РЕАЛИЗАЦИЯ ПАРАЛЛЕЛЬНОГО АЛГОРИТМА ХЭШ-СОЕДИНЕНИЯ ДЛЯ XML-ДАННЫХ И ЕГО ИССЛЕДОВАНИЕ НА МНОГОЯДЕРНЫХ МАШИНАХ[1]

Е.А. Иванникова
Санкт-Петербургский Государственный Университет
Россия, 199034, Санкт-Петербург, Университетская наб., 7-9
E-mail: IvannikovaE@gmail.com

Важное место в системах управления базами данных занимает обработка запросов. Сейчас для реляционной модели данных существуют достаточно эффективные реализации операций реляционной алгебры и разработано много методов выполнения сложных запросов. В области же выполнения запросов к xml-данным в связи с относительно недавним ее появлением исследования еще активно продолжаются. В данной работе реализуется одна из самых важных и самых сложных с точки зрения эффективного выполнения в СУБД операция соединения для xml-данных и исследуется возможность ее оптимизации путем параллельного выполнения на многоядерных машинах.

IMPLEMENTATION OF PARALLEL HASH-JOIN ALGORITHM FOR XML DATA ON MULTICORE COMPUTERS[2] / E.A. Ivannikova (Saint-Petersburg State University, Universitetskaya nab., 7-9, St.Petersburg, 199034, Russia). Valuable part of database management systems is handling of queries. When talking about relational model, nowadays we have pretty efficient implementations of relational algebra operations and variety of methods for complex queries execution. So as the field of XML data queries emerged more or less recently, research is still going strong for it. This work is dedicated to the implementation of the 'join' operation for XML data, which is one of the most important and the most difficult in terms of efficiency, and the possibility of it's optimization for processors with several cores

[1] Работа частично поддержана РФФИ (грант 10-07-00156)
[2] This work was partially supported by Russian Foundation for Basic Research RFBR, grant 10-07-00156.

Введение

В определенных классах задач появляется необходимость обработки слабоструктурированных данных. Наглядным примером могут служить задачи по обмену и интеграции данных, связанные с распространением Internet. Информация, доступная в Internet, очень разнообразна по своей структуре. Поэтому для ее представления требуются

слабоструктурированные модели данных. *Слабоструктурированными* называются данные, обладающие определенной структурой, но эта структура может оказаться непостоянной, недостаточно изученной или неполной. Как правило, такие данные не могут быть описаны с помощью какой-либо неизменной схемы, поэтому иногда их называют *не имеющими схемы* (schema-less) или *описывающими сами себя* (self-describing).

Способом представления таких данных может служить формат XML. Спецификация XML 1.0 была формально утверждена W3C (World Wide Web Consortium) в 1998 году. Сейчас на языке XML создается и хранится много документов как в сети Internet, так и за ее пределами. Реляционные, объектно-ориентированные и объектно-реляционные СУБД не слишком хорошо приспособлены для обработки данных такого рода. Поэтому появилась необходимость возникновения новых технологий для работы с XML-данными.

Существующие системы управления XML-данными можно разделить на два крупных класса: XML-приспособленные (XML-enabled DBMS) и XML-прирожденные (native XML DBMS). Как можно вынести из названий, первый класс составляют системы, существовавшие до возникновения потребности в работе с XML, и в дальнейшем к этому приспособленные. К ним можно отнести все большие промышленные СУБД, например, Oracle. Здесь заказчикам предлагается целый ряд возможностей, в том числе и Oracle XML DB для хранения и управления данными XML. Oracle также предлагает реализацию XQuery – стандарта выполнения запросов к XML.

Второй класс составляют системы, специально спроектированные и разработанные для управления XML-данными. Примером прирожденной XML СУБД может служить Sedna. Возможности, которые предоставляют оба класса систем, кратко рассмотрим дальше.

Возможности существующих xml-баз данных

Для работы с xml-данными в XML-приспособленных СУБД обычно предусмотрена возможность их преобразования к реляционной модели описания данных и наоборот. Таким образом, становятся доступными операции реляционной алгебры, но для больших данных время на преобразование между двумя формами становится очень существенным.

