, позволяющие избежать последовательного просмотра всего имеющегося объема информации. В книгах для этого присутствуют оглавление и алфавитный указатель, в справочниках и словарях — алфавитный порядок. В системах баз данных для этого используются индексы — избыточные структуры хранения, предназначенные для ускорения выборки данных по некоторому классу условий на значения атрибутов. Если поиск производится значению нескольких атрибутов, приходится иметь дело с многомерными индексами. Например, в географических информационных системах, ключ представляет собой координаты объекта в пространстве. Помимо этого, существует множество примеров применения многомерного индексирования: поиск по мультимедиа данным, которые описываются векторами признаков (например, изображение, представленное гистограммой цвета), системы автоматизированного проектирования и OLAP системы. Отдельно, среди многомерных индексов выделяют пространственные: в этом случае, компонентами ключа являются числовые данные. В этой работе рассматривается именно такой тип индексов.

В основном, при работе с подобными данными используются следующие виды запросов [6]:

- запросы на точное совпадение: найти конкретный объект;
- запросы на диапазон: найти все объекты, которые относятся к заданной области в многомерном пространстве;
- KNN — K ближайших соседей: найти K объектов, которые находятся ближе всего к заданному согласно некоторой метрике.

Распространенным подходом к созданию пространственных индексов является использование иерархических структур данных, например деревьев поиска. Такой подход широко используется, например, в геоинформационных системах и компьютерной графике [6].

В определенных условиях, а именно небольшая размерность и необходимость в

запросах на диапазон, R-дерево и структуры, основанные на нем часто используются для индексирования пространственных данных [6].

2. Постановка задачи

2.1. Мотивация работы

В последнее время, ввиду технологических ограничений, развитие процессоров пошло в сторону увеличения количества ядер. Для того, чтобы эффективно использовать этот ресурс, необходимо организовать параллельный доступ к данным. При этом возникают препятствия, из-за которых при увеличении количества вычислительных элементов, на некотором этапе производительность увеличивается незначительно, не увеличивается вообще или даже падает. Понять, в какой момент и из-за чего это происходит можно исследуя динамику производительности алгоритма при увеличении числа потоков, или, иными словами, оценив масштабируемость этого алгоритма.

Масштабируемость так же важна при использовании архитектуры клиент-сервер, когда к одному серверу обращаются много клиентов. Примером такой системы может служить сервис Google Maps, где множеству пользователей предоставляется одновременный доступ к многомерным данным.

На многоядерной машине, масштабируемый алгоритм может без существенного падения производительности обслуживать увеличивающееся количество клиентов, эффективно распределяя нагрузку по ядрам. Также, масштабируемость обеспечит увеличение производительности при переходе на новое оборудование с большим числом ядер.

Многие современные системы характеризуются большим объемом оперативной памяти ввиду её доступности. Например, в аппаратно-программном комплексе Oracle Exadata X2-8 объем ОЗУ может доходить до 2 ТБ на каждый процессор. В связи с этим, набирают популярность СУБД, работающие в памяти. Исследователи отмечают перспективность такой архитектуры [4], их мнение подтверждает возрастающее количество индустриальных СУБД, использующих этот подход (SAP HANA, VoltDB, MemSQL). Надежность (англ. durability) в этом случае может обеспечиваться ведением журнала на жестком диске и обращением к нему в случае сбоев. Так же, при оптимизации OLTP систем для работы в оперативной памяти, некоторые исследователи предлагают вовсе отказаться от использования журнала. При этом, от возможности восстановления системы после сбоев либо отказываются, либо либо используют специализированные алгоритмы, например, в случае распределенной системы восста-

новление может происходить за счет других узлов [4].

2.2. Постановка задачи

Для того чтобы определить, на что следует обращать внимание при разработке новых и оптимизации старых алгоритмов параллельного доступа для древовидных структур данных, необходимо знать, как на больших нагрузках и в условиях современного аппаратного обеспечения ведут себя существующие алгоритмы, как на производительность влияют такие факторы как блокировки или простои, вызванные обращением к памяти. Кроме того, данное поведение следует изучить в контексте нагрузок различных типов и их характеристик, таких как селективность. Целью данной работы является исследование масштабируемости алгоритмов поддержки транзакций и параллельного доступа к R-дереву, находящемуся полностью в оперативной памяти. Для достижения этой цели были поставлены следующие задачи:

- выбрать наиболее эффективные алгоритмы параллельного доступа к R-дереву;
- проанализировать производительность выбранных алгоритмов на различном количестве потоков, используя инструменты профилирования;
- на основе полученной информации сделать выводы о масштабируемости данных алгоритмов.

3. Обзор существующих подходов

Описанные тенденции в развитии аппаратного обеспечения сформировались достаточно давно, и существует множество работ, в которых предлагаются структуры данных и алгоритмы, предназначенные для работы в оперативной памяти, также как и алгоритмы, обеспечивающие эффективный параллельный доступ к данным. Во многих статьях встречаются такие исследования, но имеются существенные различия. Далее описаны некоторые из подобных работ.

В статье [7] рассматривается архитектура с неразделяемой памятью, при этом R-дерево хранится на диске. В работе [8] и множестве ей подобных также исследуется масштабируемость по количеству запросов, но рассматривается концептуально другой тип нагрузки, а именно непрерывные запросы в пространственно-временных базах данных. В этом случае задача имеет свою специфику и на первый план выходят проблемы несколько иного рода.

