Хеширование движущихся объектов

Жексуан Сонг
Факультет информатики
Университет Мэриленда
Колледж-Парк, Мэриленд 20742
zsong@cs.umd.edu

Ник Россопулос
Факультет информатики и Институт
передовых компьютерных исследований
Университета Мэриленда
Колледж-Парк, Мэриленд 20742
nick@cs.umd.edu

19 мая 2000 года

Аннотация

В реальной жизни объекты существуют как в пространстве, так и во времени. Объекты, меняющие свое положение в течение продолжительного времени, называются движущимися. С развитием беспроводной связи и технологий позиционирования становится необходимым хранить и индексировать такие объекты в базе данных. Ввиду сложности проблемы многие сугубо пространственные структуры не способны индексировать большой объём движущихся объектов в базе данных.

В этой статье мы предлагаем абсолютно новую идею, основанную на методе хеширования. Так как стало возможным заново индексировать все объекты каждый период времени, мы размещаем их в «корзинах». Если объект перемещается в пределах корзины, база данных не меняется. Использование этой техники значительно снижает количество обновлений базы данных – процедура индексирования становится приемлемой для выполнения. Также мы расширяем структуру предыдущей системы, представляя фильтрующий слой между сборщиком информации о местоположении и базой данных. Кроме того, представлены четыре разных метода, основанных на новой системе. С целью оценить различные аспекты наших индексных техник были проведены тесты производительности, и их итоги приведены в данной статье.

1. Введение

Традиционно, системы управления базой данных применяют «статичную» модель, предполагающую, что данные, размещённые в базе, остаются неизменными, пока не будут изменены явно через операцию обновления. Такая модель приемлема, если свойства объектов меняются дискретно или не меняются вовсе. Однако, в реальной жизни множество объектов постоянно меняют свойства. Пусть некоторое приложение предназначено для поддержки базы данных в системе контроля за воздушным движением. В этом случае в роли движущихся объектов выступают самолёты. Их положение постоянно меняется. Одно из возможных решений для «статичной» модели мы назвали «наивным»: обновлять информацию о положении объектов через определённый промежуток времени. Ввиду больших затрат на обновление это решение представляется неэффективным.

С развитием систем позиционирования (таких как GPS), технологий беспроводной связи и электроники, стало технически возможно и необходимо отслеживать и записывать

местоположение большого количества движущихся объектов. В соответствии с [SJL+99], рынок мобильных телефонов ожидает более чем 500 миллионов пользователей к 2002 году и миллиард к 2004, а сами мобильные телефоны превратятся в беспроводные устройства для доступа к интернету. Отслеживание положения таких устройств может существенно улучшить качество услуг связи. Появление более сложных систем баз данных становится крайне необходимо.

Новая база данных, которая работает с геометрическими изменениями во времени, называется пространственно-временной базой данных. Проблемы этой новой сферы привлекают внимание как академического, так и производственного сообществ. В [WCD+98] представлена модель Moving Objects Spatio-Temoporal (MOST) и язык (FTL) для запроса текущего и будущего положения движущихся объектов; в [TJ98] предлагается концептуальная компонентная модель для разработки пространственно-временных приложений; М. Насименто в [NST99] представляет алгоритм GSTD ("Generate Spatio-Temporal Data" («Генерирование пространственно-временных данных»)), который генерирует множества движущихся точек или прямоугольников, подчиняющихся широкому кругу распределений. Искусственно созданные алгоритмом GSTD данные используются для оценки различных методов индексирования. Система Arc View GIS [ArcV98] уже поддерживает отслеживание мобильных объектов и обработку связанных с ними запросов.

Эта статья сконцентрирована на методах индексирования большого числа перемещающихся объектов без необходимости частого обновления базы данных. Главным образом будет обсуждаться, как быстро отвечать на различные запросы о текущем положении объекта, так как такие запросы представляют собой основную операцию для других запросов, таких как запросы ближайших соседей [CG99, SK98]. Так как ответ основан на текущей информации, расположенной в базе, нужно, чтобы эта информация была как можно точнее. Основная задача - избежать недопустимо больших накладных расходов на обновление.

«Наивное» решение отказывает с ростом количества движущихся объектов. Допустим, система управления базой данных может справиться с N_t операциями в секунду (в большинстве случаев N_t менее 1000). Число объектов – N_0. Через каждый промежуток времени информация о положении требует обновления. Таким образом, требуется N_0 операций обновления на каждый период. Это займёт по меньшей мере N_0/N_t секунд. Если N_0 невелико, например, несколько тысяч в случае с управлением воздушным движением, «наивное» решение довольно не плохо. Однако, если N_0 велико, к примеру, несколько миллионов при отслеживании движения автомобилей или даже больше, в случае с мобильной связью, каждое обновление требует десятков минут и более. Это значит, что позиционная информация об объекте, размещённая в базе данных, может устареть на десять минут и более! Результат запроса таких данных представляется неприемлемым.

Альтернативный подход [KGT99, SJL+99] – это представление позиции перемещающихся объектов в виде функции от времени $f(t)$ и обновление базы данных только тогда, когда параметры этой функции изменились. В большинстве случаев используются линейные функции ввиду своей простоты. В любое данное время t_0 можно найти положение каждого объекта путём вычисления функции $f(t_0)$. Такой подход также может прогнозировать будущее положение объектов. Он может хорошо сработать на каких-либо научных базах данных, где путь каждого объекта обычно известен заранее. Однако, в реальной жизни очень сложно найти функцию, описывающую активность объектов. Или же параметры f должны меняться очень часто. Например, мы решили использовать линейную функцию для описания активности пользователя мобильного телефона. Каждый раз, когда он меняет направление или скорость движения, создаётся

запрос обновления базы данных. Такие изменения могут происходить постоянно. Так что этот метод по-прежнему требует слишком много обновлений базы данных. Ещё один недостаток заключается в том, что после некоторого количества обновлений параметров требуется произвести достаточно сложные вычисления, чтобы найти текущее положение каждого объекта. Это значительно ухудшит эффективность процедуры запроса.

Наш вклад, описанный в этой статье, включает в себя следующее:

- Мы предлагаем новую идею, основанную на технике хеширования. Каждый объект помещён в «корзину». Только лишь когда объект переходит в новую корзину, база данных производит обновление. Эта техника существенно уменьшает число обновлений базы, что позволяет системе хранить и индексировать большое количество перемещающихся объектов.
- Мы представляем новую структуру системы. Между сборщиком информации о положении и базой данных мы добавили прослойку и назвали её «позиционная пред-исполнительная часть». Эта прослойка может фильтровать большинство запросов обновления базы, основываясь на правилах, определённых вначале.
- Мы выделяем четыре различных метода. Первый разделяет пространство на маленькие «корзины». Второй допускает пересечения корзин, это снижает количество обновлений базы данных, полученных вследствие зигзагообразных движений объектов. Третий метод позволяет динамическое обновление корзин, это повышает эффективность размещения при неравномерном распределении объектов. И последний, четвёртый, метод совмещает в себе лучшие стороны второго и третьего методов. В экспериментальном разделе мы также приводим некоторые рекомендации по выбору метода.

Во второй части обсуждается соответствующая работа в области пространственно-временных баз данных. Затем, в третьей части, мы излагаем основную идею наших методов, а также новую систему структур данных, необходимых для воплощения идеи. В следующей части мы представляем четыре особых метода. Результаты экспериментов приведены в части 5, а последняя часть содержит выводы и направления дальнейших исследований.

2. Связанная работа

В последнее время была проделана большая работа по индексированию положения движущихся объектов. Эти работы по большей части сконцентрированы на точечных данных. Относящаяся к этой теме работа может быть разделена на две категории в зависимости от информации, размещённой в базе данных.

Первый подход подразумевает размещение информации о положении перемещающихся объектов, полученной путём периодичной выборки. Движение объектов между двумя позициями выборки описывается с использованием интерполяции. Интерполяция может быть как линейной, являющейся простейшей, так и полиномиальной сплайновой [BBB87]. Затем движение одного объекта в d-мерном пространстве описывается как траектория в (d+1)-мерном пространстве, включающем время. [TUW98]. Методы, соответствующие этому подходу главным образом сфокусированы на индексировании траекторий.

В [PTJ99] авторы определяют метод, основанный на R-дереве и названный STR-дерево. Там используется способ линейной интерполяции, так что траектория объекта

представляется в виде множества отрезков. В STR-дереве при выполнении операции вставки отрезки внутри одной и той же траектории наиболее вероятно будут размещены вместе. Позже, в [PTJ00] эти же авторы предлагают другую структуру, TB-дерево, которая сохраняет траектории целиком. Авторы утверждают, что эти два новых дерева работают лучше при индексации движущихся объектов, чем семейство традиционных R-деревьев.

Так как этот подход использует интерполяцию, чтобы описать движение объекта между двумя выбранными точками, могут появиться некоторые неопределённости. Волфсон и др. в [WCD+98] рассматривают проблему неточности и в настоящее время DBMS может обеспечить связность запросов. Пфосер и Дженсен отметили в [PJ99], что при ограничении скорости возможное положение объекта между двумя выбранными позициями должно быть как бы «затмением», а не просто отрезком. И запросы должны этот фактор учитывать.

Недостатком этого подхода является то, что при большом количестве объектов после каждой выборки может происходить слишком много операций с базой данных. Например, в STR-дереве каждая выборка будет генерировать n вставок отрезков в базу, где n - это количество объектов. Ввиду ограниченности базы данных выборка не может происходить очень часто для большого n. Это значительно увеличит фактор неточности и вызовет неоднозначность запроса.

Второй подход же использует функцию для описания перемещения объекта и размещает эту функцию в базе данных. Пусть, например, в одномерном пространстве в момент времени t_0 позиция объекта – x_0, и объект движется с постоянной скоростью v. В любой момент времени t позиция объекта может быть описана как $f(t) = x_0 + v*(t - t_0)$. Если просто поместить f в базу данных, не будет нужды делать обновление базы данных до тех пор, пока объект не изменит скорость.

В [KGT99] авторы используют линейные функции для описания траекторий объектов. Так как в пространственной базе данных весьма непросто индексировать непрерывную линию, авторы отображают линию в точки на двойной плоскости. Такое преобразование позволяет сформулировать задачу более интуитивно понятно. Однако, следует отметить, что запрос приличного диапазона преобразуется к многоугольнику в двойном пространстве. Это чуть усложняет запрос.

А. Систла и др. предлагают модель данных, которая называется MOST в [SWC+97]. В этой модели каждый объект имеет специальный атрибут – функцию. Это функция от времени. Без осуществления явного обновления позиция каждого объекта может быть найдена путём комбинирования этого атрибута с другими стандартными атрибутами (такими как позиция и время). Модель также позволяет DBMS обрабатывать мгновенные, продолжительные и постоянные запросы.

TPR-дерево – структура, основанная на R*-дереве [SJL+99] – использует очень похожую идею. В этом дереве позиция перемещающегося объекта представляется в виде ссылки на позицию и соответствующего вектора скорости. Вектор скорости может рассматриваться так же как атрибут-функция в MOST. TPR-дерево также поддерживает эффективную обработку запросов текущего и прогнозируемого положения движущихся объектов.

Как обсуждалось в первом разделе, этот подход может частично решить проблему обновления базы данных, если движение объектов происходит по какому-либо закону (как, например, движение частиц в научных экспериментах). В реальной жизни же невозможно отыскать простую функцию, которая описывала бы движение людей. Даже скорость автомобиля на трассе может меняться очень часто. По определению, каждое изменение скорости вызывает обновление базы данных. Поэтому общее число обновлений всё ещё велико.