Несмотря на отсутствие таких преобразований в XML-прирожденных СУБД, обработка XML данных тоже не очень эффективна. Это связано с отсутствием хорошей алгебры и методов оптимизации. Хотя и существует множество опубликованных работ на эту тему, исследования в данной области еще не достигли желаемых результатов и активно продолжаются.

Решаемая задача

Важное место в системах управления базами данных занимает обработка запросов.
В первых коммерческих приложениях, выполненных на базе реляционной модели данных, низкая производительность при обработке запросов была большим недостатком. С тех пор было произведено множество исследований, посвященных этой проблеме, и сейчас для реляционной модели данных существуют достаточно эффективные реализации операций реляционной алгебры и разработано много методов выполнения сложных запросов. В области же выполнения запросов к xml-данным в связи с относительно недавним ее появлением исследования еще активно продолжаются. В данной работе реализуется одна из самых важных и самых сложных с точки зрения эффективного выполнения в СУБД операция соединения для xml-данных и исследуется возможность ее оптимизации путем параллельного выполнения на многоядерных машинах. Операция соединения в XQuery алгебре имеет схожий смысл с операцией соединения в реляционной алгебре и в результате предоставляет некоторую выборку из декартового произведения входных данных. Отметим, что операция соединения может служить базовой для некоторых других операций, что

подтверждает важность решаемой задачи. Примером может служить удаление дубликатов. Так, чтобы выявить наличие дубликатов в потоке данных, нужно выполнить соединение этих данных с собой.

Процесс выполнения запросов

Здесь мы приведем основные этапы обработки запросов:

1. Синтаксический анализ.

Назначение этого этапа состоит в синтаксическом анализе запроса на языке высокого уровня (как, например, SQL) и его преобразование в выражение рассматриваемой алгебры (в случае SQL – реляционной). Здесь же происходит отклонение запросов, которые некорректно сформулированы или содержат противоречивые требования.

2. Оптимизация запроса.

На данном этапе, применяя правила преобразования, оптимизатор запросов преобразует входное выражение некоторой алгебры в эквивалентное выражение, обработка которого будет заведомо более эффективной.

Здесь можно выделить два уровня: логическую оптимизацию и стоимостную оптимизацию. Первый уровень происходит из тех систем, где другая оптимизация плохая и заключается в преобразованиях, не учитывающих расчет реальной стоимости выполнения, но заведомо её улучшающих. Это, например, переписывание вложенных запросов через не вложенные, опускание выборки ближе к листовым операциям в плане выполнения и т.п.

Для конкретного запроса может существовать множество эквивалентных выражений в рассматриваемой алгебре (логических планов). Для каждой операции алгебры может существовать множество алгоритмов ее реализации (физических операций), которые успешны при различных параметрах входных данных. Для каждой операции можно вычислить ее стоимость в соответствии с некоторой моделью стоимости. Выявление оптимального плана с точки зрения такой модели стоимости и выполняется на этапе стоимостной оптимизации.

3. Выполнение запроса на этапе прогона.

В нашей работе мы займемся операцией соединения на физическом уровне и дальше рассмотрим основные алгоритмы, разработанные для выполнения операции в реляционной модели.

Алгоритмы соединения в реляционной алгебре

Существует три физических оператора соединения: соединение вложенных циклов (Nested Loops Join, NL), соединение слиянием (Merge Join, MJ) и хэш-соединение (Hash Join, HJ). Подробнее с ними можно ознакомиться в [1].

Данные алгоритмы, изначально разработанные для применения в реляционных СУБД, могут быть также применимы и в XML СУБД.

Отметим, что у каждого из алгоритмов соединения есть множество разновидностей для применения в зависимости от характеристик входных данных и параметров используемой системы. Т.к. нас в данной работе интересует эффективность выполнения операции соединения на многоядерных системах, будем использовать алгоритм хэширующего соединения в связи с тем, что проще реализовать его параллельный вариант.

