УДК 004.421.2:519.178

И. В. Поляков, **А. А. Чеповский**, **А. М. Чеповский**

Национальный исследовательский университет
Высшая школа экономики
ул. Мясницкая, 20, Москва, 101000, Россия

E-mail: achepovskiy@hse.ru

ХРАНЕНИЕ И ОБРАБОТКА ГРАФА СОЦИАЛЬНЫХ СЕТЕЙ

Представлена специализированная структура данных, предназначенная для хранения и выполнения различных операций с графами социальных сетей больших объемов. Предложенная структура хранения ориентирована на поддержку операций пополнения и выгрузки подграфов и поиска кратчайших путей между двумя группами вершин.

Ключевые слова: хранилище графов, алгоритмы теории графов, большие данные.

Введение

Специализированные базы данных для хранения и обработки графов получили широкое развитие в последние годы. Они имеют обширную область применения от химических и биологических задач до задач хранения и обработки графов социальных сетей. Их преимуществом по сравнению с реляционными базами данных является поддержка более сложных операций в рамках моделей исследуемых предметных областей.

Каждая такая база данных имеет свои особенности и свой язык запросов. Сравнительную характеристику наиболее популярных баз данных для хранения и обработки графов можно найти в [1–4]. Отличия между ними выражаются главным образом в оптимизации поддержки различных наборов алгоритмов обработки графов. Так, *Neo4j* ориентирована на максимально быстрый обход графа и поиск путей, а *HyperGraphDB* – база с поддержкой хранения гиперграфов, *Sones* поддерживает хранение взвешенных графов и гиперграфов.

В силу того что большинство баз данных для хранения и обработки графов имеют широкую специализацию и поддерживают большой диапазон различных операций, для конкретных предметных областей и задач могут быть построены более эффективные решения.

Постановка задачи

Обозначим множество всех потенциальных узлов графа V_m. Пусть $E_m = V_m \times V_m$ – множество всех возможных ориентированных ребер. Под графом будем понимать произвольную пару множеств $\{V, E\}$, $V \subset V_m$, $E \subset E_m$. На практике каждому хранимому графу сопоставляется также некоторая таблица атрибутов. Атрибуты вершины ν будем обозначать через $A_i(\nu)$, $i = 1,...,M$, атрибуты ребра e – $A_i(e)$, $i = 1,...,M$. Предполагается, что каждая функция A_i отображает множества всех возможных вершин и ребер в пространство значений X_i соответствующего атрибута. Все такие пространства должны содержать специальный эле-

Поляков И. В., Чеповский А. А., Чеповский А. М. Хранение и обработка графа социальных сетей // Вестн. Новосиб. гос. ун-та. Серия: Информационные технологии. 2013. Т. 11, вып. 4. С. 77–83.

ISSN 1818-7900. Вестник НГУ. Серия: Информационные технологии. 2013. Том 11, выпуск 4

мент *NULL*. Предполагается, что $A_i(\nu) = NULL$, если для данной вершины *i*-й атрибут не определен. Аналогичное соглашение выполняется для ребер.

На каждом пространстве X_i определена некоторая хэш-функция *H*, отображающая элементы данного пространства на множество натуральных чисел. Отметим, что данная таблица не является фиксированной и может динамически изменяться на протяжении всего времени существования графа.

Для каждой вершины ν под $E(\nu)$ будем понимать все ребра, инцидентные данной вершине. Для каждой связи l через $\nu_1(l)$, $\nu_2(l)$ обозначим те вершины, которые она соединяет.

Сконструируем такую модель хранения графа, которая ориентирована главным образом на поддержку операций пополнения, объединения и пересечения графов, а также поиска кратчайших путей между двумя группами вершин. Структуры данных должны быть рассчитаны на работу с графами, содержащими до 100 миллионов вершин и нескольких миллиардов связей, поэтому хранение графа в виде матрицы смежности не представляется возможным. За основу взят способ хранения графа в виде списков смежности.

