Санкт-Петербургский Государственный Университет
Математико-механический факультет
Кафедра системного программирования

Алгоритм приближённого join’а на потоках данных

Курсовая работа студента 445 группы
Землянского Юрия Андреевича

Научный руководитель ……. профессор Б.А. Новиков

Санкт-Петербург
2011

Оглавление

Введение

В настоящее время существуют информационные системы и приложения, работающие с действительно огромными массивами данных, при анализе которых традиционные алгоритмы *data-mining*'а не всегда применимы. Для таких данных используется модель – поток. Примеры, где модель применима:

- Телекоммуникации и компьютерные сети
  - записи звонков
  - сетевой трафик
  - клики пользователей на страницах
  - логи Web-сервиса
- Бизнес
  - банковские транзакции
  - биржевые статистики
- Метеорологические данные
- и .т.п.

В данной работе разрабатывается алгоритм для *join*'а – т.е. нахождения пар элементов, удовлетворяющих какому-то предикату – в применении для модели потока данных. Как будет написано далее, алгоритм будет эффективен по времени и используемой память, но будет предоставлять только приближённые результат. Таким образом, его можно использовать и там, где точные данные не особенно важны – например, при организации информационного поиска.

Цели и задачи

Попробуем формализировать модель потока данных, с которой и будем работать в дальнейшем.

- Входные данные - набор точек в n-мерном пространстве.
- Данные поступают последовательно
- Размер входных данных заранее неизвестен и может быть произвольно большим.

Соответственно, на алгоритмы , которые будут работать с потоками, накладываются ограничения

- Алгоритмам предоставляется лишь один последовательный обход этим данных.
- Временная сложность должна быть линейной от размера входных данных
- Размер используемой памяти ограничен константой, не зависящей от размера входных данных

Алгоритм разрабатывался для одной из задач *similarity join*. Как уже писалось ранее, входными данными являются точки в пространстве, а условие для join'а двух точек $x = (x_1, \dots, x_n)$ и $y = (y_1, \dots, y_n)$ будет иметь такой вид $\sum(x_i - y_i)^2 < \varepsilon$.

Алгоритм должен находить все пары подходящих точек во входных данных. Применимо к потокам мы получаем следующий формат запросов:

- Найти точки, близкие к данной - чтобы пара удовлетворяла условию *join*'а.
- Добавить точку к множеству

Можно показать, что при таких ограничениях найти все пары точек невозможно. Любые программы, решающие эту задачу, будут находить лишь приближённое решение. Существуют несколько подходов к оцениванию качества приближённого решения (подробнее в [1], [2]). Показателями качества результата может считаться

- Процент найденных пар
- Отношение взвешенной оценки найденных пар к полному ответу. (Если в наших интересах найти не только пары точек, расстояние между которыми меньше константы, но среди них найти как можно более близкие).

Существующие решения

Для *join*'а на потоках данных существует целый ряд алгоритмов, описанных в [1], [2]. Из-за ограничений на память поддерживать все значения в хэш-таблице невозможно. Существует целый ряд алгоритмов - *Rate-Based Progressive Join* – которые отличаются политикой сохранения точек в памяти. При этом учитываются такие параметры входных данных:

- Частота появления элемента
- Количество найденных пар с этим элементов
- Время последнего появления элемента

Качество результата при применении той или иной стратегии зависит от конкретных данных.

Общая идея алгоритма

Для задачи *similarity join* в нашей формулировке, но без ограничений на представление входных данных в виде потоков, разработано много эффективных методов и алгоритмов. Наиболее популярные алгоритмы используют иерархические структуры в виде дерева, где в листах находятся точки множества, а вершины хранят какую-то общую информацию о области, в которой лежат точки из поддерева. За счёт этого поиск подходящих пар для *join*'а можно сильно ускорить. Краткий обзор этих методов содержится в статье [3]. Типичные структуры, использующиеся в этих алгоритмах - R-trees [4], R*-trees [5], M-trees [6] and Filter-trees [7].
Однако, сразу применить эти алгоритмы к потокам не получается в силу ограничений на время и используемую память.

Основная идея этой работы состоит в том, чтобы строить похожие иерархии онлайн - на основе существующих алгоритмов кластеризации, а затем эффективно осуществлять поиск близких точек, пользуясь полученной структурой.

Алгоритмы кластеризации

Существуют алгоритмы кластеризации, работающие эффективно на потоках данных. Обзор таких алгоритмов есть в [11], [12]. Наиболее популярные – это, например, CluStream [8], HPStream [9], STREAM [10].

В качестве дальнейших исследований можно попробовать использовать другие алгоритмы, но в нашей работе мы стали использовать BIRCH.

Алгоритм BIRCH относиться к иерархическим алгоритмам кластеризации. Структура, которая строиться во время алгоритма и хранит информацию о обработанных точках, называется *Cluster-Feature Tree*. Подробнее про сам алгоритм можно прочитать в [13]. Но так как нам потребуется несколько модифицировать алгоритм под нашу задачу, мы опишем общую схему его работы.

Cluster-Feature Tree представляет собой дерево, где

- Каждый лист соответствует какому-то множеству точек
- Внутренняя вершина соответствует объединению множеств точек своих детей
- В каждой вершине, которой соответствует множество точек P, храниться *Cluster-Feature Vector* = $< |P|, SumX, SumX2 >$, где
    - $SumX_i = \sum_{x \in P} x_i$
    - $SumX2_i = \sum_{x \in P} x_i^2$
- Количество детей у одной вершины - не более, чем B.
- Размер занимаемой памяти ограничен константой M.