Язык запросов XQuery

Для работы с XML данными рабочей группой W3C был разработан стандартный язык запросов, получивший название XQuery. Основными возможностями являются выражения пути и FLWR конструкции.

Для обозначения пути в XQuery используется синтаксис XPath, где выделяются следующие оси для определения набора узлов относительно данного узла: ось self (.) – текущий узел , child (/) – дочерние узлы, parent (..) – родительский узел, descendant-or-self (//) – узел и его потомки, и др.

Примером пути может служить такое выражение: //employee/name, которое возвращает список имен всех сотрудников организации.

Другим важным аспектом XQuery является наличие FLWR выражений, которое формируется с помощью конструкций FOR, LET, WHERE и RETURN. Данное выражение позволяет связать значения с одной или несколькими переменными и затем использовать эти переменные для формирования результата. Оператор for позволяет указанной переменной проходить в цикле по узлам, которые возвращает соответствующее ей выражение. Оператор let также связывает переменную с выражением, но не поддерживает циклический перебор. Для примера, конструкция for $a in /people/person приводит к получению нескольких привязок, каждая из которых связывает переменную $a с одним элементом person из списка people. А конструкция let $a := /people/person связывает переменную $a со списком, содержащим все элементы person.

Оператор where является необязательным в FLWR выражении и позволяет задать дополнительные ограничения на кортежи, полученные в ходе выполнения for/let операций. Например, for $a in /people/person where ($a/age > 20 and $a/age <30) ограничивает область цикла переменной $a со всех людей только на тех, которые подходят под заданный возрастной интервал.

Оператор return формирует результат выражения FLWR и должен обязательно присутствовать. Данный оператор выполняется ровно один раз для каждого кортежа связанных элементов, полученных в ходе выполнения for/let и удовлетворяющих where-условиям. Выражение for $a in /people/person where ($a/age > 20 and $a/age <30) return $a/name вернет последовательность элементов name тех людей, возраст которых удовлетворяет условиям where.

Соединение в XQuery

Покажем теперь какая связь между вышесказанным и операцией соединения. Навигационные выражения (выражения пути) используются в так называемом структурном соединении, где условием соединения является сравнение в иерархии. Т.е. оно определяет, состоят ли два узла в определенном отношении, например, parent-child.

Далее, пусть в некотором магазине есть база данных книг, которые там бывают. Для каждой книги поддерживается информация о числе ее оставшихся экземпляров. Также есть база данных авторов, где хранится некоторая информация о них, включая контакты. Для тех книг, экземпляров которых осталось мало, например, меньше трех, мы хотим сделать новый заказ, для чего, допустим, нам нужна контактная информация об их авторах. Получить нужные сведения можно, выполнив следующее FLWR выражение:

```
for $a in //author
    for $b in //book
        where $a/name = $b/author and $b/count < 3
        return ($b/title, $a/name, $a/phone, $a/e-mail)
```

Результат этого выражения представляет как раз соединение данных об авторах и книгах магазина с последующей выборкой по условию where и проекцией на элементы return. В перечисленные операции здесь вложен тот же смысл, что и в реляционном случае. Такое соединение в контексте XML называется соединением по значению.

Отметим также, что можно выполнить данный запрос в другой форме:

```
for $b in //book
    where $b/count < 3
    for $a in //author
        where $a/name = $b/author
        return ($b/title, $a/name, $a/phone, $a/e-mail).
```

Такое выражение эквивалентно первому в том смысле, что его выполнение приведет к такому же результату. Но потенциально является более эффективным, т.к. сначала выборкой уменьшается последовательность обрабатываемых дальше данных для операции соединения. Поиск таких решений и составляет предмет логической оптимизации запросов.

XML-алгебры

Производительность системы при обработке запросов во многом зависит от используемого оптимизатора. Возможности оптимизатора в свою очередь от рассматриваемой алгебры. Сейчас предложено множество различных XML-алгебр, но они не удовлетворяют требованиям (выдвинутым в работе [3]), необходимым для создания эффективного оптимизатора запросов.