Также, стоит отметить статью [5]. В этой работе авторы сфокусировались на случае, когда данные хранятся в оперативной памяти. Специально разработанный для таких условий метод обеспечения целостности B-дерева был адаптирован под R-дерево, в работе присутствуют некоторые выводы о масштабируемости и причин такого поведения исследуемых алгоритмов. Однако, в той работе не берутся в расчет алгоритмы, обеспечивающие транзакционность.

4. Прототип многомерного индекса

На кафедре системного программирования математико-механического факультета СПбГУ разрабатывается прототип многомерного индекса[1]. Прототип реализован на C/C++ с использованием библиотеки pthreads, оперирует данными в оперативной памяти, поддерживает транзакции и параллельный доступ. На данный момент, другие существующие исследовательские платформы с похожей функциональностью не обеспечивают транзакционность и параллельный доступ, либо предназначены для исследования алгоритмов и структур, работающих с дисковой памятью, поэтому именно данный прототип был выбран в качестве тестовой платформы в этом исследовании.

4.1. Структура GiST

Новые типы индексов появляются достаточно часто, однако, несмотря на все преимущества, немногие из них используются на практике, так как встраивание в существующие системы слишком затратно. Одной из причин этого является то, что для применения в реальных условиях требуется широкая функциональность, например, поддержка параллельного доступа к данным. Для решения этой проблемы, было предложено обобщенное дерево поиска (Generalized Search Tree, GiST) [2]. Основная его идея следующая: у многих индексов есть общие свойства, так, например, во многих деревьях поиска, ссылки на сами данные располагаются только на последнем уровне, при этом все листья находятся на одинаковом расстоянии от корня. Собрав подобные характерные особенности, авторы [3] предложили структуру-шаблон, где уже реализована наиболее типичная для многих деревьев поиска функциональность. Если в системе реализован GiST, для добавления индекса на его основе нужно только написать часть, уникальную для данного конкретного дерева. Позже, были предложены алгоритмы обеспечения корректности параллельного доступа и поддержки транзакций для GiST. Исследование подобных обобщенных структур позволяет применить результаты сразу ко всему спектру деревьев, которые можно реализовать с их помощью. В описанном выше прототипе реализовано обобщенное дерево поиска, на его основе реализованы несколько вариантов R-дерева и алгоритмов обеспечения параллельного доступа.

[1]работа частично поддержана грантом РФФИ №12-07-31050

Перед рассмотрением изложенных ниже алгоритмов, следует подробнее представить структуру GiST.

GiST — сбалансированное дерево (иными словами, все его листья находятся на одном уровне). Листовая вершина содержит массив пар вида (p, ptr), где ptr - указатель на объект базы данных, а p - предикат, соответствующий объекту, на который указывает ptr. Что конкретно представляет собой предикат зависит от конкретного дерева, реализованного с использованием GiST, в случае R-дерева, предикатом является многомерный прямоугольный параллелепипед, а в случае B-дерева — числовой отрезок. Внутренние вершины содержат массив пар (p, ptr), где p — также предикат, а ptr — указатель на другую вершину. Далее в этой работе, такие пары именуются записями.

GiST обладает следующими свойствами:

- Любая некорневая вершина содержит от kM до M записей, где M и k — константы, $\frac{2}{M} \leq k \leq \frac{1}{2}$. Корень может иметь от 2 до M детей, если не является листом;

- Для любой записи (p, ptr), находящейся в листовой вершине, p является верным для объекта, на который указывает ptr;

- Для любой записи (p, ptr), находящейся во внутренней вершине, p является верным для всех объектов, достижимых из узла, соответствующего ptr. Следует отметить, что для произвольной записи $(p^{'}, ptr^{'})$, достижимой из ptr это не означает импликацию $p \Rightarrow p$', а лишь означает, что для всех объектов, достижимых из $ptr^{'}$, выполняется $p^{'}$ и p;

- У корня всегда есть как минимум две дочерние вершины за исключением случая, когда он является листом (в противном случае он будет единственной вершиной в дереве и может иметь ноль или одну дочернюю);

- Все листья находятся на одном уровне.

4.1.1. Предикат

Предикат соответствует поисковому ключу и представляют собой класс, реализующий набор методов. GiST параметризуется этим классом, то есть для добавления в

систему, где уже есть GiST, нового дерева, достаточно реализовать лишь предикат и следующий набор методов:

- `Consistent(`E, q`)` по данной записи $E = (p, ptr)$ и предикату q, возвращает "ложь", если предикат $p \wedge q$ принимает значение "ложь".

- `Union(`P`)` по заданному списку предикатов $P = (p_1, ptr_1), ..., (p_n, ptr_n)$ возвращает предикат p, такой, что он выполняется для объединения $p_1, ..., p_n$

- `Penalty(`E_1, E_2`)` по двум записям $E_1 = (p_1, ptr_1), E_2 = (p_2, ptr_2)$ возвращает штраф за вставку E_2 в поддерево, корнем которого является p_1. В большинстве случаев, его можно понимать как разницу в размере p_1 и `Union((`E_1, E_2`)`, или насколько хуже станет дерево после вставки $E2$ в ptr_1. Под словом "хуже" понимается увеличение времени поиска.