При добавлении новой вершины мы запускаем рекурсивную процедуру, которая

- Каждый раз выбирает ближайшую вершину (расстояния до множества полагается за расстояние до центра масс) и запускается от неё
- Если текущая вершина - лист и расстояние до центра масс меньше константы T, то добавляем точку к вершине. Иначе создаём новый лист из нашей вершины.
- Если у какой-то вершины оказалось слишком много детей – разбиваем всех её детей на два множества, каждое из которых будем представлять новой вершиной. Подвешиваем эти две вершины в качестве новых двух детей.
- Если объём занимаемой памяти превышает M, то увеличиваем T, и строим новое дерево, последовательно добавляя в него листья из старого.

Теперь изменим структуру дерева так, что её можно было пользоваться в алгоритме join’а.

- Прежде всего, нам нужно хранить сами точки, чтобы можно было формировать ответ из пар похожих точек. Поэтому теперь листья хранят не только *Cluster-Feature Vector*, но и список всех точек.
- В случае если дерево занимает слишком много памяти нам нужно удалить какие-то поддеревья. Стратегии удаления можно выбирать из использующихся в алгоритмах точного join’а на потоках.

Алгоритм приближённого join'а

Во время работы нашей программы мы будем поддерживать структуру модифицированного *Cluster-Feature Tree*. И обработка запроса "добавить точку к множеству" состоит в добавлении точки в *Cluster-Feature Tree*.

Теперь перейдём к тому, как искать точки близкие к нашей, подходящие под условие join'а.

Прежде всего, заметим, что на основе статистики, которую мы храним в каждой вершине, можно подсчитать "радиус" соответствующего множества точек – то, насколько сильно точки разбросаны вокруг центра масс. В нашем случае, за радиус мы брали корень из дисперсии для набора точек.

На вход алгоритм получает точку $point$ и максимальный порог для расстояния $treshold$. Вызывает от корня дерева рекурсивную процедуру, которая перебирает каждого из потомков вершины и

- Если круг с радиусом $treshold$ и центром в точке $point$ полностью лежит в круге для множества точек у вершины, то в ответ надо выдать все точки из этого поддерева
- Если круги пересекаются, и их "пересечение" больше константы $magic$,то запускаем процедуру от этого потомка. ("Пересечение" оценивалось как отношение $\left(\frac{\text{radius+treshold}-\text{distance}}{\text{radius}}\right)$, а константа $magic$ была параметром алгоритма.

Реализация алгоритма на псевдокоде:

```
FindPairs(node, point, treshold){
     result := EmptySet()

     If node Is Leaf Then
          For point2 In Points(node) Do
               If Distance(point2, point) <= threshold Then
                    result.Add(Pair(point2, point))
               End
          End
          Return result;
     End

     For child In Children(node)
          distance := Distance(point, node)
          radius := Radius(node)

          If radius + distance <= threshold Then
```

```
                result.Add(GetAllPoints(node));
            Else If radius + treshold <= distance Then
                // do nothing
            Else If ((radius+treshold-distance)/radius > magic)
            Then
                FindPairs(child, point, treshold);
            End
        End
        Return result;
End;
```

Эксперименты

Алгоритм был реализован на языке JAVA и выложен в открытый репозиторий Google Code [14].

В качестве входных данных использовался генератор входных данных “Spatial Data Generator”. [15], а так же данные с [16]

Качество алгоритма оценивалось как процент верно найденных пар к их действительному числу.

Ниже приведена список результатов. Для каждого отдельного набора данных алгоритм подстраивался необходимым образом.

Размер файла	Используемая память	Полнота
0.5M	0.5M	69%
3M	1M	43%
10M	5M	21%
50M	5M	10%

Выводы

В итоге нами был предложен алгоритм, реализующий similarity join в условиях потоковых входных данных и ограничения на используемую память.

Список использованной литературы

[1] Junyi Xie and Jun Yang. "A Survey of Join Processing in Data Streams"
[2] Ahmed K. Elmagarmid, "Data Streams : Models and Algorithms"
[3] Brent Bryan, Frederick Eberhardt, Christos Faloutsos, "Compact Similarity Joins"
[4] A. Guttman, “R-trees: A dynamic index structure for spatial searching,”
[5] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, “The r*- tree: An efficient and robust access method for points and rectangles,”
[6] P. Ciaccia, M. Patella, and P. Zezula, “M-tree: An efficient access method for similarity search in metric spaces,”
[7] K. C. Sevcik and N. Koudas, “Filter trees for managing spatial data over a range of size granularities”
[8] Charu C. Aggarwal, Jiawei Han, Jianyong Wang, Philip S. Yu, A Framework for Clustering Evolving Data Streams”
[9] Charu C. Aggarwal, Jiawei Han, Jianyong Wang, Philip S. Yu, A Framework for Projected Clustering of High
Dimensional Data Streams
[10] Liadan O’Callaghan, Nina Mishra, Adam Meyerson, Sudipto Guha, “Streaming-Data Algorithms for High-Quality Clustering”
[11] Sudipto Guha_ Adam Meyerson y Nina Mishra z Rajeev Motwani x
Liadan O’Callaghan, “Clustering Data Streams: Theory and Practice”
[12] Alireza Rezaei Mahdiraji, “Clustering data stream: A survey of
algorithms”
[13] Tian Zhang, Raghu Ramakrishnan, Miron Livny, “BIRCH: An Efficient Data Clustering Method for Very Large
Databases”
[14] Birch-Join // Репозиторий проекта “birch-join” на GoogleCode
URL : http://code.google.com/p/birch-join/
[15] rtreeportal.org // Spatial Data Set Generator
URL :
http://www.rtreeportal.org/index.php?Itemid=41&id=28&option=com_content&task=view
[16] dbkgroup.org // Наборы тестов и генераторов для spatial алгоритмов
URL : http://dbkgroup.org/handl/generators