Существующие алгебры можно разделить на два класса: основанные на деревьях и основанные на кортежах (кортеж-ориентированные). Первые оперируют с множествами деревьев. Их операции генерируют деревья в качестве промежуточных результатов и применяют к ним методы сопоставления с образцом. Операции кортеж-ориентированных алгебр определены над структурами, представляющими собой множество кортежей. В некоторых алгебрах в качестве элементов кортежей допускаются только атомарные значения, в некоторых последовательности атомарных значений, а в некоторых множества кортежей.

В данной работе реализуется операция соединения для последовательностей кортежей. Какую структуру здесь имеет кортеж, будет описано ниже.

Реализация

Технические характеристики: программа разработана на платформе Microsoft.NET с использованием Visual Studio 2008. Язык программирования – C#.

В программе выполняется соединение двух xml-последовательностей. Единицей в этой последовательности является кортеж (tuple), представляющий собой список элементов. Каждый элемент (element) хранит в себе информацию о своем названии (element_name) и значении (т.е. содержании - value). Также включает сведения о том, является данный элемент атомарным или сложным и содержит список своих атрибутов (attrList). Атрибут представляется парой (имя атрибута, значение атрибута). Так основные структуры данных имеют следующее описание:

```
struct attribute { string attr_name; string attr_value;}

struct element { string element_name; string value; bool isAtom;
        List<attribute> attrList; }

struct tuple { List<element> content;}
```

Приведем пример возможного кортежа (tuple):

```
<person id="person4">
    <name>Beshir Magnusson</name>
```

```
        <address postcode=”123456”>
        <street>76 Lavington St</street>
                <city>Toronto</city>
        </address>
</person>
```

Отметим, что атрибут id элемента person будет рассматриваться как атомарный элемент id первого уровня вложенности, т.е. как дочерний элемент для person. Такое представление верно только для атрибутов элементов самого внешнего уровня. Так, содержимое кортежа (tuple’s content) составляют следующие элементы:

```
<id>person4</id>                          - атомарный элемент id
<name>Beshir Magnusson</name>             - атомарный элемент name
<address postcode=”123456”>
    <street>76 Lavington St</street>          - сложный элемент address с
    <city>Toronto</city>                        атрибутом postcode
</address>
```

Отметим, что атомарный элемент у нас это, по сути, простой элемент (без вложенных xml-тегов) первого уровня вложенности. Все остальные элементы представляются сложными и их внутреннюю структуру, за исключением выделения атрибутов (как postcode), разбирать не будем.

Так, слияние производится над двумя последовательностями таких xml-кортежей. Напомним, что мы рассматриваем слияние по значению. В программе может выполняться слияние по значению атрибута самого внешнего элемента (атрибута id элемента person), по значению атомарного элемента (как name) или по атрибуту сложного первого уровня вложенности (postcode).

Использование параллельности:
Параллельное выполнение программы обеспечено с помощью встроенной библиотеки System.Thread. В ходе программы выполняется параллельный разбор входных последовательностей (один поток обрабатывает одну входную xml-последовательность).
в требуемые структуры, называемые кортежами. Далее происходит хэширование одной последовательности кортежей и полученная хэш-таблица рассылается по потокам. Кортежи из второй последовательности также распределяются между потоками и затем каждый поток выполняет фазу пробы со своими данными.

Описание и анализ экспериментов

На вход программе подаются две xml-последовательности. Для проведения экспериментов, чтобы получить входные данные, мы использовали генератор xml-документов XMark и затем путем выполнения некоторых запросов к сгенерированному документу в СУБД eXist получали xml-последовательности.

Ниже приведен ряд диаграмм. На оси абсцисс всегда указан средний размер соединяемых последовательностей в мегабайтах (Мб.), на оси ординат – время выполнения соединения. Эксперименты проводились для данных, имеющих размер в диапазоне от 0,2 Мб. до 42,2 Мб. Соединение выполнялось с использованием одного и двух ядер для одного, двух и четырех потоков. Количество используемых ядер указывалось с помощью утилиты ICE Affinity [9].

Первая диаграмма (рис. 1.1.) показывает зависимость времени выполнения соединения на одном ядре от размера входных последовательностей. Здесь присутствует два графика для одного и двух потоков. Так как на рисунке в диапазоне от 0,2 до 4,6 Мб.