- `PickSplit(`P`)` разбивает множество P из $M + 1$ записи на два, каждое не менее kM записей размером. Интуитивно, при этом опять же должна минимизироваться некоторая метрика, характеризующая, насколько в результате пострадает время поиска в дереве.

4.1.2. Поиск

Под поиском имеется ввиду извлечение из дерева всех объектов, для которых заданный предикат q истинен. Поиск происходит следующим образом: начиная с корня, производится обход дерева, при этом отбрасываются вершины, ptr для которых ложно `Consistent(`E, q`)` где E — запись, содержащая ptr.

4.1.3. Вставка

Вставка новой записи E состоит из трех частей:

1. Нахождение листа L, такого, что вставка в него E повлечет за собой как можно меньший штраф (определяемый как `Penalty(`E', E`)`, где E' - запись, содержащая ссылку на L);

2. Вставка E в L, при переполнении (если размер L станет больше M) происходит расщепление вершины, как это происходит описано далее;

3. Изменение предикатов в дереве таким образом, чтобы сохранились свойства GiST, описанные выше.

4.1.4. Расщепление

Если в вершине N уже находится M записей, при этом необходимо вставить туда ещё одну, происходит расщепление: создается новая вершина, в которую переносится часть записей из переполненной, разбиение выбирается при помощи метода `PickSplit`. Запись, соответствующая новой вершине, вставляется в родителя N, который так же при необходимости расщепляется. Этот процесс может дойти до корня, и в этом случае не будет родительской вершины, куда бы мы вставили ссылку на новую вершину тогда создается новый корень с двумя записями: со ссылкой на старый корень и на новую вершину. Для того, чтобы обеспечить возможность доступа к дереву сразу нескольким пользователям, необходимо позаботиться о двух моментах:

- Обеспечение целостности дерева, структура самого индекса не должна пострадать от того, что его изменяют сразу несколько потоков;
- Обеспечения требуемого уровня изоляции транзакций. Если пользователю необходимо выполнить набор операций, это должно выглядеть так, будто кроме него базой данных никто не пользуется, ничего не добавляет, изменяет или читает.

5. Алгоритмы, обеспечивающие целостность дерева

5.1. Обобщение алгоритма R-link

В статье, где впервые был предложен способ обеспечения параллельного доступа и изоляции транзакций специально для GiST [3], описывается метод, представляющий модификацию алгоритма B-link дерева, превосходство которого над своими аналогами было подтверждено в нескольких работах.

Преимущество этого алгоритма состоит в том, что в большинстве ситуаций каждый поток держит замок максимум на одну вершину. Достигается это за счет использования порядкового номера узла (англ. node sequence number, NSN) и объединения вершин, полученных в результате расщепления, в связный список. При параллельном доступе к GiST, проблема возникает при расщеплении вершины. Допустим, некоторый поток совершает обход дерева, это значит, что в некоторый момент времени, он сохраняет ссылку на определенную вершину, затем, через некоторое время возвращается к ней и обходит записи, которые в ней находятся. Тогда в случае расщепления этой вершины, при обходе будет затронута только часть записей, а новая вершина, образованная в результате того расщепления, останется незамеченной. Вкратце, суть этого алгоритма можно описать следующим образом. Для решения этой проблемы, предлагается присваивать узлам порядковый номер следующим образом: в дереве глобальный счетчик, при расщеплении, новому узлу присваивается значение NSN старого узла, а старому присваивается значение этого счетчика, после чего счетчик увеличивается, и в старый узел добавляется ссылка на новый, называемая ссылкой направо (англ. rightlink). Теперь, при обходе дерева, достаточно сохранить вместе с узлом значение его NSN, а при обходе его записей, сравнить текущее значение NSN с сохраненным, и если они не совпадают - добавить в обход вершину, на которую указывает ссылка направо.

Более формальное описание алгоритма выглядит следующим образом: Вместо замков, используются защелки (latches), отличие которых от замков (locks) заключается в двух аспектах. Во-первых, они адресуются физически в отличие от замков, доступ к которым зачастую организуется с использование хэш-таблиц. Во-вторых, защелки берутся на короткий период времени и не требуют проверок на образование взаимных блокировок (тупиков). Защелки привязаны к узлам (в реализации, исполь-

зованной в этой работе - являются их полями) и могут иметь два режима блокировки: блокировка на запись (далее она будет называться $XLatch$, от англ. eXclusive Latch), в таком режиме защелкой может обладать только один поток и на чтение (далее - $SLatch$, от англ. Shared Latch), в таком режиме защелкой могут обладать несколько потоков одновременно.

5.1.1. Поиск

Дерево обходится в глубину с использованием стека. В стека хранятся пары вида (ссылка на узел, старый NSN этого узла). На каждом шаге обхода, происходит следующее:

1. Из стека извлекается очередная вершина и старое значение её NSN;
2. Берется защелка в режиме $SLatch$;
3. Если NSN из стека меньше текущего NSN вершины, в стек добавляется правая ссылка этой вершины и старое значение NSN;
4. Если вершина является листом, к результирующему набору добавляются все записи, которые в ней содержатся и соответствуют предикату, по которому производится поиск;
5. Если вершина внутренняя, в стек добавляются все записи, которые в ней содержатся и соответствуют предикату, по которому производится поиск.