3. Техники хеширования

Этот раздел представляет основную идею техники хеширования и соответствующую структуру данных. Главная цель нашего метода – уменьшить число обновлений базы данных таким образом, чтобы система имела возможность размещать и индексировать большое количество движущихся объектов.

Основное отличие между движущимися объектами и неподвижными заключается в том, что положение движущихся объектов часто меняется. В базе данных, если мы хотим отслеживать точное положение перемещающихся объектов, неизбежным становится большое количество её обновлений. В связи с этим мы предлагаем так называемую «размытую» идею: не стоит обновлять положение объектов в базе до тех пор, пока они не отклонятся от своего первоначального положения достаточно сильно. К примеру, нам надо отследить перемещения путешественника в Вашингтоне. Пусть в момент времени t_0 он находится возле монумента Вашингтона, который есть в нашей базе. Он может ходить вокруг него всё время, однако, мы не будем сохранять эти перемещения, пока он не удалится достаточно от изначальной позиции, скажем, в момент времени t_1, когда он будет находиться на капитолийском холме. Любые движения между t_0 и t_1 попросту не будут отображаться в нашу базу данных.

Появляется некая неопределённость в запросах. Например, если мы хотим найти положение объекта O прямо сейчас, ответом на запрос базы данных будет что-то вроде «объект O в данный момент находится в области, близкой к p_0, где p_0 - это информация о положении, размещённая в базе». Так же в диапазонных запросах: пусть дан диапазон R. Результат содержит две части: какие-то объекты находятся точно в диапазоне запроса, а какие-то потребуют дальнейшего уточнения.

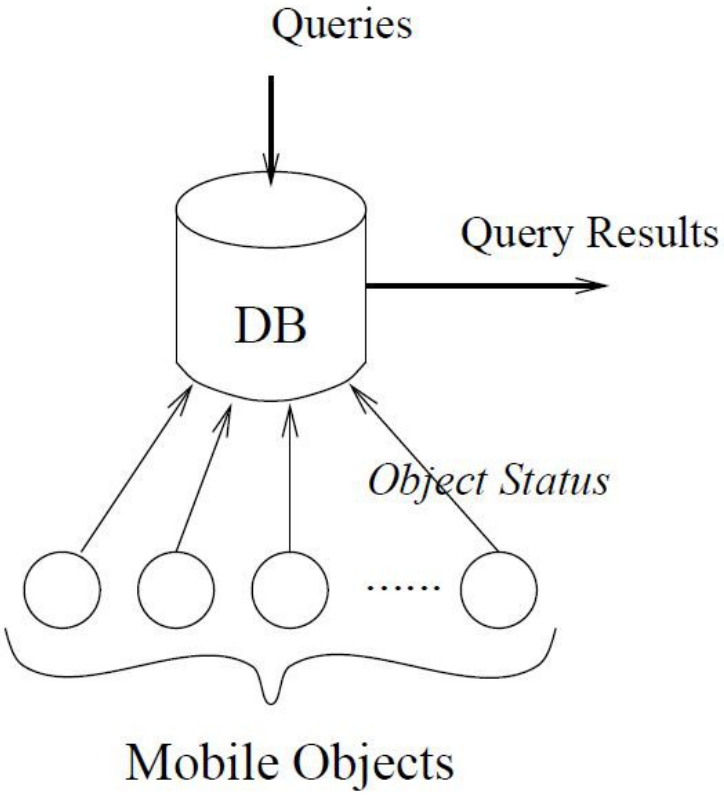

Рисунок 1. Структура обычного метода

С целью решить эту проблему мы разработали целую новую структуру. Перед тем, как мы её представим, давайте сначала посмотрим, на что похожи структуры обычных

методов. На рисунке 1 показан общий принцип. В традиционных структурах движущиеся объекты посылают свою самую последнюю информацию (положение, функцию, скорость и т. д.) напрямую в базу данных. Получив такую информацию, база выполняет соответствующие обновления. База данных всегда хранит последний статус каждого объекта, и ответы на запросы основываются на этой информации. База данных может использовать различные индексные структуры (STR-дерево, TPR-дерево), чтобы ускорить процедуры обновления и обработки запросов.

Структура нашего метода (рис. 2), названная техникой хеширования, работает по-разному. Сначала представим хеш-функцию, которая получает в качестве входных данных текущий статус объекта. Исходя из этой функции система способна найти, к какой корзине принадлежит каждый объект. База данных хранит только информацию о корзинах: сколько объектов в корзине, в какой корзине каждый объект находится в настоящее время. Между базой данных и движущимися объектами мы добавили набор фильтров, который называется «позиционные пред-исполнительные части» (LP). Каждая такая часть следит за маленьким подмножеством объектов и использует массив для хранения последнего статуса этих объектов.

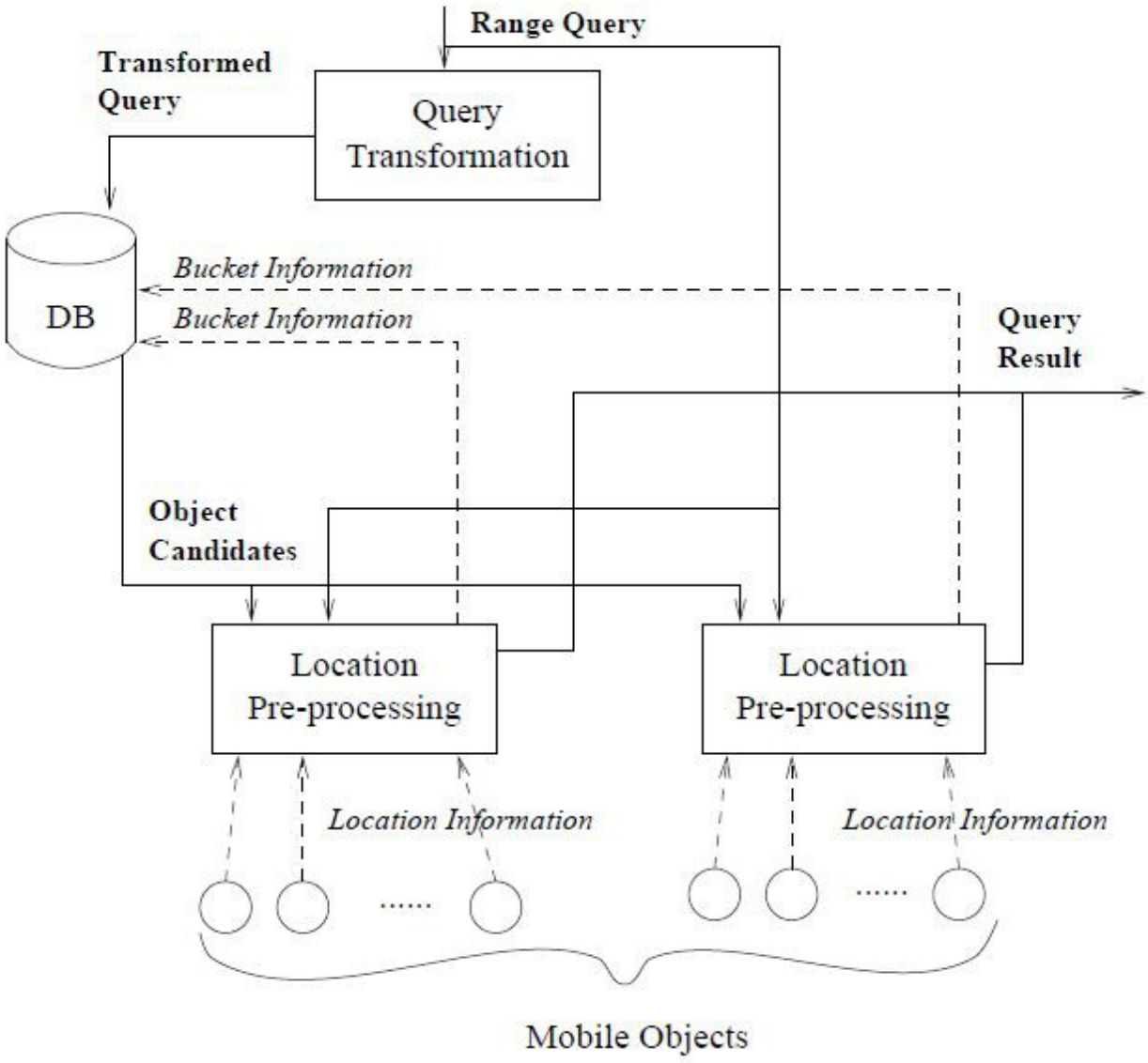

Рисунок 2. Структура техники хеширования

Когда объект меняет своё местоположение и генерирует запрос обновления, запрос сначала направляется в соответствующую LP. LP локально обновляет статус объекта, затем она применяет хеш-функцию к последнему статусу объекта, чтобы посмотреть, находится ли объект в той же корзине. Если да, запрос просто игнорируется. Для тех объектов, которые перемещаются в новую корзину, запросы преобразуются в запросы обновления корзины и посылаются в базу данных.

В нашей структуре весьма большая работа базы данных проделывается в LP. Такой подход обладает множеством преимуществ. Во-первых, каждая LP просматривает малое количество объектов, и LP работают параллельно. Когда объекты обновляют статус, система может сразу же закончить соответствующее изменение. Поэтому становится возможным делать выборки с большей частотой. Во-вторых, система очень хорошо масштабируется. Когда количество объектов растёт, нужно просто добавить больше LP. И нет необходимости производить значительные изменения в базе данных. Эта разработка делает возможной обработку большого числа объектов.

Каждая корзина в базе данных содержит все объекты, которые имеют одно и то же значение хеш-функции. Иногда корзина может рассматриваться как регион в рабочем пространстве. Регион – это объединение всех возможных положений, которые имеют одинаковое значение хеш-функций.

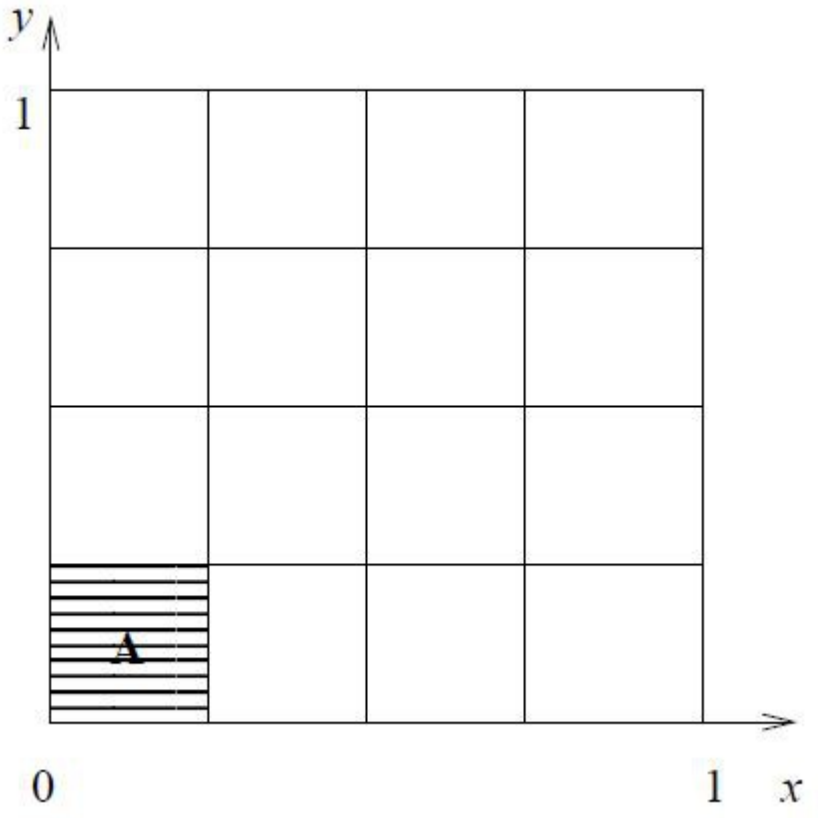

Рисунок 3. Хеш-функция разбила рабочее пространство на 16 регионов

Пример 3.1. Пусть двухмерное рабочее пространство – это квадрат $[0,1]^2$. Выбранная хеш-функция: $f(p_{x,y}) = (int(y*4))*4 + int(x*4)$, где (x,y) - это текущее положение. Использование этой функции разбивает рабочее пространство на 16 регионов (см. рис. 3). В этом случае корзина 0 может рассматриваться как регион A, потому что если объект в регионе A, и если применить к нему хеш-функцию, получим, что объект должен находиться в корзине 0.