графики почти сливаются, в более крупном масштабе этот фрагмент можно увидеть на рис. 1.2.

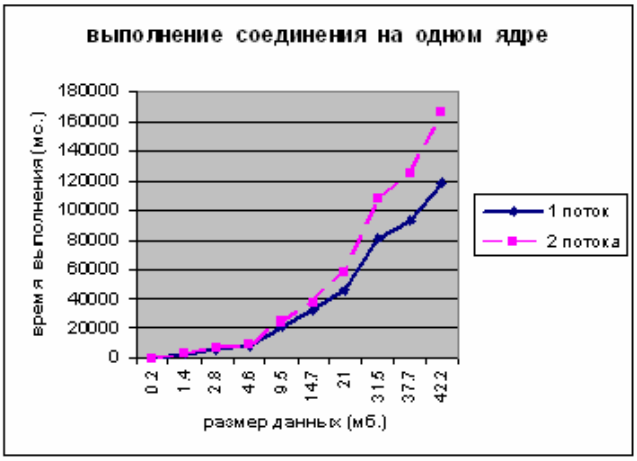

рис. 1.1.

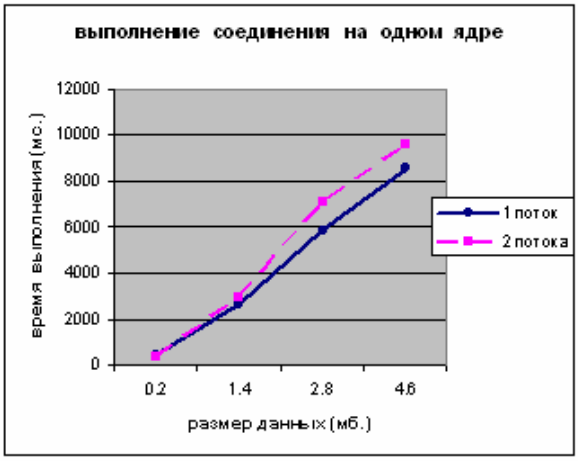

рис. 1.2.

Из этих двух диаграмм видно, что программа с одним потоком выполнения на одном ядре работает быстрее, чем с двумя потоками. Поэтому не выгодно использовать параллельный алгоритм соединения для исполнения на одном ядре. Исключение составляет только соединение небольших данных, примерно до 1 Мб, при которых время выполнения с двумя потоками меньше времени выполнения с одним потоком в среднем на 14%. Дальше посмотрим, как меняется производительность при использовании двух ядер.

На рисунках 2.1, 2.2 и 2.3 можно увидеть результаты выполнения соединения для одного и двух потоков с использованием двух ядер.

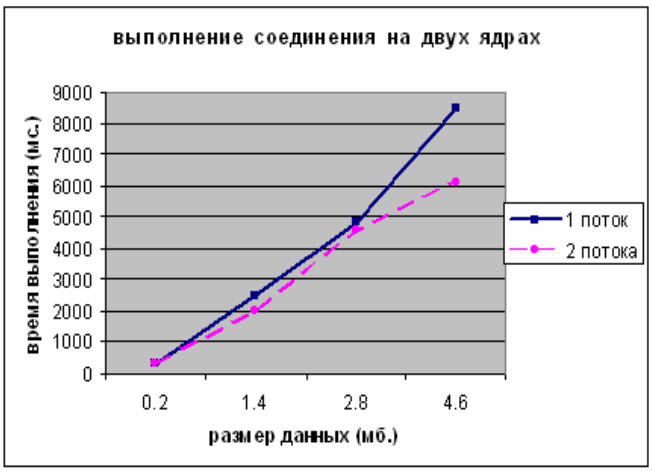

рис. 2.1.