Структура хранилища

В качестве базовой структуры хранилища используется специализированная файловая система, которая реализуется внутри некоторого файла стандартной файловой системы компьютера, поэтому под логическим файлом будем понимать файл, принадлежащий данной специализированной системе. Предполагается, что каждый логический файл состоит из некоторых записей переменной длины.

Поддерживаются следующие операции:

- адресация логических файлов целыми числами;
- добавление записи в конец логического файла;
- удаление логического файла целиком;
- чтение всех записей логического файла с произвольной позиции в прямом и обратном направлениях.

Таким образом, система состоит из файлов, адресуемых числами $1,...,N$, типичное значение для *N* – порядка 100 000. При этом *i*-й файл состоит из блоков $\{B_{ij}\}$, $j = 1,...,C_i$ фиксированной длины, размером порядка 300 Кб, объединенных в двунаправленный список. В оперативной памяти для каждого логического файла хранится так называемая карта блоков $\{M_{ij}\}$, $j = 1,...,C_i$, представляющая собой двунаправленный список, который хранит смещения блоков файла. Также элемент M_{ij} этой карты содержит в себе битовую маску фиксированной длины *L* (обычно порядка 32 000), предназначенную для сохранения значений некоторых выбранных хэш-функций, вычисляемых от каждой записи *r*, добавленной в блок B_{ij}. При этом в битовой маске устанавливаются биты с номерами $H(A_i(r))\%L$. Это позволяет более эффективно выполнять операцию чтения файла с условием фильтрации по атрибутам, пропуская некоторые блоки, в которых нет записей, содержащих данные атрибуты.

В зависимости от режима использования в памяти может осуществляться буферизация данных, добавляемых в конец файла. Это позволяет снижать нагрузку на дисковую подсистему в процессе активной записи данных в конец файла. При необходимости данные в файлах могут сжиматься одним из стандартных алгоритмов. Сжатие происходит при заполнении буфера. Единицей сжатия в таком случае является один заполненный буфер. Главные преимущества от использования собственной файловой системы состоят в следующем:

1) отсутствие системных вызовов при работе с собственной файловой системой;

2) в традиционных файловых системах имеется существенное ограничение на количество одновременно открытых файлов, которое отсутствует при использовании собственной файловой системы;

3) управление буферизацией и хранением данных прозрачно и не зависит от особенностей операционной системы;

4) отсутствие сложной системы адресации (структуры каталогов) и лишних механизмов (права доступа) приводит к отсутствию дополнительных накладных расходов на организацию хранения данных.

Идентификация вершин и хранение списков смежности

Все добавляемые в граф вершины идентифицируются возрастающими натуральными числами. Атрибуты вершин складываются в некоторый файл по порядку. Если атрибуты очередной вершины требуют для своего хранения S байт, то для них резервируется $1,2 \cdot S + C$ байт, где C – параметр алгоритма, который выбирается в зависимости от параметров атрибутов того графа, который планируется хранить. Наличие дополнительного места позволяет в дальнейшем изменять атрибуты некоторой вершины, в том числе путем добавления новых атрибутов. Если свободного места недостаточно, для модифицируемой вершины выделяется еще одна запись, и обе записи объединяются в двунаправленный список. В дальнейшем возможно выделение дополнительных записей. При этом возникает некоторая фрагментация, которая может быть устранена путем перестроения файла атрибутов вершин, что потребует полной остановки работы хранилища.

Для каждой вершины ν в оперативной памяти выделяется дескриптор $D(\nu)$ фиксированного размера, содержащий указатель на первую запись в файле атрибутов, соответствующую данной вершине.

Каждой вершине ν сопоставлен один логический файл $F(\nu)$, в котором хранятся записи, описывающие связи этой вершины. При этом один логический файл может быть сопоставлен нескольким вершинам, т. е. равенство $F(\nu_1) = F(\nu_2)$ возможно для двух несовпадающих вершин ν_1 и ν_2. Номер данного файла хранится в дескрипторе вершины. Также в дескрипторе хранятся счетчик количества связей данной вершины и счетчик количества выполненных операций кластеризации для данной вершины. В дескрипторе каждой вершины хранится количество различных вершин, смежных с ней, привязанных к тому же логическому файлу. Это значение пересчитывается в момент проведения очередной процедуры кластеризации для данной вершины.