Как говорилось ранее, появляется некоторая неточность, если просто использовать базу данных для ответа на запросы. В нашей же структуре нижеследующие части разработаны специально для того, чтобы минимизировать или вовсе исключить неточности. Когда приходит диапазонный запрос, сначала он посылается в часть «трансформации запроса» (QT). QT преобразует диапазонный запрос в запрос корзины. К примеру, предположим, что запрос выглядит так: «Найти все корзины, пересекающиеся с R». Каждая корзина в базе данных имеет один из трёх следующих статусов:

1. Корзина не пересекается с диапазоном запроса. В этом случае объекты корзины не могут быть в диапазоне запроса.
2. Диапазон запроса перекрывает корзину. Значит, объекты корзины должны быть в диапазоне запроса.
3. Корзина пересекается с диапазоном запроса. Этот случай – кое-что посложнее. Пока база данных не имеет возможности различить, какие объекты корзины попадают в диапазон, а какие нет. Есть два возможных пути. Первый: с помощью статистики дать приблизительный результат. В этой области было проделано много работы [PIH+96, APR99]. Второй ищет все объекты в корзине и отправляет их id в LP. Затем LP проверяют последнюю позицию объектов, чтобы увидеть, действительно ли они находятся в диапазоне запроса, а затем отправляют результат. Мы применяем последний подход.

На рисунке 2 курсивом показана процедура индексирования, а жирным шрифтом - процедура запроса.

Пример 3.2. (традиционная структура). Наша структура в состоянии использовать одну LP, чтобы смоделировать традиционное решение. Определим хеш-функцию $f(p) = 0$ для каждого объекта p. База данных имеет только одну корзину, и она пересекается с любым диапазоном запроса. В соответствии с нашей разработкой, LP перепроверяет каждый объект и сообщает результат. При таком моделировании наша база данных становится бесполезной, а всю работу выполняет LP.

Разработанная нами структура очень гибка. Выбирая разные хеш-функции, мы получаем разные методы. В следующем разделе мы представим четыре различных метода, основанных на этой структуре.

4. Хеш-функции

Этот раздел посвящён четырём различным хеш-функциям, а также здесь приведено подробное описание соответствующих методов. Первая функция основана на методе разделения пространства. После проверки производительности обнаружилось два основных недостатка. Следующие два метода эти недостатки устраняют. Последний метод совмещает в себе две предыдущие идеи.

4.1. Метод разделения пространства на неперекрывающиеся части

Одной из главных целей использования техники хеширования является уменьшение числа обновлений базы данных. Основная идея первого метода заключается в том, чтобы разделить пространство на несколько частей. Каждая часть соответствует корзине базы данных. Только если объект покидает одну часть и переходит в другую, выполняется обновление базы. Подробности изложены ниже.

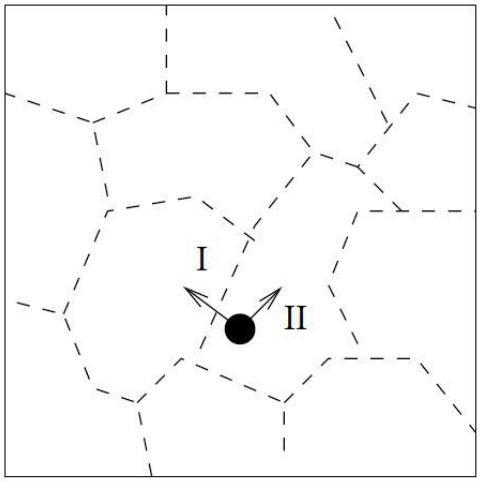

Рисунок 4. Пример движений объекта

Первый шаг этого метода – разделение пространства на множество маленьких кусочков. На рисунке 4 рабочая область – это квадрат. Пунктирные линии делят область на 12 частей. Регион, охватываемый каждой частью, рассматривается как корзина базы данных. Совершенное разделение предусматривает, что каждая корзина в любой момент времени содержит примерно одинаковое количество объектов. Однако, разделение производится заранее, и мы не располагаем данными о том, как же движутся объекты. Очень сложно сразу найти подходящее разделение. В каком-нибудь особом случае, когда, к примеру, объекты распределены равномерно и движутся хаотично, разделение на одинаковые зоны может стать почти совершенным. Если известно, что объекты двигаются возле какого-то предопределённого места, можно использовать диаграмму Вороного.

После разделения пространства следует присвоить каждой части уникальный идентификатор. Хеш-функция сейчас выглядит так: $f(p) = i$, где p - это объект, а i - идентификатор корзины, в которой находится p.

Если в момент времени t объект переходит из одной части в другую (например, путь I на рисунке 4), LP, следящая за этим объектом, посылает запрос обновления в базу данных. Запрос будет похож на что-то вроде *update(part_id, old_bucketid, new_bucketid, t)*. Иногда после смены позиции объект остаётся в прежней части (путь II на рисунке 4). И в этом случае база данных ничего не знает о подобных перемещениях.

После определения *f* запрос становится интуитивно понятен. В случае специального запроса база данных отправляет его LP, которая следит за объектом. LP узнаёт текущее положение объекта и выдаёт результат. Для диапазонного запроса процедура в точности такая же, как та, что обсуждалась в предыдущем разделе. Единственное, на чём бы хотелось заострить внимание, так это на том, что информация корзины в этом случае статична, а это значит, что эта информация (размер, положение и т. д.) не меняется при однажды определённой хеш-функции f. Это позволяет нам использовать какую-либо из существующих пространственных индексных структур (R*-дерево, Quad-дерево) для того, чтобы разместить корзины в базе данных. Это существенно ускорит процедуры запросов к базе данных.

Ещё одна важная проблема, требующая рассмотрения – размер частей. С одной стороны, если разделить пространство на очень большие части, объекты, вероятно, будут перемещаться, не покидая пределов одной части. Это значит, что потребуется меньше обновлений базы данных. Но в то же время при обработке диапазонного запроса корзины имеют больше шансов быть затронутыми, а это вызовет больше проверок объектов и участит связь базы данных с LP. С другой стороны, если размер корзины слишком мал, то

хотя запрос и может стать быстрее (база данных в состоянии завершить большую часть обработки, и меньшее количество объектов потребует дальнейшей проверки с помощью LP), затраты на размещение корзин в базе данных значительно возрастают. База данных будет подавлена огромным количеством запросов обновления, которые могут быть неотфильтрованы LP. Для этого есть компромисс. В следующем разделе мы представим затраты на операции и подробнее обсудим, как же правильно выбрать размер частей.

4.2. Метод разделения на увеличенные части

По сравнению с другими методами, метод разделения неперекрывающихся частей порождает меньше обновлений базы данных. Однако, в таком подходе есть два недостатка. В этом и следующем подразделах они будут рассмотрены отдельно.

В методе разделения пространства на неперекрывающиеся части объекты, перемещающиеся зигзагообразно вдоль границы корзины, могут спровоцировать серьёзную проблему. Так как одна зона рабочей области соответствует корзине базы данных, далее в статье эти термины имеют одинаковое значение.

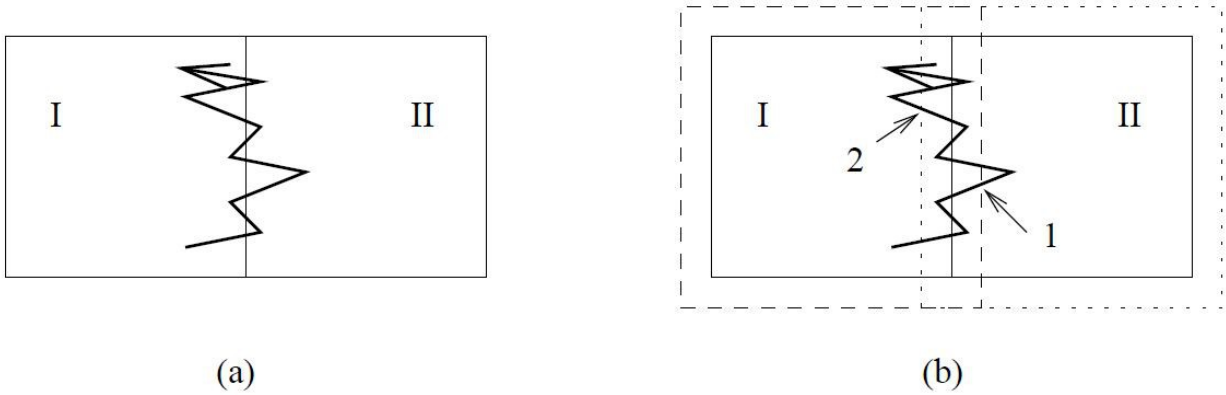

Рисунок 5. Объект перемещается вдоль границы

Посмотрите на рисунок 5(a). I и II – это две корзины. Объект движется вдоль их границы. Каждый раз, когда он пересекает границу (из корзины I в корзину II или наоборот), LP генерирует запрос обновления. Сначала объект в корзине I. По завершении его пути мы получаем целых восемь запросов обновления базы данных.

Чтобы разрешить эту проблему, мы чуть-чуть увеличим размер каждой корзины так, чтобы между корзинами появилась зона перекрытия. Запрос обновления будет генерироваться только если объект покинет увеличенную зону. Посмотрите на рисунок 5(b). Пунктирный и точечный квадраты представляют две увеличенные корзины. Сначала рассмотрим объект в корзине I (пунктирный квадрат). В точке 1 он покидает корзину I и переходит в корзину II (точечный квадрат). Затем он перемещается по корзине II и в точке 2 снова возвращается в корзину I. Теперь такой же путь объекта вызовет только два обновления базы данных. Метод разделения на увеличенные части работает следующим образом. Сначала создаются неперекрывающиеся части. Затем каждый объект хешируется в корзины в соответствии со своим изначальным положением. После этого каждая корзина увеличивается на величину *б*. Такое увеличение означает, что центр корзины не меняется, а внешние стороны увеличиваются на малую величину *б*. Например, если корзина покрывает прямоугольник $[x_0, y_0]$ $[x_1, y_1]$, то после увеличения она будет покрывать прямоугольник $[x_0-б, y_0-б]$ $[x_1+б, y_1+б]$.