5.1.2. Вставка

1. Этот шаг очень похож на процедуру поиска с тем лишь различием, что обход производится по единственному пути. Лист, в который следует вставить новую запись, находится с помощью обхода дерева, при этом выбирается путь с наименьшим штрафом. На этот лист берется защелка в режиме $XLatch$. При этом, ссылки на все вершины, из которых состоит выбранный путь (кроме самого листа) и текущие значения их NSN сохраняются в стеке в порядке обхода;
2. Если выбранный лист переполнен, вызывается процедура расщепления.

3. Освобождаются все защелки, взятые на предыдущих шагах (за исключением листа, в который происходит вставка);

4. Вызывается процедура обновления предикатов в узлах, располагающихся на пути от корня до выбранного листа.

5.1.3. Расщепление

В эту процедуру передается два параметра: переполненная вершина, в которую необходимо вставить новую запись и стек, содержащий всех его предков в порядке от родителя этой вершины к корню (родителем эта вершина должна быть только на момент добавления в стек во время первого этапа процедуры вставки, после чего может быть расщеплена другим потоком, то же самое относится и к остальным предкам в стеке).

1. Взять защелку на чтение на родителя переполненной вершины;

2. Если NSN родителя отличается от сохраненного в стеке, пройтись по ссылкам направо и найти вершину, которая в данный момент является настоящим родителем. При переходе по ссылке, отпускается защелка текущей вершины и берется защелка вершины, на которую указывает ссылка направо. В результате, блокированной оказывается только настоящий родитель;

3. Создать новую вершину и взять на неё защелку в режиме $XLatch$;

4. Вызвать `PickSplit`, чтобы распределить записи из переполненной вершины между ней и новой вершиной;

5. Присвоить новой вершине NSN старой, а значение NSN старой вершины изменить на значение глобального счетчика, инкрементировать этот счетчик;

6. Если в родительской вершине недостаточно места, использовать рекурсию и вызвать этот же метод для родительской вершины;

7. Вставить новую вершины в родительскую и обновить предикат для текущей вершины (в ней стало меньше записей, предикат должен измениться соответственно).

Для краткости изложения, опущены подробности, касающиеся процедуры обновления предикатов после вставки вершины и расщепления корня.

5.2. OLFIT

Алгоритм обхода дерева, при котором блокировки не берутся. Таким образом, алгоритм только читает данные, не изменяя их. Это приводит к более эффективному использованию кэш-памяти. После чтения узла, проверяется, захвачена ли его защелка, либо был ли он изменен другим потоком в процессе чтения. В этих двух случаях, содержимое узла считывается снова.

6. Алгоритмы, обеспечивающие изолированность транзакций

Уровень изоляции, при котором пользователь видит данные так, будто все транзакции выполняются строго последовательно, называется полной сериализуемостью. Его обеспечение весьма затратно, приводит к ухудшению производительности и не всегда требуется, поэтому зачастую рассматриваются более мягкие условия, точнее, выделяют четыре уровня изоляции, которые характеризуются набором ситуаций, которые они позволяют избежать:

- Потерянное обновление: две транзакции читают, изменяют и записывают один и тот же участок памяти, при этом данные, записанные одной из транзакций теряются.
- Грязное чтение: возможно считывание данных, измененных другой транзакцией, которая впоследствии была отменена.
- Неповторяемое чтение: при повторении поискового запроса, считанные данные могут отличаться от результатов первого запроса, так как были изменены другой транзакцией.
- Фантомное чтение: при повторении поискового запроса, в результирующем наборе могут появиться новые записи.

Соответственно, различают четыре уровня изоляции:

- Read Uncommitted гарантирует отсутствие грязного обновления;
- Read Committed предотвращает потерянное обновление и грязное чтение;
- Repeatable Read, когда не возникает ни потерянное обновление, ни грязное чтение, ни неповторяемое чтение;
- Serializable, при котором отсутствуют все вышеперечисленные аномалии.

В данной работе, объектом исследования был выбран уровень Read Committed, так как он достаточно распространен (например, он стоит по умолчанию в СУБД

PostgreSQL) и в то же время довольно слабо изучен, в частности, известно не так много алгоритмов, обеспечивающих именно его.

Классическим алгоритмом, используемым для обеспечения этого уровня изоляции, является следующий: при вставке, удалении или изменении записи, транзакция берет на нее блокировку в неразделяемом режиме до момента фиксации, таким образом реализуется алгоритм строгого двухфазового блокирования (S2PL). При поиске же блокировка берется в разделяемом режиме лишь до его окончания.