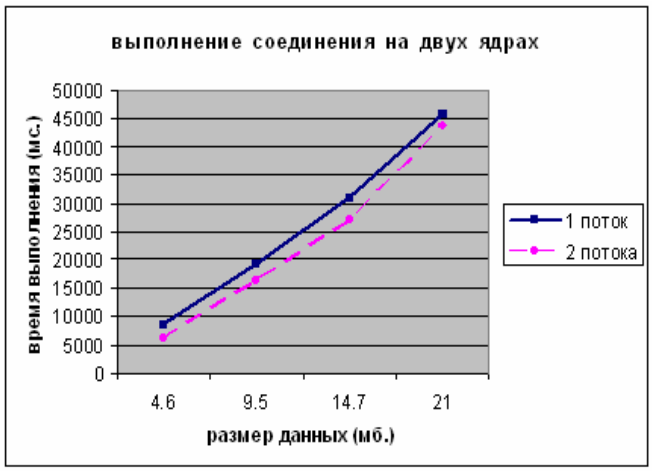

рис. 2.2.

Для данных менее 1 Мб. выигрыша во времени при использовании двух потоков вместо одного нет. Для данных из диапазона 1 – 4,6 Мб. (рис. 2.1) преимущество программы с двумя потоками заметно и состоит в уменьшении времени выполнения на 6-27%. Для данных из диапазона 4,6-21 Мб. (рис. 2.2) выигрыш во времени также сохраняется и составляет 5-13% по сравнению с однопоточной программой.

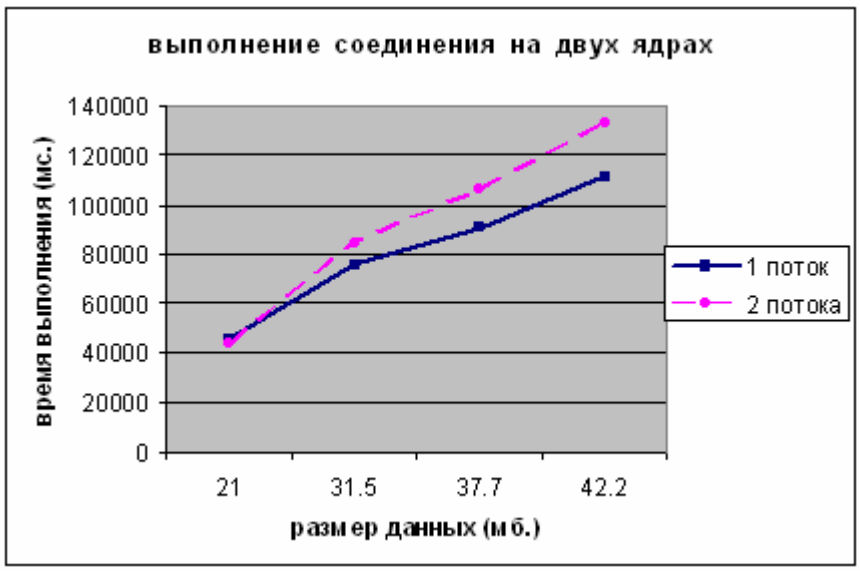

рис. 2.3.

При объеме входных данных от 21 до 42,2 Мб. соединение с одним потоком выполняется быстрее, чем с двумя потоками. При этом, при увеличении данных растет разрыв во времени.

Как показано выше для данных из диапазона 1 – 21 Мб. время выполнения соединения уменьшается при работе двух потоков на двух ядрах по сравнению с одним потоком на двух ядрах. Также время уменьшается при переходе с одним потоком от выполнения на одном ядре к двум. Чтобы оценить суммарный выигрыш в производительности, ниже приведены две диаграммы (рис. 3.1 и 3.2). На каждой диаграмме присутствуют три графика, соответствующие исполнению соединения с одним потоком на одном ядре, с одним потоком на двух ядрах и двумя потоками на двух ядрах.