На следующем шаге нужно найти хеш-функцию и поместить её в LP. Если просто использовать информацию о текущем положении объекта в качестве входных данных, возникнет проблема: когда объект перемещается в зону покрытия двух корзин, функции

будет непросто решить, в какой именно корзине должен быть объект. Поэтому в этом методе мы представляем новый атрибут, который называется previous_bucketid. Этот атрибут запоминает, в какой корзине объект был доселе. Он также передаётся в хеш-функцию. Каждый отрезок времени, если объект остаётся в зоне, покрытой только одной корзиной, всё нормально. Иначе же, если зона покрывается более, чем одной корзиной, LP сперва проверяют, находился ли объект ранее в какой-нибудь из этих корзин. Если да, LP не посылают запрос обновления базе. Если нет – корзина для объекта определяется произвольно.

Последующая часть индексирования и процедура запроса такие же, как в методе разделения пространства на неперекрывающиеся части. Мы не приводим их детали здесь.

4.3. Метод хеширования с Quad-деревом.

Ещё один недостаток метода разделения пространства на неперекрывающиеся части состоит в том, что корзины не имеют возможности менять свои размеры и положение после первоначальной установки. Нижеследующий пример демонстрирует, что иногда это может вызвать неприятности.

На рисунке 6 показан случай неравномерности. На этот раз будем использовать функцию разделения на одинаковые части: в базе появляется 16 корзин. В момент времени t_0 все объекты находятся в верхнем левом углу. Потом они начинают движение по стрелке. В момент t_1 объекты достигают правого нижнего угла. Между t_0 и t_1 всего несколько корзин могут содержать объекты. (*только* в моменты t_0 и t_1). Остальные же пусты. В таком случае наша индексная структура неэффективна при диапазонном запросе.

Чтобы найти решение этой проблемы, представляем некоторые динамические структуры. Основная идея заключается в том, чтобы динамически изменять зоны покрытия каждой корзины. Если корзина содержит слишком много объектов, мы её расщепляем на несколько корзин поменьше и перераспределяем объекты старой корзины. С другой стороны, если несколько корзин содержат слишком мало объектов, мы сливаем их в одну корзину побольше и размещаем объекты вместе.

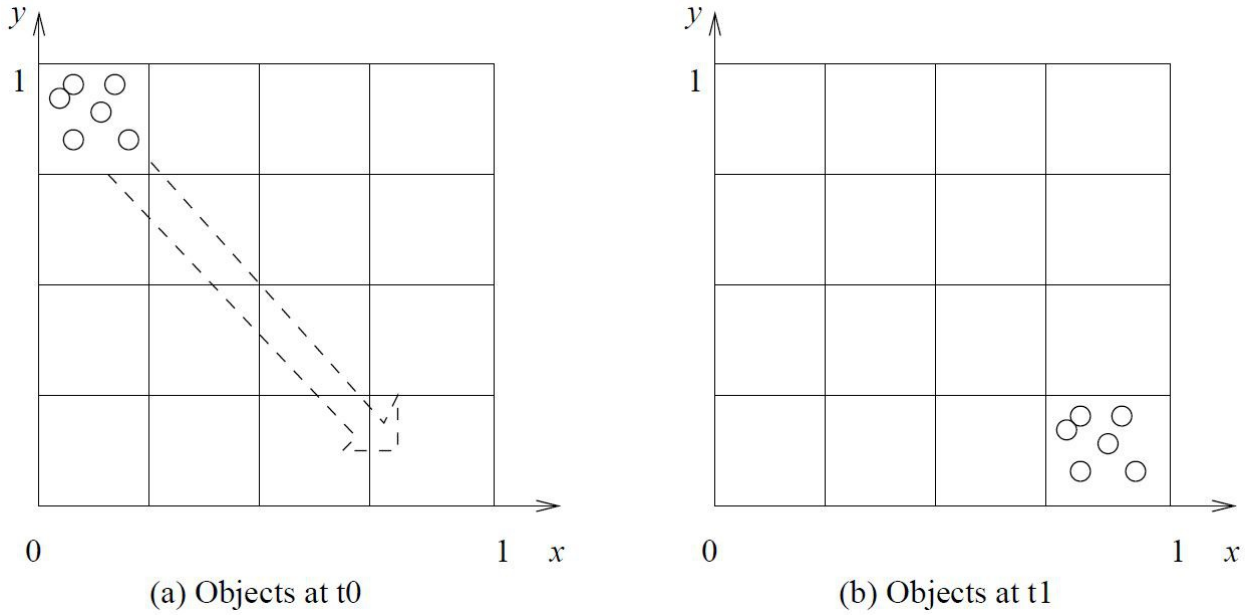

Рисунок 6. Неравномерное распределение объектов

Некоторые детали:

- В базе данных создаётся новая часть, которая называется «часть управления корзиной» (ВМ). ВМ используют пространственную индексную структуру для организации корзин. Любые изменения в корзинах (добавление или удаление объекта) повлекут за собой действия со стороны ВМ. ВМ проверяет изменение в корзине и решает, нужна ли операция расщепления (слияния).
- В качестве пространственной структуры мы использовали Quad-дерево [Sam90], так как оно имеет простую структуру и алгоритмы расщепления и слияния. В отличие от R-дерева, каждый внутренний узел Quad-дерева имеет строго четыре потомка, и ни один узел не пересекается с другим.
- В узле хранится следующая информация: зона покрытия, количество объектов внутри зоны в настоящий момент, флаг о том, листовой это узел или нет, указатели на потомков, если не листовой, указатель на родителя, если не корень и т. д. Каждый листовой узел соответствует корзине в базе данных.
- Когда количество объектов одного листового узла превышает M, этот узел расщепляется. Пусть M – это максимальное число объектов, помещающихся на одной странице диска. Алгоритм расщепления сначала создаёт четырёх потомков, каждый из которых покрывает четверть родительской зоны. Затем алгоритм сообщает базе данных, что надо создать четыре корзины. Объекты старой корзины проверяются и помещаются во вновь созданные корзины. После этого старые корзины удаляются из базы данных, и количество объектов в новых корзинах сообщается Quad-дереву и сохраняется в новых листовых узлах. Алгоритм расщепления вызывается рекурсивно, если какая-либо из корзин-потомков всё ещё содержит объектов больше, чем M.
- Условие для слияния чуть посложнее. Если листовой узел содержит менее m объектов, нужно проверить, сколько объектов содержится в соседних узлах-братьях. Очень вероятно, что один из братьев всё ещё плотный (т.е. содержит много объектов). Так как слияние – это операция дорогостоящая, не хотелось бы слишком скоро расщеплять вновь слитую корзину. Поэтому в своём алгоритме мы определяем условие как «листовой узел содержит менее m объектов, и число объектов его родителя менее 3*M/4». Когда узел встречает это условие, его родительский узел становится новым листом дерева. База данных создаёт новую корзину. Объекты старых корзин перемещаются в новую, а старые попросту удаляются.
- Так как теперь структура корзины динамична, LP нужно знать о текущей структуре, чтобы должным образом фильтровать запросы обновления. Есть два пути. Первый метод – позволить ВМ транслировать индексную структуру LP после каждого её изменения. Этот метод осуществим, только если структура корзин меняется нечасто или в системе не слишком много LP.

Другой метод – разрезать рабочее пространство на кусочки поменьше – объединения. Объединения не имеют возможности расщепляться. Каждая корзина – это множество объединений. Разделим, к примеру, двухмерную область на 2^{10} x 2^{10} одинаковых по размеру объединений. По определению Quad-дерева, узлы уровня меньшего, чем 10 – это квадраты, покрывающие какие-либо объединения, и ни одно объединение не расположено в двух различных узлах Quad-дерева. LP знают размер объединения заранее. Если объект остаётся в одном и том же объединении, считается, что невозможно «перелистнуть» корзину. LP фильтруют движения такого типа. Если объект покидает объединение, он может либо остаться в прежней корзине, либо переместиться в

другую. Для таких движений LP не могут решить, фильтровать их или нет. Поэтому LP передают их BM. BM перепроверяет запросы и выполняет повторную фильтрацию.

Алгоритм расщепления BM требует дальнейших изменений, пока он не будет подходить к последнему случаю.

При расщеплении узла нужно сделать ещё одну проверку, чтобы определить, одинакового ли размера узел и объединение. Если да, то расщепления больше не допускаются.

Код алгоритмов расщепления и слияния приведен в приложении А.

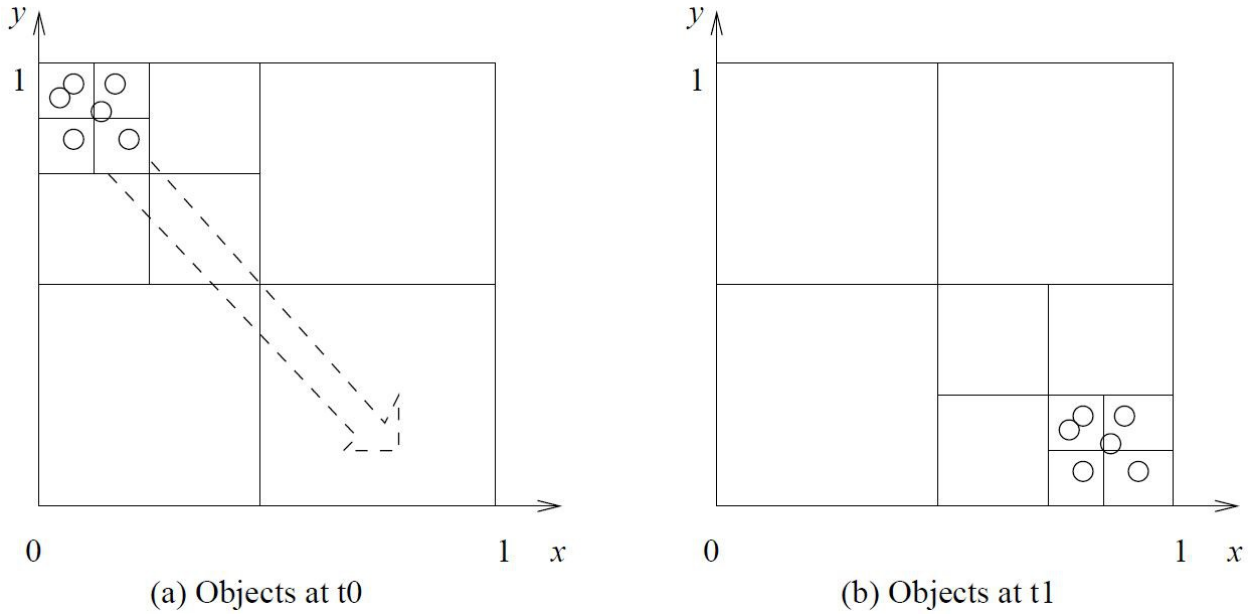

Рисунок 7. Динамические структуры корзин

На рисунке 7 показано, как выглядят корзины после применения структуры динамических корзин.

При обработке запроса его диапазон сначала попадает к BM. BM использует индексную структуру, чтобы решить, какие из корзин требуют дальнейшей проверки. Следующие этапы практически такие же, как и раньше.

4.4. Расширенный метод хеширования с Quad-деревом

Последний метод совмещает в себе идеи метода хеширования с Quad-деревом и метода разделения на увеличенные части. В этот раз мы чуть-чуть увеличим каждый узел Quad-дерева.

Сначала построим Quad-дерево, основанное на начальном распределении объектов. Затем, как в методе разделения на увеличенные части, выполним *б*-увеличение всех узлов (и внутренних, и листовых). Благодаря простой структуре Quad-дерева после этого шага древовидная структура всё ещё в приемлемом состоянии, т. е. область каждого внутреннего узла всё ещё покрывает зоны своих потомков.