6.1. Модифицированный алгоритм для Read Commited

На конференции СПИСОК-2012 был представлен алгоритм, обеспечивающий уровень изоляции Read Commited [9]. При вставке новых записей, удалении или изменении существующих, используется протокол S2PL, однако при поиске ни замки, ни защелки не берутся, из-за чего повышается уровень параллелизма, то есть количество операций, выполняемый одновременно. Это достигается за счет добавления к записям, в которых хранятся ссылки на данные, двух полей. Одно из них — поле, описывающее текущее состояние, которое принимает одно из четырёх значений:

- $Valid$ означает, что в данный момент, запись не изменяется никакой из транзакций;
- $Deleted$, это значение необходимо для реализации логического удаления, то есть когда приходит запрос на удаление элемента, он фактически не удаляется из структуры, а помечается как удаленный и в дальнейшем при обходе дерева игнорируется;
- $ProcessDelete$ означает, что некоторая незафиксированная транзакция в данный момент удаляет эту запись, то есть всем другим транзакциям, она должна быть видна;
- $ProcessInsert$ означает, что незафиксированная транзакция в данный момент вставляет эту запись и никакая другая транзакция не должна её видеть;
- $ProcessUpdate$ означает, что запись изменяется (под изменением подразумевается изменение данных, соответствующих заданному ключу) и другим транзакциям должно быть видно старое значение. Если выставлено это значение, то

наряду с новыми данными, в записи хранятся старые, то есть данные, какими до обновления.

Другое поле — в случаях $ProcessDelete$, $ProcessInsert$ или $ProcessUpdate$, это идентификатор незафиксированной транзакции, которая в данный момент вставляет, удаляет или модифицирует запись. В остальных случаях — любое значение. При поиске, данные, на которые ссылается запись считываются либо пропускается в зависимости от значений этих двух полей:

- $Valid$ — считываются;
- $Deleted$ — пропускаются;
- $ProcessDelete$ — пропускаются, если идентификатор транзакции совпадает с идентификатором читающей транзакции, иначе считываются;
- $ProcessInsert$ — считываются, если идентификатор транзакции совпадает с идентификатором читающей транзакции, иначе пропускаются;
- $ProcessUpdate$ — считываются новые данные, если идентификатор читающей транзакции такой же, как и тот, что сохранен в записи, иначе считываются старые данные.

Таким образом, степень параллелизма возрастает в обмен на небольшое увеличение объема используемой памяти.

6.2. Диспетчер замков

Допустим, два потока одновременно выполняют транзакции. Пусть первый поток захватил замок $l1$, второй — замок $l2$. Тогда если первый поток попытается завладеть $l2$, в то время как второй — замком $l1$, возникнет ситуация, в которой оба потока будут бесконечно долго ждать своей очереди. Подобное состояние называется взаимной блокировкой, или тупиком. Для предотвращения таких ситуаций, используются диспетчер блокировок (англ. lock manager). Он работает следующим образом: строится граф ожидания транзакций. Это направленный граф, в котором вершины соответствуют активным транзакциям, и ребро (A, B) означает, что транзакция A захватила

некоторый ресурс, а транзакция B ждет, пока этот ресурс освободится. При взаимной блокировке, в графе ожидания образуется контур, поэтому для предотвращения тупиков достаточно периодически совершать обход графа, откатывая транзакции, которым соответствуют вершины, находящиеся в контурах. При этом, достаточно откатить только одну транзакцию, участвующую во взаимной блокировке.

Предотвращение взаимных блокировок было реализовано путем поиска контуров при взятии каждого замка. При этом замки хранятся непосредственно в записях листовых вершин, что позволяет отказаться от хэш-таблицы, разделяемой всеми потоками, с помощью которой адресовались бы замки и которая могла бы быть бутылочным горлышком, ограничивающим производительность при увеличении количества потоков.

7. Анализ производительности

7.1. Масштабируемость

Масштабируемость (англ. Scalability) — это способность системы к увеличению производительности при добавлении новых ресурсов или способность эффективно перераспределять имеющиеся ресурсы при увеличении нагрузки. В этой работе на многоядерной системе с фиксированной конфигурацией процессора изменяется количество клиентов, то есть объем нагрузки, и масштабируемость определяется тем, насколько эффективно ядра процессора используются при обработке запросов от увеличивающегося числа клиентов. Такой подход применяется, например, при использовании инструментов нагрузочного тестирования Apache JMeter и HP LoadRunner.

Увеличение производительности характеризуется следующими метриками:

- Ускорение (англ. speed up) показывает, как производительность увеличивается при добавлении в систему новых ресурсов, то есть как при фиксированной нагрузке и наращивании мощностей системы увеличивается степень параллелизма. Определяется следующей формулой:

$$\text{Ускорение} = \frac{\text{время вычислений на однопроцессорной системе}}{\text{время вычислений на многопроцессорной системе}}$$

  Эта метрика используется при вычислении производительности запросов на чтение. В лучшем случае, ускорение прямо пропорционально количеству задействованных процессорных элементов.

- Масштабирование (англ. scale up) описывает способность системы выполнять более объемные задачи за то же время при добавлении в систему новых элементов (или увеличении степени параллелизма). Это понятие не следует путать с масштабируемостью. Тогда как масштабирование является метрикой, масштабируемость — это свойство системы. Масштабирование определяется формулой: $\frac{time1}{time2}$, где $time1$ — время, вычисления задачи на однопроцессорной системе, $time2$ — время вычисления большей задачи на многопроцессорной системе. Говорят, что масштабирование линейно, если производительность остается на том же уровне, когда и нагрузка, и вычислительные мощности наращиваются пропорционально, в этом случае, её значение равно единице. В зависимости от того,

как определяется размер нагрузки, различают два типа масштабирования:

- по количеству транзакций, то есть увеличивается количество транзакций, поступающих в систему в единицу времени. Вместе с этим, может увеличиваться и размер самой базы данных. Такую метрику целесообразно использовать для измерения производительности на нагрузках, включающих запросы, изменяющие данные. Далее в этой работе используется именно эта метрика;
- по размеру базы данных, используется в OLAP системах.