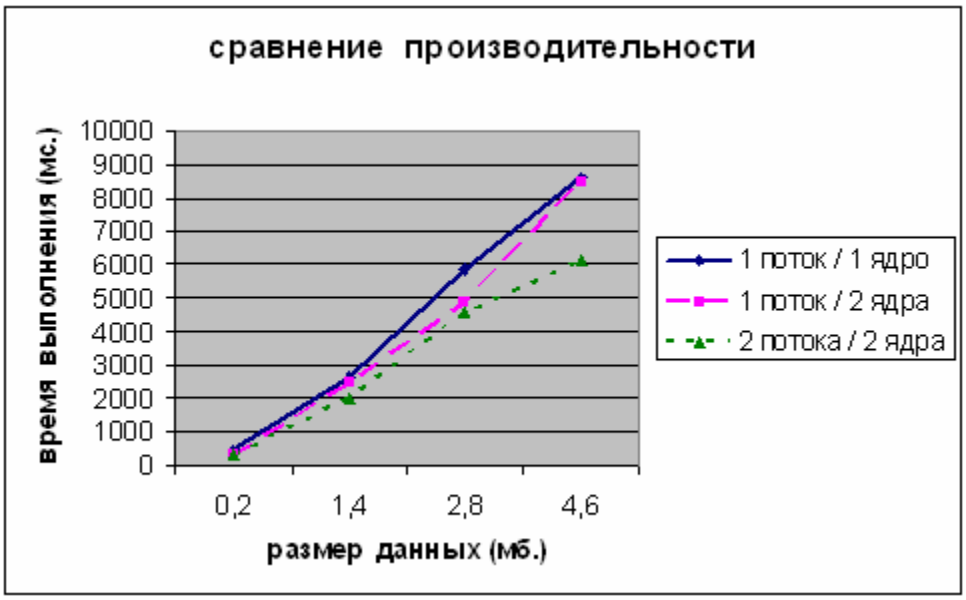

рис. 3.1

Для данных в диапазоне 0,2-4,6 Мб. (рис. 3.1) время соединения с двумя потоками на двух ядрах по сравнению с одним потоком на одном ядре уменьшается на 22-37%.

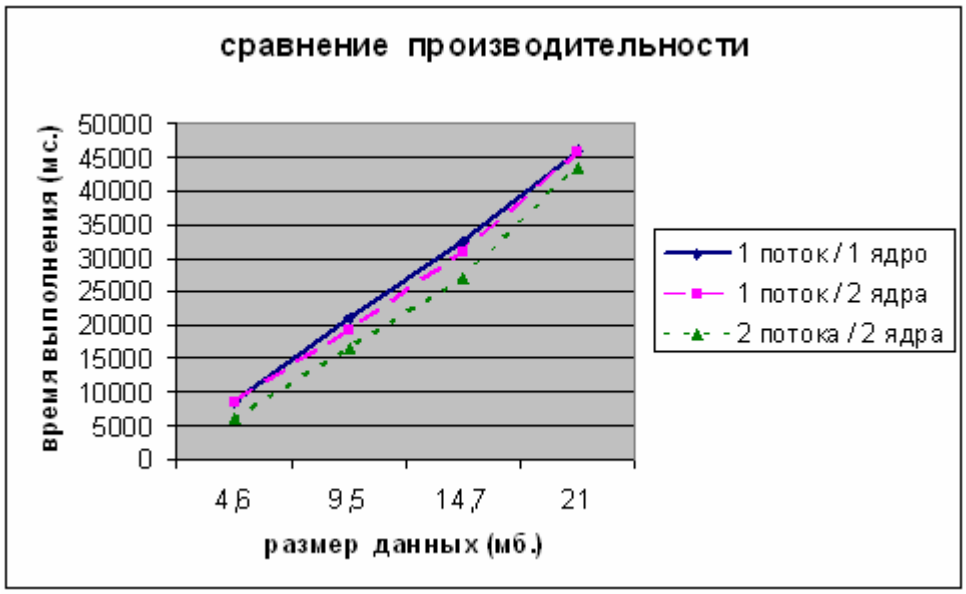

рис. 3.2

Для данных в диапазоне 4,6 – 21 Мб. (рис. 3.2) время соединения уменьшается на 5,5 – 21,5 %.

Для данных размера больше 24 Мб. (рис. 3.3) выгоднее использовать один поток на двух ядрах вместо двух. Такое снижение производительности при увеличении размера данных может быть связано с ограниченным размером кэша.