Как в методе разделения на увеличенные части, EQ-дерево использует увеличенные узлы при индексировании. Алгоритмы индексирования, вставки и удаления почти такие же, как в методе хеширования с Quad-деревом. Поэтому не будем более говорить о них.

В последних четырёх подразделах мы обсудили четыре метода, основанных на нашей новой структуре системы. В следующей, экспериментальной, части мы подробнее расскажем, как выбрать подходящий размер корзины и сравним производительность этих четырёх методов на примере обработки диапазонного запроса.

5. Результаты экспериментов

Чтобы использовать достоинства описанных методов, мы написали программу-симулятор и оценили их.

5.1. Экспериментальная установка и генерирование данных

Так как реальных данных в предметной области доступно немного, мы напишем собственный симулятор данных, основанный на популярной и широко используемой среде для сравнительных анализов «Generate Spatio-Temporal Data (GSTD)» [NST99]. Как и GSTD, наш симулятор поддерживает три начальных распределения: равномерное, по Ципфу и по Гауссу. В нашем случае мы только лишь индексируем и запрашиваем текущий статус движущихся объектов. Поэтому мы внесли следующие изменения:

- В нашей системе используется единый промежуток времени для всех объектов, и каждый объект сообщает своё последнее местоположение после каждого такого промежутка. Глобальный таймер работает постоянно. В GSTD промежуток каждого объекта находится в пределах некоего домена. И там присутствует верхняя граница глобального таймера. После прохождения этой границы все объекты становятся неактивными.
- В нашей системе объекты, вышедшие за пределы рабочей области, всё ещё считаются активными. Позже они ещё способны вернуться в рабочую область. В GSTD вышедшие объекты помечаются как неактивные.
- Система GSTD обеспечивает данными для индексации траекторий. Поэтому ей требуется вести очень большой журнал, хранящий записи о каждом движении объектов. В нашей же системе нам требуется только текущее положение объектов, поэтому мы не ведём этот журнал. Это позволяет нам проводить эксперимент на большом числе объектов и в течении длительного промежутка времени.
- Объекты в нашей системе могут иметь различные начальное распределение и тип движения. Например, мы допускаем ситуацию, при которой одна половина объектов распределена неравномерно, а вторая половина – напротив, равномерно. В GSTD все объекты должны иметь единое начальное распределение.

В своём эксперименте мы использовали язык Java ввиду хорошей поддержки им многопоточности. После каждого промежутка времени каждый движущийся объект вычисляет своё новое положение. В системе присутствуют две LP. Каждая из них следит за половиной объектов. После каждого промежутка времени LP проверяют последний статус движущихся объектов и посылают фильтрованную информацию BM. BM собирает информацию и сообщает о результате эксперимента.

Используемая нами машина имеет следующие компоненты: процессор Pentium II частотой 300MHz и 128Mb оперативной памяти. Используется 20 байт для описания каждого двухмерного объекта (2 переменные типа double для положения и одна переменная типа int для идентификатора). Размер страницы – 4096, что позволяет размещать на одной страницы до 204 двухмерных объектов. Для обоих методов с Quad-деревом мы выбрали M равным 200 и m равным 50.

Большинство других методов также индексируют историческую информацию для каждого объекта. Но они не позволяют индексировать большое число объектов. Поэтому несправедливо сравнивать с ними наш метод. Помимо наших четырёх методов мы также

использовали метод с R-деревом. Он использует традиционный статичный метод в задаче движущихся объектов. Он также содержит R-дерево в базе данных. После каждого промежутка времени R-дерево обновляется в соответствии с последним положением объектов.

5.2. Обозначения

Обозначения, использованные в этом разделе для удобства описаны в таблице 1. А аббревиатуры алгоритмов, использованных в этой части, находятся в таблице 2.

5.3. Затраты на индексирование и обработку запросов

Перед запуском экспериментов исследуем факторы, влияющие на итоговую производительность.

Общие затраты на индексирование C_{index} главным образом состоят из трёх частей: обновление базы данных, изменение структуры корзины и связь между LP и базой данных. Другие факторы, такие как сбор информации о последнем местоположении или обновление LP не затрачивают много ресурсов ввиду параллельной структуры системы. Поэтому, $C_{index} = DU\#*C_u+C_b+C_c$.

v	Скорость объекта
$\acute{v}$	Средняя скорость всех объектов
σ(v)	Стандартное отклонение скорости
S	Размер корзины
DU#	Количество обновлений базы данных
σ(D)	Стандартное отклонение начального распределения
Cu	Затраты на одну операцию обновления базы данных, включающую процедуры удаления и вставки
Cb	Затраты на изменение структуры корзины
Cc	Затраты на связь между LP и базой данных
Cq	Затраты на запрос к базе данных

Таблица 1. Обозначения, использованные в эксперименте

RT	Метод с R-деревом
SP	Метод разделения на неперекрывающиеся части
ASP	Метод разделения на увеличенные части
QH	Метод хеширования с Quad-деревом
EQH	Расширенный метод хеширования с Quad-деревом

Таблица 2. Аббревиатуры алгоритмов

Для методов SP и ASP $C_b = 0$, т. к. их структуры не имеют корзин. Для других же методов, т. к. структуры корзин содержатся в памяти, C_b во многом определяется затратами на перемещение объектов из старых корзин в новые.

Итоговая стоимость запроса C_{query} включает в себя две части. Первая – это общая стоимость запроса базы данных, а вторая - стоимость связи между базой данных и LP. Так что $C_{query} = C_q+C_c$.

5.4. Наборы данных

Набор данных состоит из 100 000 объектов в рабочей области. Мы изучаем производительность различных методов на двух типах начального распределения объектов и двух типах движения, описанных ниже.

5.4.1. Два типа начального распределения

Хотя, как и в GSTD, мы воплотили три типа начального распределения, мы считаем, что только два из них полезны для реального применения. Первый – это равномерное распределение. В этом случае объекты равномерно распределены по рабочему пространству. Второй – распределение по Гауссу. Здесь объекты сгруппированы вокруг одной или нескольких центральных точек. Такое распределение можно рассматривать как неравномерное. Также в нашем случае мы допускаем комбинацию этих двух распределений (к примеру, одна половина объектов распределена равномерно, а другая – по Гауссу).

При использовании распределения по Гауссу значение $\sigma(D)$ установлено в 0.1 для области объединения.

5.4.2. Два типа движения

Метод GSTD очень хорош для описания движения объектов. В своей программе мы одолжили идеи этого метода. В общем, мы определили два типа движения для эксперимента. Первый – хаотичное движение. Его параметры приведены в таблице 3. Детальное описание каждого из них можно найти в [NST99].

Второй тип движения, который мы назвали направленным движением, используется, чтобы сымитировать движение объектов из нижнего левого угла в верхний правый. Параметры его – в таблице 4. Этот тип хорош, если мы изучаем движение автомобилей в час пик.

$\acute{v}$	$\sigma(v)$	$max_x(speed)$	$max_y(speed)$	$min_x(speed)$	$min_y(speed)$
0	0.005	0.005	0.005	-0.005	-0.005

Таблица 3. Параметры произвольно движущихся объектов

$\acute{v}$	$\sigma(v)$	$max_x(speed)$	$max_y(speed)$	$min_x(speed)$	$min_y(speed)$
0.005	0.005	0.01	0.01	0	0

Таблица 4. Параметры для направленно движущихся объектов

5.5. Множество запросов

Множество запросов состоит из тысячи прямоугольников, находящихся внутри рабочей области. Центры прямоугольников выбираются произвольно. Размер прямоугольника запроса составляет 1% от общей площади рабочей области. В конце каждого промежутка времени из множества запросов произвольно выбирается один и применяется к текущему распределению объектов.

5.6. Результаты эксперимента

В первом эксперименте сравним производительность различных техник как при индексировании, так и при обработке запросов.

5.6.1. Эксперимент 1: влияние размера корзины

В этом разделе мы бы хотели исследовать производительность индексирования при разных S и v методом SP.

Мы определили хеш-функцию так: *f(x,y) = i*int(y*i)+int(x*i)*. Меняя значение i, мы получим i x i одинаковых по размеру корзин. Затем присвоим v значение 0.005 и запишем производительность для разных значений i. Результат показан на рисунке 8. Из него следует, что производительность пропорциональна i. На рисунке 9 мы сделали постоянным S, благодаря чему удалось узнать, что производительность также пропорциональна v. Так что основной вывод эксперимента таков: DU# $\propto \frac{v}{\sqrt{S}}$.