7.2. Универсальный закон масштабируемости

Универсальный закон масштабируемости (англ. universal scalability law, USL) [1] позволяет приближенно описать производительность системы в зависимости от количества задействованных процессорных элементов следующим образом:

$$C(N) = \frac{N}{1 + A * (N - 1) + B * N * (N - 1)}$$

Где A и B - параметры, $0 \leq A, B \leq 1$. A описывает накладные расходы вследствие конкуренции из-за разделяемых ресурсов, таких как блокировки или аппаратные ресурсы. Параметр B показывает, как на производительность влияет попарное взаимодействие аппаратных узлов. В частности, этот показатель описывает ухудшение производительности вследствие когерентности кэша [1]. $B > 0$ означает, что начиная с некоторого значения, добавление в систему новых ресурсов приведет к падению производительности, и чем больше B, тем сильнее деградация. В случае линейной масштабируемости, $A = B = 0$.

7.3. Использованная нагрузка

Были рассмотрены два типа нагрузки:

- С преобладающим количеством поисковых запросов: 85% поисковых запросов, 5% запросов на удаление, 5% запросов на обновление, 5% запросов на вставку
- С преобладающим количеством запросов, манипулирующих данными: 20% поисковых запросов, 20% запросов на удаление, 40% запросов на обновление, 20%

запросов на вставку

7.4. Профилирование

Для профилирования использовалась библиотека PAPI (Performance Application Programming Interface). Она позволяет во время исполнения программы считывать значения аппаратных счетчиков производительности. Счетчики представляют собой специальные регистры, встроенные в процессор и позволяющие получить информацию о заданном наборе событий, таких как выполненные инструкции или количество простоев, связанных с кэш-промахами. Эксперименты проводились на следующей конфигурации:

- Аппаратное обеспечение: Intel Core i7-2670QM, 2.20 GHz, Hyper-Threading, L1 кэш 32 КБ, L2 кэш 256 КБ, L3 кэш 6 МБ, 6 ГБ ОЗУ;
- Программное обеспечение: x86-64 GNU/Linux, ядро 3.11.0-12-generic, gcc 4.8.1.

На Рис. 1 и Рис. 2 показана производительность на нагрузке с большой долей поисковых запросов и на нагрузке с большой долей запросов, манипулирующих данными. Проверить накладные расходы, связанные с взаимодействием потоков можно используя инструменты профилирования. С помощью библиотеки PAPI было замерено время, потраченное на блокировки и количество простоев, связанных с кэш-промахами в пересчете на один поток. На Рис. 3 отображено, как в зависимости от количества потоков меняется доля времени, затраченная на блокировки (простои во время ожидания и накладные расходы, связанные с взятием блокировки) Рис. 4 показывает, какое в среднем на один поток количество холостых циклов процессора пришлось на кэш-промахи.

Селективность поискового запроса описывает, как много данных этому запросу удовлетворяют. Коэффициент селективности заданного запроса - это количество удовлетворяющих запросу объектов, деленное на размер всего множества, в котором происходит поиск. Возникла следующая гипотеза: селективность поисковых запросов влияет лишь на количество разделяемых потоками ресурсов, при этом накладные расходы вследствие взаимодействия аппаратных узлов никак от неё не зависят. Иными словами, параметр A универсального закона масштабируемости зависит от значения

селективности поисковых запросов следующим образом: чем больше селективность, тем меньше A. В то же время, B не изменяется. Для проверки этого предположения, была построена модель согласно универсальному закону масштабируемости для двух разных значений селективности, 0.1 и 0.0001 и двух нагрузок, определенных выше. В результате, были получены интервальные оценки для A и B:

- Для нагрузки с большой долей поисковых запросов, селективностью 0.1: $A \in (0.0960.117)$, $B \in (0.00680.0087)$;
- Для нагрузки с большой долей поисковых запросов, селективностью 0.0001: $A \in (0.0630.093)$, $B \in (0.00710.0097)$;
- Для нагрузки с большой долей обновлений, селективностью 0.1: $A \in (0.0970.124)$, $B \in (0.00730.0097)$;
- Для нагрузки с большой долей обновлений, селективностью 0.0001: $A \in (0.1070.122)$, $B \in (0.0100.011)$;

Полученные оценки позволяют принять гипотезу в случае большой доли поисковых запросов. Однако, во втором случае были получены неожиданные результаты, этот сценарий требует дальнейших исследований.

7.5. Выводы

- Ухудшение производительности, связанное с кэш-промахами и замками в пересчете на один поток не увеличивается.
- В случае смешанной нагрузки и большого количества поисковых запросов, селективность влияет лишь на количество разделяемых потоками ресурсов, при этом накладные расходы вследствие взаимодействия аппаратных узлов никак от неё не зависят.

8. Заключение

В ходе данной дипломной работы было проведено исследование масштабируемости алгоритмов параллельного доступа к R-дереву.