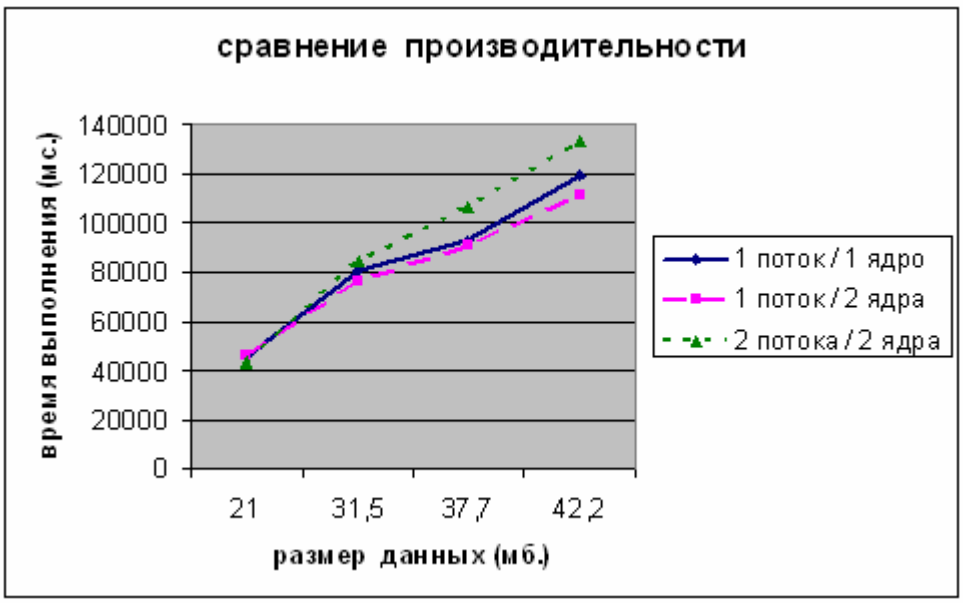

рис. 3.3

Также в ходе экспериментов было установлено, что выполнение разбора входных xml-последовательностей в требуемые структуры (последовательности кортежей) занимает 93-98% от всего времени работы программы. Т.е., непосредственно соединение с фазами хэширования и пробы занимает не больше 7% от общего времени.

Заключение

В работе был реализован параллельный алгоритм хэш-соединения для двух xml-последовательностей и проведен ряд экспериментов для оценки его производительности при исполнении на одном и двух ядрах. В ходе анализа экспериментов показано, что для данных

размера до 24 Мб. удается уменьшить время соединения на 6-37% при использовании двух потоков на двух ядрах по сравнению с одним потоком на одном ядре. Для данных большего размера удается уменьшить время на 2-7% путем использования двух потоков на одном ядре. При использовании двух потоков на двух ядрах производительность начинает снижаться, что может быть связано с ограниченным размером кэша. Кроме того, разбор входных xml-последовательностей в используемые для соединения структуры, занимает от 93% до 98% от общего времени работы программы, что очень много. Поэтому дальнейшие исследования можно проводить в направлении улучшения структур и оптимизации разбора входных xml-последовательностей.

Библиография:

1. Craig Freedman, публикации Introduction to Joins, Nested Loops Join, Merge Join, Hash Join и Parallel Hash Join. http://blogs.msdn.com/craigfr.
2. Yannis E. Ioannidis, Query Optimization.
3. Лукичёв М. С., Оптимизация запросов в слабоструктурированной модели данных.
4. David J. DeWitt и др., Implementation techniques for main memory database systems.
5. Xin Zhang и др., Honey, I Shrunk the XQuery! – An XML Algebra Optimization Approach.
6. Byron Choi и др., The XQuery Formal Semantics: A Foundation for Implementation and Optimization.
7. David J. DeWitt, Robert Gerber, Multiprocessor Hash-Based Join Algorithms.
8. W3C Recommendation, XML Path Language (Xpath) и XML Query Language (XQuery).
9. ICE Affinity, http://www.ice-graphics.com/ICEAffinity