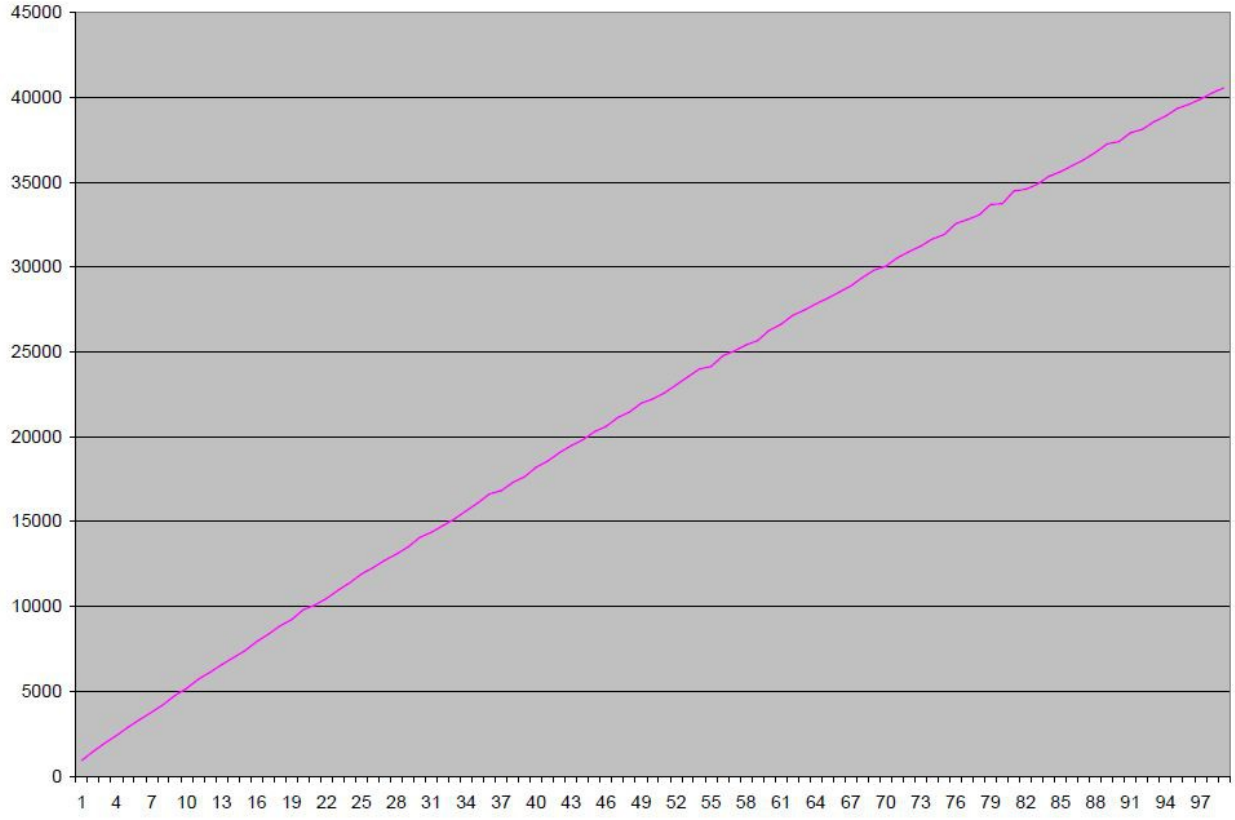

Рисунок 8. Производительность vs S, v = 0.005

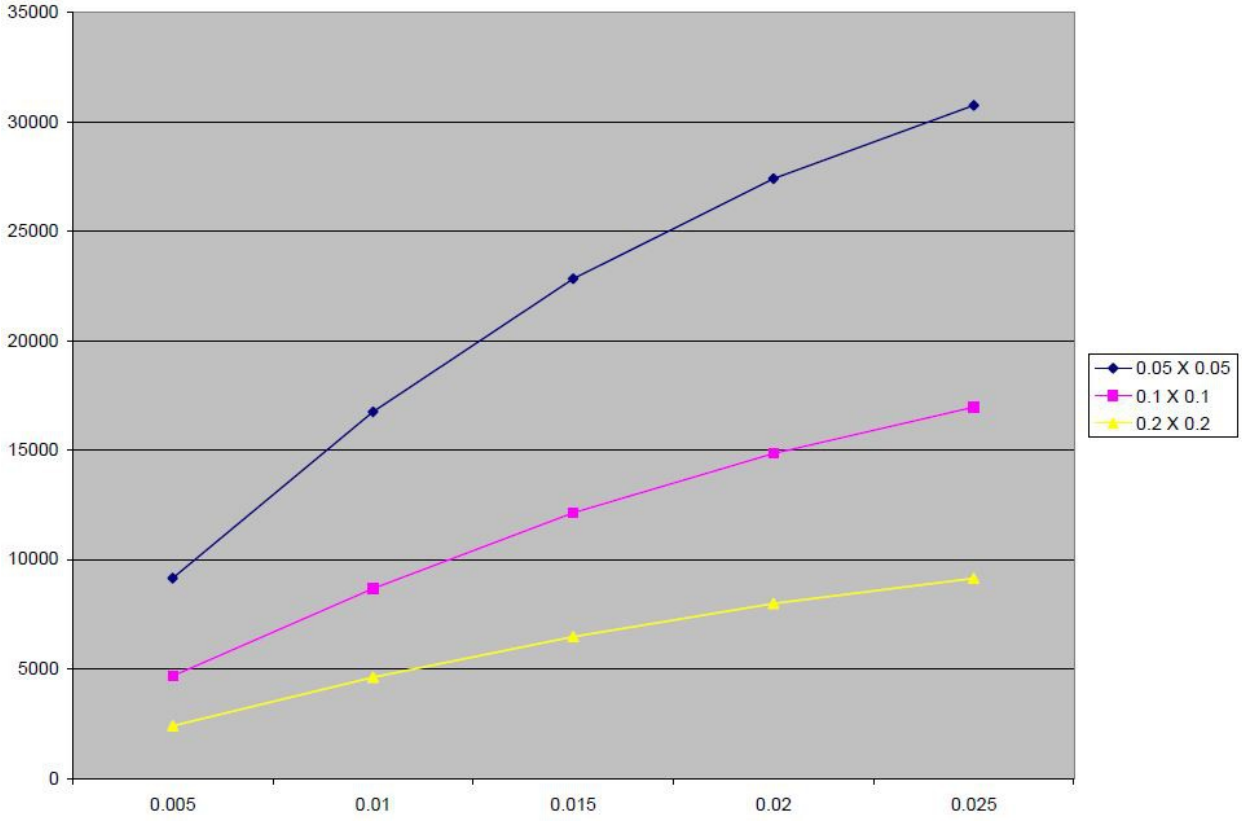

Рисунок 9. Производительность vs v

5.6.2. Эксперимент 2: влияние размера перекрытия

В этом эксперименте мы смотрим на влияние размера перекрытий при методе ASP. Размер корзины: 0.1 x 0.1. Рисунок 10 показывает среднее число обновлений базы данных для различных размеров перекрытий. Значение x – это размер увеличения для каждой стороны, y – общее число обновлений базы данных. Две кривые линии показывают результаты при разных v.

Понятно, что допуск перекрытий между корзинами значительно помогает. Это может отфильтровать множество запросов на обновление базы данных, сгенерированных объектами вблизи границы корзины. Заметим, что кривые стремительно убывают в начале, а затем убывают медленнее в обоих случаях. И это явление более очевидно при низких значениях v. Мы предлагаем взять v за величину перекрытий. Например, если v равна 0.005, мы должны делать расширение каждой корзины на 0.005.

5.6.3. Эксперимент 3: исследование производительности индексирования

В этом эксперименте мы бы хотели узнать производительность индексирования различных методов при различном начальном распределении и типе движения. Мы считаем количество обновлений базы данных, а дисковые страницы используются для размещения всех данных. Количество обновлений базы данных состоит из двух частей: запросы обновления базы, сгенерированные LP и операции обновления при слиянии и расщеплении корзин. На рисунках 11 и 12 приведены результаты.