- Были выбраны наиболее эффективные алгоритмы:
  - Обеспечивающий корректность параллельного доступа к R-дереву;
  - Обеспечивающий уровень изоляции Read Committed.
- Проанализирована производительность выбранных алгоритмов;
- На основе полученной информации сделаны выводы о масштабируемости.

Список литературы

[1] Gunther Neil J., Subramanyam Shanti, Parvu Stefan. A Methodology for Optimizing Multithreaded System Scalability on Multi-cores // CoRR. — 2011. — Vol. abs/1105.4301.

[2] Hellerstein Joseph M., Naughton Jeffrey F., Pfeffer Avi. Generalized Search Trees for Database Systems // Proceedings of the 21th International Conference on Very Large Data Bases. — VLDB '95. — San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1995. — P. 562–573. — URL: http://dl.acm.org/citation.cfm?id=645921.673145.

[3] Kornacker Marcel, Mohan C., Hellerstein Joseph M. Concurrency and Recovery in Generalized Search Trees // SIGMOD Rec. — 1997. — jun. — Vol. 26, no. 2. — P. 62–72. — URL: http://doi.acm.org/10.1145/253262.253272.

[4] OLTP Through the Looking Glass, and What We Found There / Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, Michael Stonebraker // Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data. — SIGMOD '08. — New York, NY, USA : ACM, 2008. — P. 981–992. — URL: http://doi.acm.org/10.1145/1376616.1376713.

[5] Performance Evaluation of Main-Memory R-tree Variants / Sangyong Hwang, Keunjoo Kwon, SangK. Cha, ByungS. Lee // Advances in Spatial and Temporal Databases / Ed. by Thanasis Hadzilacos, Yannis Manolopoulos, John Roddick, Yannis Theodoridis. — Springer Berlin Heidelberg, 2003. — Vol. 2750 of Lecture Notes in Computer Science. — P. 10–27. — URL: http://dx.doi.org/10.1007/978-3-540-45072-6_2.

[6] R-Trees: Theory and Applications (Advanced Information and Knowledge Processing) / Yannis Manolopoulos, Alexandros Nanopoulos, Apostolos N. Papadopoulos, Y. Theodoridis. — Secaucus, NJ, USA : Springer-Verlag New York, Inc., 2005. — ISBN: 1852339772.

[7] Tam Edward Nai-Biu. R-tree Indexing by Multiple Processors. — URL: citeseer.ist.psu.edu/211031.html.

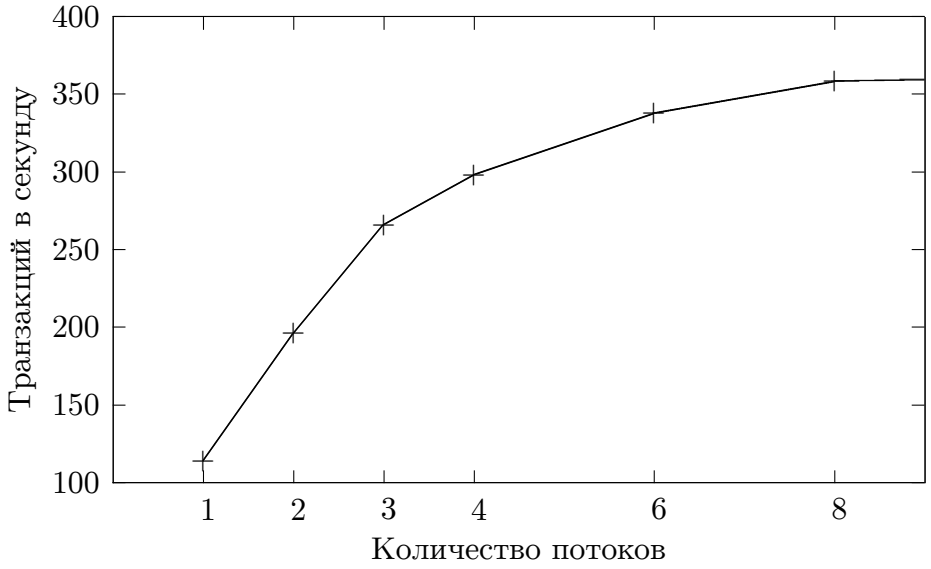

Рис. 1: Нагрузка с большой долей поисковых запросов

[8] Xiong Xiaopeng. Scalability in Spatio-temporal Data Management Systems : Ph. D. thesis / Xiaopeng Xiong. — West Lafayette, IN, USA : Purdue University, 2007. — AAI3287281.

[9] Федотовский П.В. Чернышев Г.А. Смирнов К.К. Реализация уровня изоляции Read Committed для древовидных структур данных // Конференция СПИСОК. — 2012.