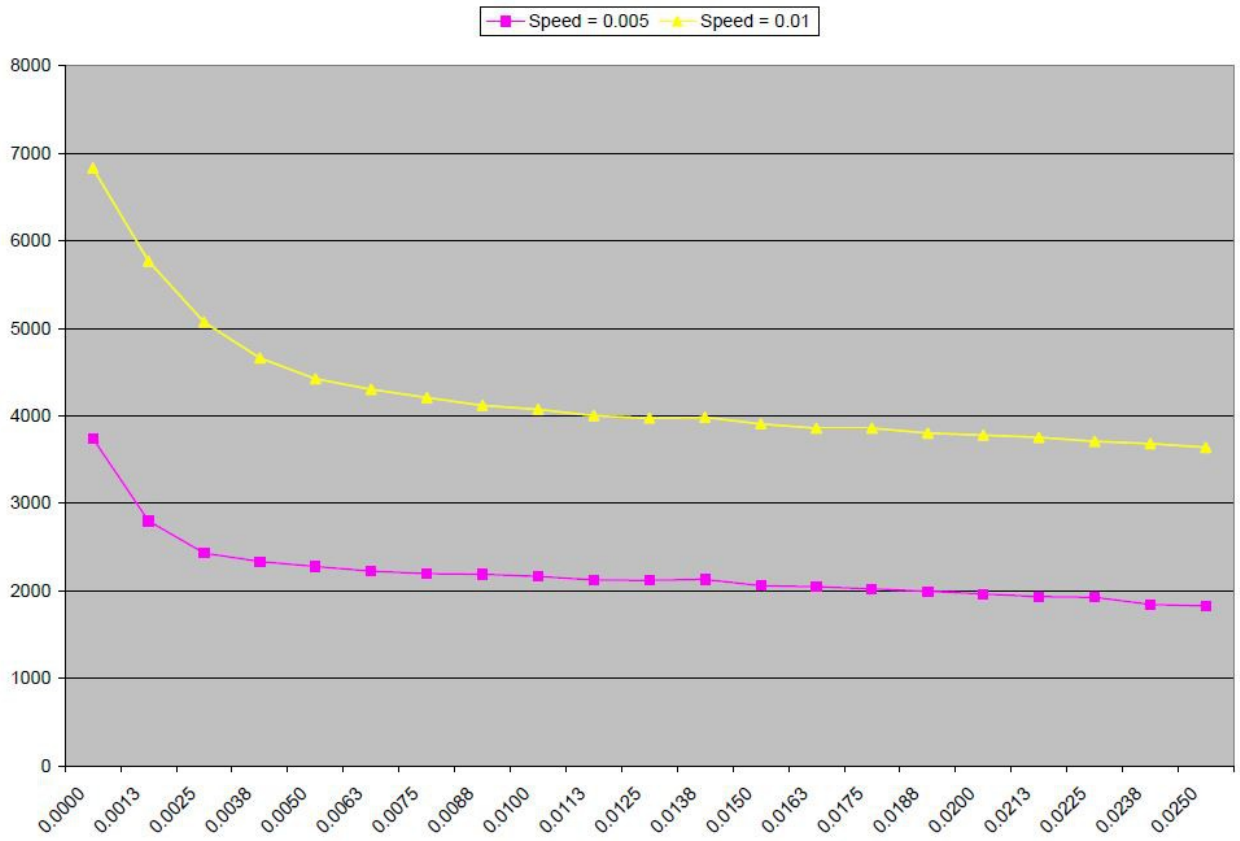

Рисунок 10. Влияние размера перекрытий

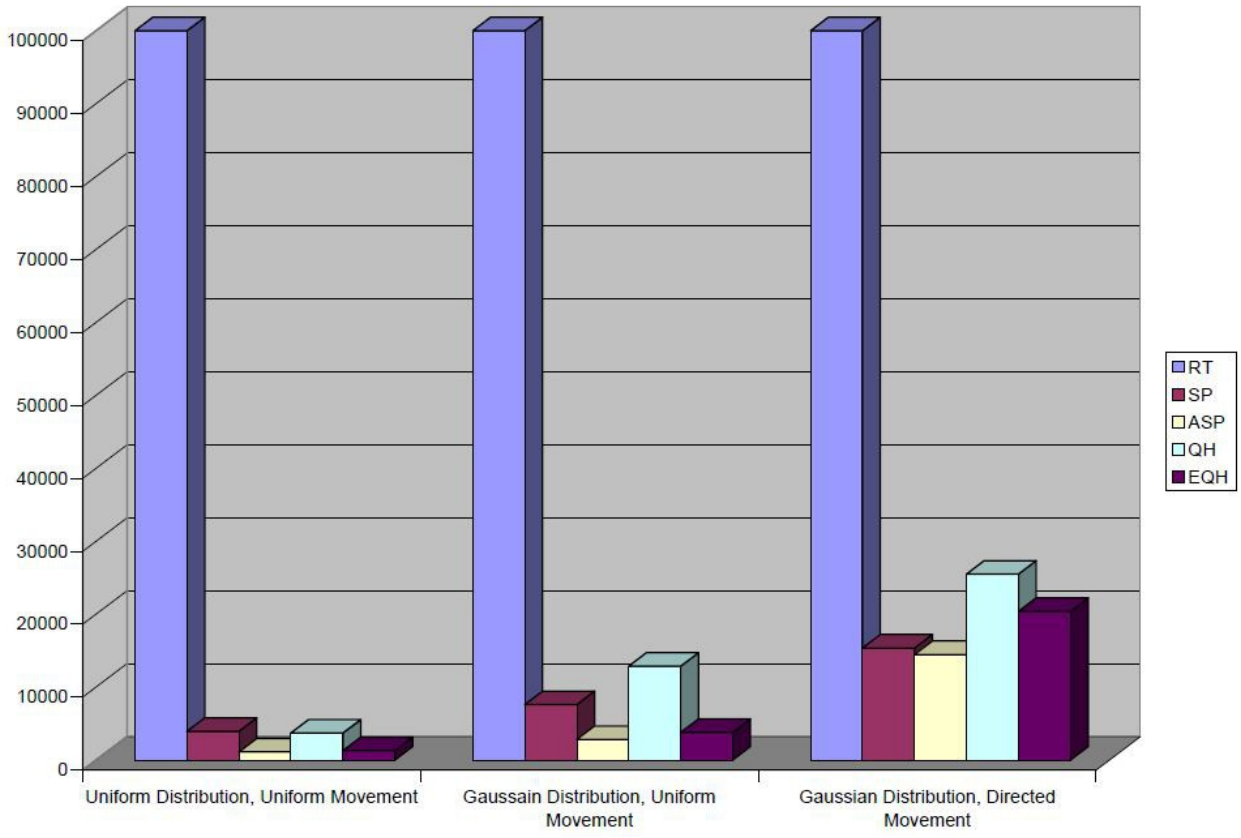

Рисунок 11. Количество обновлений базы данных

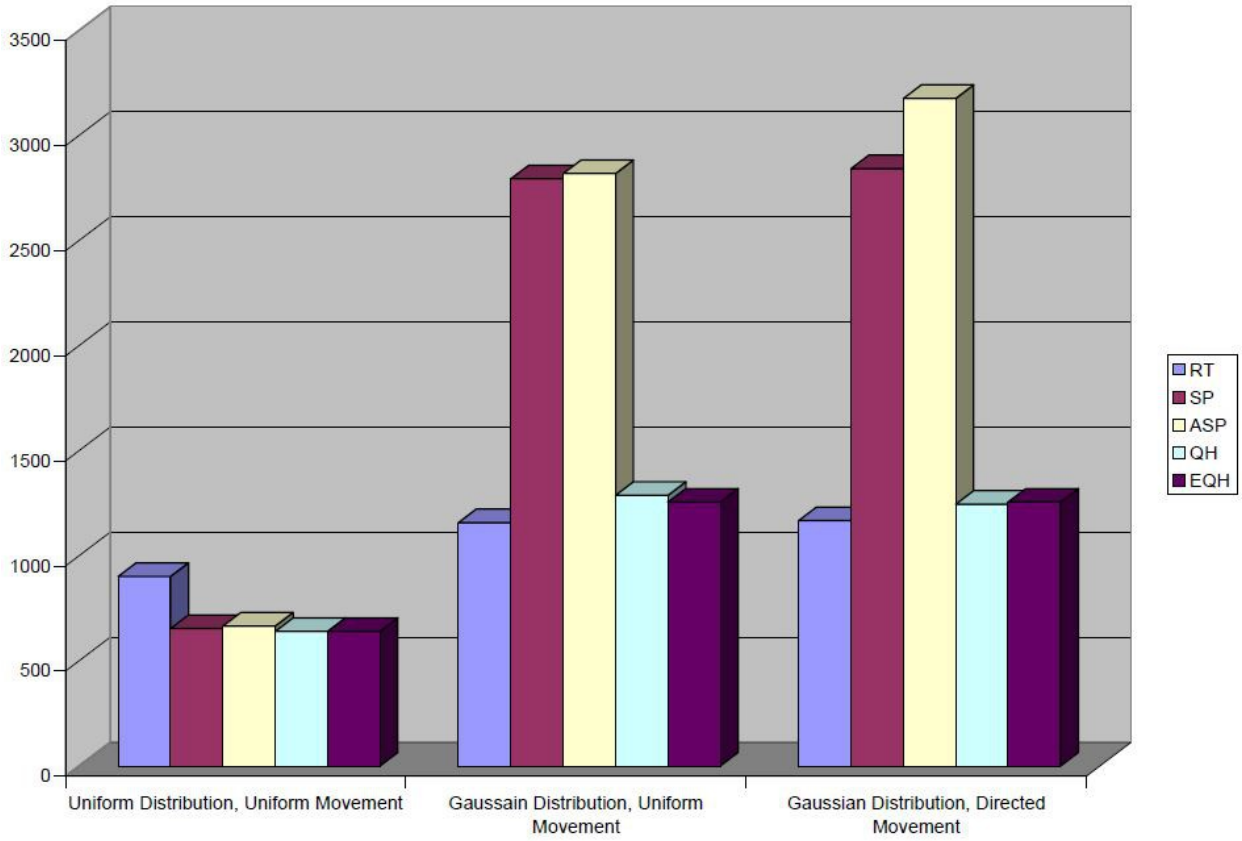

Рисунок 12. Количество использованных дисковых страниц

Факты, которые можно заметить из этих рисунков:

- Метод RT обновляет положение всех объектов после каждого промежутка времени. Поэтому количество обновлений базы данных равно количеству объектов. Остальные четыре метода намного лучше, чем RT.
- При равномерном начальном распределении объектов. Производительность и QH такая же, как и в SP. Это нормально, т. к. если объекты равномерно распределены, Quad-дерево хорошо сбалансировано, и листовые узлы имеют одинаковую высоту. Поэтому листовые узлы (корзины) имеют одинаковый размер. Вся структура превращается в разделение на одинаковые участки метода SP. По этой же причине производительности ASP и EQH одинаковы.
- В методе ASP мы установили значение размера зон перекрытия равным v. Было обнаружено, что общее количество обновлений базы данных составляет 60-70 процентов от их числа в методе SP. Улучшение налицо.
- Если объекты распределены по Гауссу, количество обновлений базы данных в методе QH больше, чем в SP, и вот почему: в методе SP все корзины одинакового размера. Неважно, где находится объект, он всегда имеет одинаковую возможность пересечь границу корзины и вызвать обновление базы данных. А в QH используется Quad-дерево. Каждый листовой узел этого дерева, являющийся корзиной в базе, не может вмещать более, чем M объектов. Так как объекты расположены близко к центру рабочей области, размер корзин, близких к центру, теперь значительно меньше, чем размер внешних корзин. Так что объекты в этих корзинах с большей вероятностью будут пересекать границы, порождая обновления

базы. Ещё одной причиной является то, что преобразования корзины в QH (операции слияния и расщепления) порождают дополнительные обновления базы данных (хотя и не так много в настоящем эксперименте). По этой же причине EQH порождает больше обновлений, чем ASP.

- Преимущество QH и EQH заложено в их эффективном способе хранения. На рисунке 12 видно, что при равномерном распределении количество дисковых страниц, использованных во всех четырёх методах, почти одинаково. Однако, если первоначальное распределение неравномерно, QH и EQH используют около половины тех, что используют SP и ASP, так как в методах QH и EQH присутствует организационная часть, которая сливает корзины с малым числом объектов.
- Когда тип движения задан направленным, система вызывает много операций слияния и расщепления в QH и EQH. Это порождает дополнительные обновления базы данных. Поэтому в QH и EQH общее число обновлений базы намного больше, чем таковое при равномерном движении. Однако, хранение всё ещё эффективно.

5.6.4. Эксперимент 4: Сравнение производительности запросов

В этой части мы протестируем производительность запросов всех наших четырёх методов. Будем собирать информацию двух типов: количество дисковых страниц, к которым был получен доступ и объектов, проверенных в LP. Первый тип обычно используется для освещения производительности метода, а второй – для оценки затрат на связь.

При равномерном распределении, как на этапе индексации, производительности SP и QH почти одинаковы. Так же как у ASP и EQH. Но при распределении по Гауссу QH и EQH проверяют на 30% меньше дисковых страниц, и связь стоит дешевле вдвое, чем в SP и ASP. Это и есть то, для чего изначально создавались эти два метода. А EQH чуть хуже, чем QH, так как после расширения узла индексного дерева вероятность пересечения каждого узла с зоной запроса немного возрастает. Это увеличивает затраты на запрос. Хотя различие крайне мало.

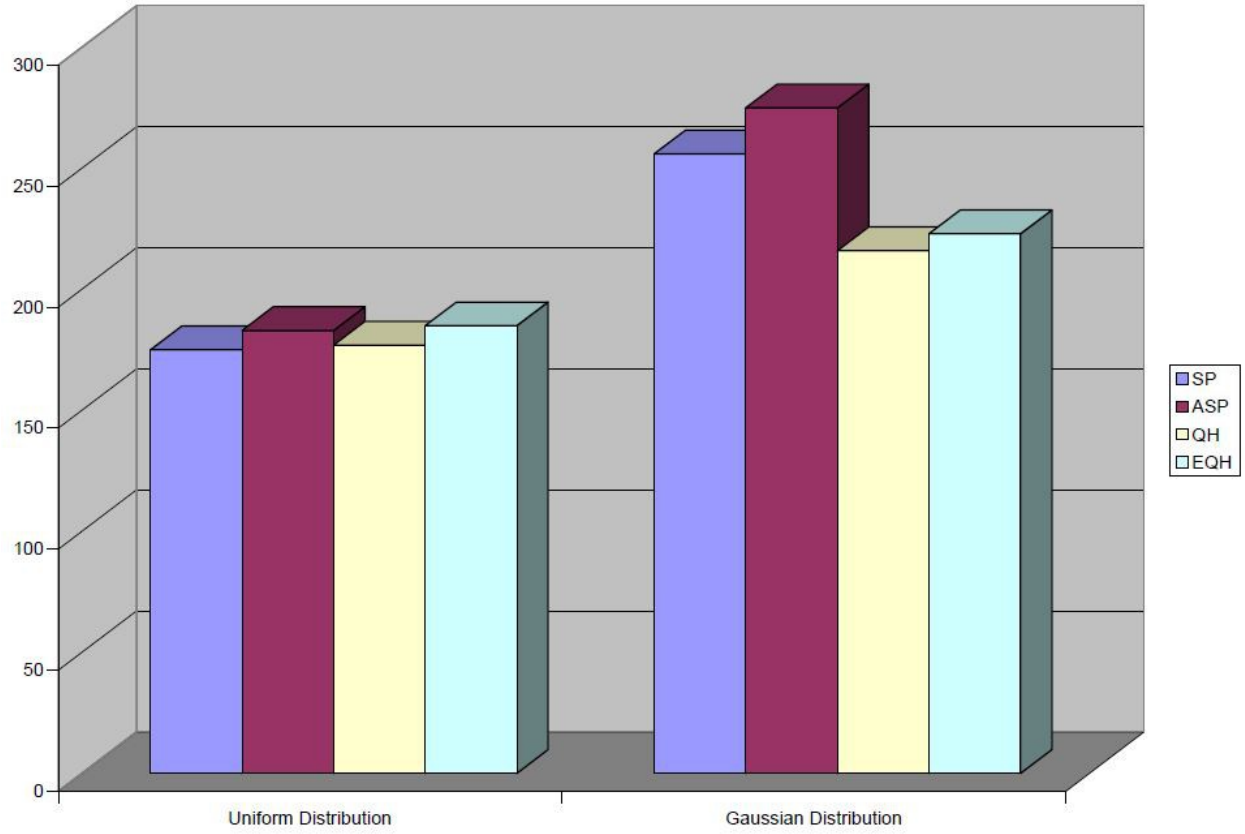

Рисунок 13. Количество проверенных дисковых страниц

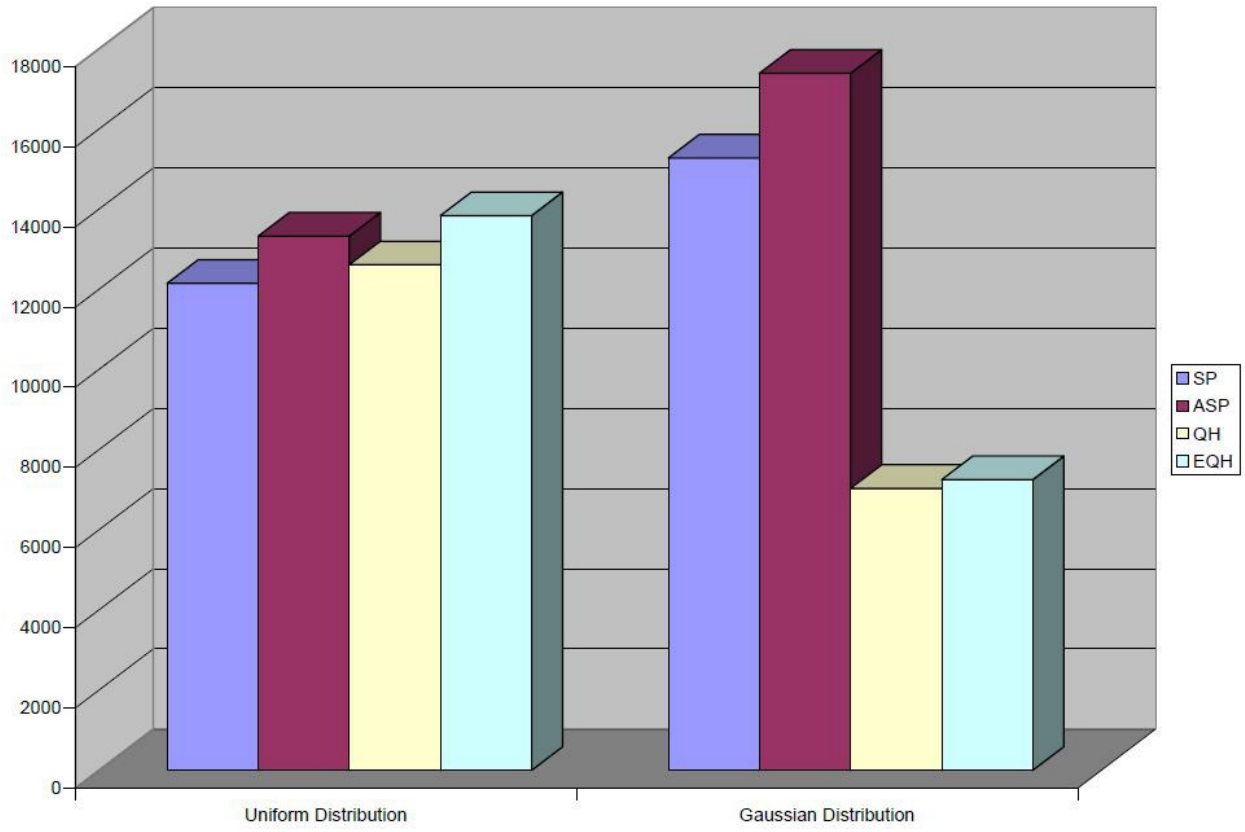

Рисунок 14. Количество объектов, проверенных LP

5.6.5. Обсуждение и выбор метода

Первое обсуждение посвятим тому, надо ли увеличивать корзины, то есть использовать ASP вместо AS или EQH вместо QH. Наш ответ – «да». Выигрыш от этого колоссален: фильтруется 30-40% запросов на обновление базы данных. И плата за это невелика: запрос становится чуть медленнее. Лучший размер для перекрытий – v, который, как правило, очень мал.

Надо ли динамически менять структуру корзины? Зависит от ситуации. Статическую структуру корзин (как в SP и ASP) легко написать. Каждая LP должна только лишь запомнить хеш-функцию, данную ей вначале. Также эта структура хорошо работает в случае равномерного распределения. Динамическая структура (в QH и EQH) более сложна в написании. Дополнительные затраты динамических корзин включают: древовидную структуру, поддерживаемую в памяти, дополнительные затраты на связь между LP и базой данных и т. д. Однако, этот метод эффективно отвечает на запросы.

Мы рекомендуем использовать метод ASP, если распределение близко к равномерному, а запросы поступают не слишком часто. В противном случае используйте метод EQH.

6. Заключение и перспективы

В данной статье мы исследовали проблемы, связанные с движущимися объектами. Главная идея, которую мы использовали, называется техникой хеширования. Она позволяет базе данных просто сохранять информацию корзины каждого объекта, вместо того чтобы запоминать многие детали. Также мы предлагаем четыре метода хеширования, построенные на новой структуре системы, созданной специально для этой техники. Эксперимент показал, что методы, основанные на технике хеширования, порождают намного меньше обновлений базы данных, чем статические методы индексации. Это делает возможным индексировать и управлять большим количеством движущихся объектов и впредь. Дальнейшие исследования включают в себя следующее. Сейчас мы можем только индексировать и обрабатывать запросы по точечным объектам. На следующем этапе мы хотим найти метод индексировать прямоугольники и объекты, меняющиеся непредсказуемо (например, для слежения за лесными пожарами и т. д.). М-Л Ло и др. предлагают смесь хеширования со статичным пространственным окружением [LR96]. Мы одолжим некоторые идеи и используем их в динамическом окружении. И, наконец, мы хотим найти больше хеш-функций, основанных на структуре нашей системы.

Благодарности: авторы хотели бы поблагодарить Самира Хуллера за его полезный совет.

Ссылки

[APR99] S. Acharya, V. Poosala, S. Ramaswamy. *Selectivity Estimation in Spatial Databases* Proc. of SIGMOD 1999.

[ArcV98] *ArcView GIS* ArcView Tracking Analyst, 1998.

[BBB87] R. Bartels, J. Beatty, B. Barsky. *An Introduction to Splines for Use in Computer Graphics & Geometric Modeling* Morgan Kaufmann Publishers, Inc., 1987.

[BBK98] Stefan Berchtold, Christian Bhon, Hans-Peter Kriegel. *The Pyramid-Technique: Toward Breaking the Curse of Dimensionality* Proc. of the ACM SIGMOD, 1998.

[BKK96] Stefan Berchtold, Daniel A. Keim, Hans-peter Kriegel. *The X-tree: An Index Structure for High-Dimensional Data* Proc. of VLDB, 1996.

[BKS+90] Bechmann N., Kriegel H.P., Schneider R., Seeger B.. *The R*-tree: An Efficient and Robust Access Method for Points and Rectangles* Proc. of the ACM SIGMOD, 1990.

[CG99] Surajit Chaudhuri, Luis Gravano. *Evaluating Top-k Selection Queries* Proc. of VLDB, 1999.

[Gut84] A. Guttman. *R-Trees, A Dynamic Index Structure for Spatial Searching* Proc. of the ACM SIGMOD, 1984.

[KGT99] G. Kollios, D Gunopulos, V. J. Tsotras. *On Indexing Mobile Objects* In Proc. of PODS, 1999.

[LJF95] Lin K, Jagadish H. V., Faloutsos C. *The TV-Tree: An Index Structure for High-Dimensional Data* Proc. of VLDB, 1995

[LR96] Ming-Ling Lo, Chinya V. Ravishankar. *Spatial Hash-Joins* Proc. of the ACM SIGMOD, 1996.

[NHS84] J. Nievergelt, H. Hinterberger, K.C. Sevik. *The Grid File, An Adpatable, Symmetric Multikey File Structure* ACM Transactions on Database Systems, Vol. 9, 1, 1984.

[NST99] M. A. Nascimento, J. R. O. Silva, Y. Theodoridi. *Evaluation of Access Structures for Discretely Moving Points* Intl. Workshop on Spatio-Temporal Database Management (STDBM'99), Edinburgh, UK, September 1999.

[PIH+96] V. Poosala, Y. E. Ioannidis, P. J. Haas, E. J. Shekita. *Improved Histograms for Selectivity Estimation of Range Predicates* Proc. of SIGMOD 1996.

[PJ99] D. Pfoser, C. S. Jensen. *Capturing the Uncertainty of Moving-Object Representations* Advances in Spatial Databases, 6th International Symposium, SSD'99, Hong Kong, China, July 20-23, 1999.

[PTJ99] D. Pfoser, Y. Theodoridis, C. S. Jensen. *Indexing Trajectories of Moving Point Objects* Chorochronos Technical Report, CH-99-3, October, 1999.

[PTJ00] D. Pfoser, Y. Theodoridis, C. S. Jensen. *Novel Approaches in Query Processing for Moving Objects* Chorochronos Technical Report, CH-00-3, February, 2000.

[Sam90] H. Samet. *The Design and Analysis of Spatial Data Structures* Addison-Wesley, Reading, MA, 1990.

[SK98] Thomas Seidl, Hans-Peter Kriegel. *Optimal Multi-Step k-Nearest Neighbor Search* Proc. of SIGMOD 1998.

[SR99] Zhexuan Song, Nick Roussopoulou. *Hashing Technique: A Framework for Indexing High Dimensional Data* Technical Report, CS-TR-4059, University of Maryland, 1999.

[SJL+99] S. Saltenis, C. S. Jensen, S. T. Leutenegger, M. A. Lopez. *Indexing the positions of Continuously Moving Objects* CHOROCHRONOS Technical Report CH-99-19, 1999.

[SWC+97] A. P. Sistla, O. Wolfson, S. Chamberlain, S. Dao. *Modeling and Querying Moving Objects* Proc. of ICDE 1997.

[TJ98] Nectaria Tryfona, Christian S. Jensen. *A component-Based Conceptual Model for Spatiotemporal Applications Design* CHOROCHRONOS project, technical report CH-98-10, 1998.

[TUW98] J. Tayeb, O. Ulusoy, O. Wolfson. *A Quadtree Based Dynamic Attribute Indexing Method* The Computer Journal, 41(3), 1998.

[WCD+98] O. Wolfson, S. Chamberlain, S. Dao, L. Jiang, G. Mendex. *Cost and Imprecision in Modeling the Position of Moving Objects* In Proc. of ICDE, 1998.

Приложение А. Алгоритмы метода хеширования с Quad-деревом

В этом приложении мы приводим псевдокод наших алгоритмов, использованных в методе хеширования с Quad-деревом.

```
addObject (MovingObject mo) {
     if (! area.isIn(mo.getLocation()))
     //если mo не в зоне этого узла
          return;
     numberOfObjects++; //обновляем количество объектов этого узла

     if (isLeaf) { //листовой узел
     /*обновить корзину в базе данных*/
          if (numberOfObjects > M && level < MAXLEVEL)
   //проверка уровня гарантирует дальнейшее нерасщепление объединения
               split();
     }
     else { //внутренний узел
          for (/*для каждого потомка c*/)
               if (c.area.isIn (mo.getLocation()))
                   c.addObject(mo);
     }
}

split() {
     /*создать четыре узла*/
     isLeaf = false; //теперь это внутренний узел
     /*создать четыре новые корзины в базе данных*/
     for (/*все объекты mo, находящиеся в этой корзине*/)
     addObject(mo); //рекурсивный вызов допускает последующее
//расщепление
     /*очистить текущие корзины*/
}
```

Листинг 1. Добавление объекта в методе хеширования с Quad-деревом

```
remove (MovingObject mo) {
     if (! area.isIn(mo.getLocation()))
          return;
     if (isLeaf) {
          /*удалить объект, находящийся в корзине*/
          for (Qtreenode = this; node != NULL; node = node.parent)
               node.numberOfObjects--;
               //обновить количественный атрибут вплоть до корня
          if (numberOfObjects < m && parent != NULL) //не корень,
//сливаем
               parent.merge();
          return;
     }
     else { //внутренний узел
     /*рекурсивный вызов функции remove для каждого потомка*/
     /*остановить, когда потомок удалит объект*/
     }
}

merge() {
     if (numberOfObjects < 3*M/4) { //проверка второго условия
     /*создать корзину в базе данных*/
     removeAllChildren(this);
     //поместить все объекты этого узла в новую корзину
     isLeaf = true;
     if (numberOfObjects < m && parent != NULL)
          parent.merge(); //рекурсивная проверка верхнего уровня
     }
}
```

Листинг 2. Удаление объекта в методе хеширования с Quad-деревом