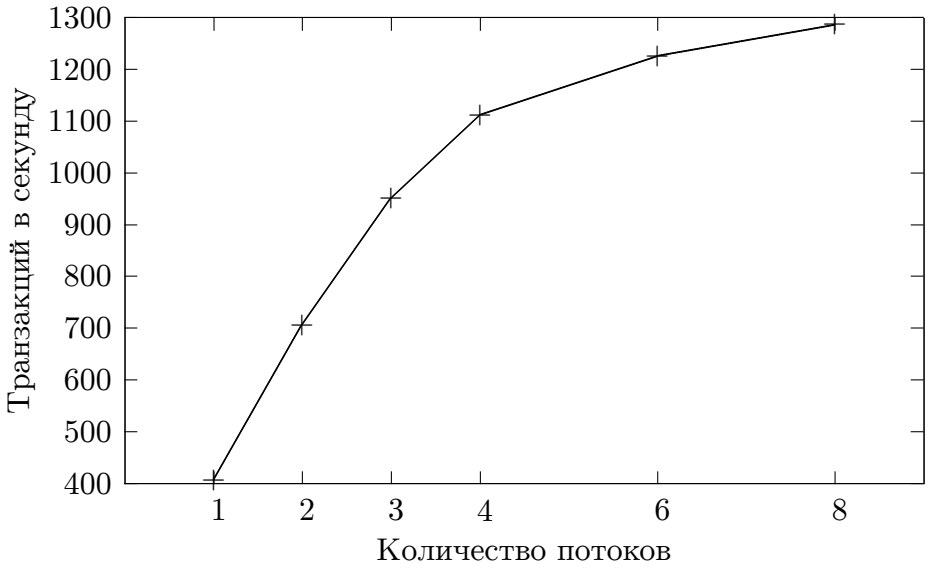

Рис. 2: Нагрузка с большой долей запросов, манипулирующих данными

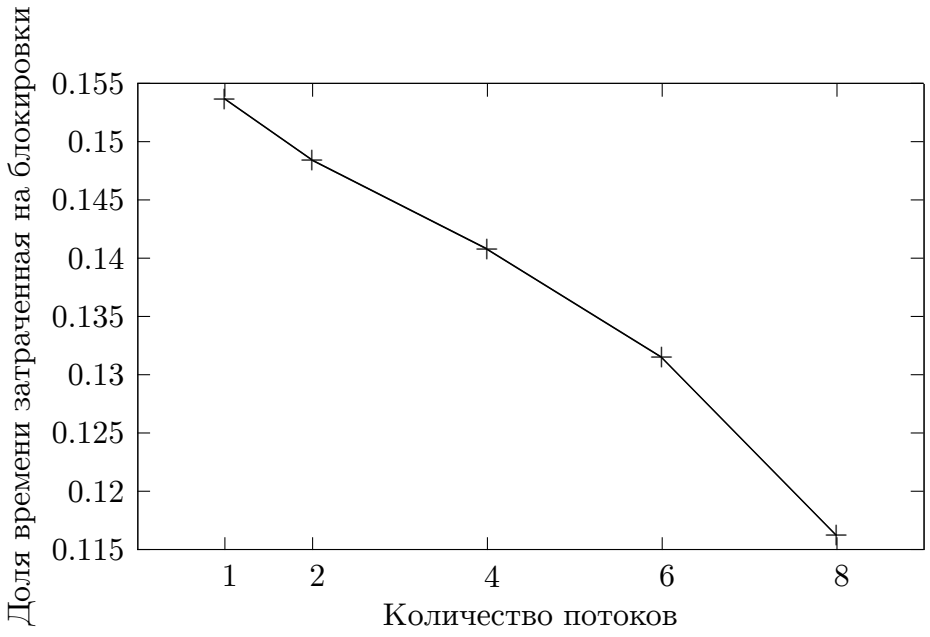

Рис. 3: доля времени, потраченная на блокировки в зависимости от количества потоков

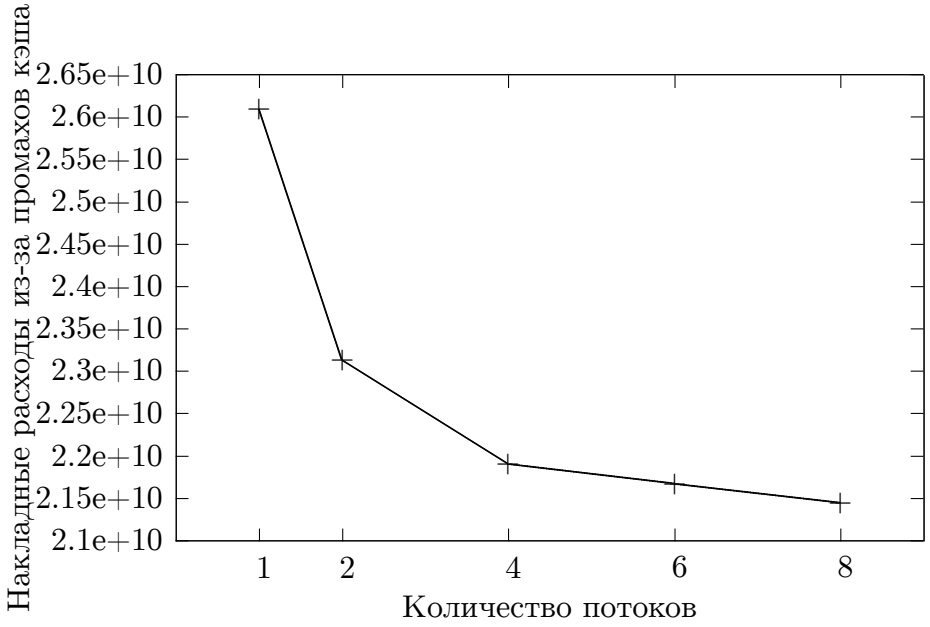

Рис. 4: Время, потраченное на кэш-промахи в зависимости от количества потоков