Каждая связь l, добавляемая в хранилище, формирует некоторую запись, которая содержит идентификаторы двух соединяемых вершин $\nu_1(l)$, $\nu_2(l)$, два бита, характеризующие ее направленность (связь может быть ненаправленной), а также все атрибуты связи $A_i(l)$. Сформированная запись записывается впоследствии в файлы с номерами $F(\nu_1(l))$, $F(\nu_2(l))$. Кроме того, каждой связи присваивается некоторый уникальный идентификатор $\mathrm{id}(l)$, который также добавляется в виде отдельного поля к добавляемой записи.

После добавления некоторой связи l может возникнуть необходимость в ее полном удалении или изменении содержащихся в ней атрибутов. Соответствующие операции реализованы в хранилище. В случае изменения атрибутов формируется запись, описывающая новую версию связи, при этом новая запись содержит тот же самый идентификатор связи $\mathrm{id}(l)$. Затем эта запись добавляется в файлы $F(\nu_1(l))$, $F(\nu_2(l))$. В случае полного удаления связи формируется служебная запись с флагом удаления, с идентификатором $\mathrm{id}(l)$, которая также добавляется в файлы $F(\nu_1(l))$, $F(\nu_2(l))$.

Таким образом, чтение списка смежности вершины ν требует чтения логического файла $F(\nu)$ в обратном порядке с последующей фильтрацией всех прочитанных записей по полю $\nu_1(l)$. Кроме того, для одного и того же идентификатора связи в полученном списке может

быть несколько записей. Это происходит в случае выполнения операций изменения или удаления связей. Если первой идет служебная связь с флагом удаления, то все записи с заданным идентификатором должны быть выброшены из финального списка смежности, в противном случае в финальный список смежности следует поместить только самую первую запись (при чтении в обратном порядке).

Если необходимо привязать некоторую вершину к другому логическому файлу, следует сначала скопировать список смежности в другой логический файл, предварительно прочитав его. Затем необходимо удалить все записи о данной вершине из старого логического файла. Это можно осуществить путем добавления специальной служебной записи, содержащей флаг удаления всех записей данной вершины, записанных ранее в логический файл. После этого нужно прописать другой номер логического файла в дескрипторе вершины $D(\nu)$. Такой способ перемещения не требует чрезмерно больших затрат, однако приводит к некоторому дублированию данных, которое в дальнейшем может быть устранено проведением процедуры дефрагментации того логического файла, откуда был удален список смежности.

Индексы по атрибутам вершин

Для более быстрого выполнения операций поиска вершин по конкретным значениям атрибутов в хранилище присутствует специализированный индекс по атрибутам вершин. Данный индекс также использует собственную копию специализированной файловой системы, описанной выше. Система содержит K логических файлов. При добавлении вершины ν с атрибутами $A_i(\nu)$, $i=1,...,M$, для i-го атрибута формируется запись, содержащая в качестве полей:

1) значение хэш-функции на данном атрибуте $H(A_i(\nu))$;
2) номер атрибута i;
3) идентификатор вершины $\mathrm{id}(\nu)$.

Данная запись заносится в логический файл с номером $H(A_i(\nu))\%K$. В дальнейшем это позволяет более эффективно искать вершины, у которых значение i-го атрибута равно A. Для этого необходимо загрузить логический файл с номером $H(A)\%K$ и отфильтровать полученное множество записей по первым двум полям. Извлекая идентификаторы вершин из полученных записей, можно сформировать множество вершин, заведомо содержащее все искомые. Однако из-за коллизий хэш-функции там могут оказаться вершины, для которых значение i-го атрибута отличается от A. Для фильтрации таких вершин необходимо для каждой полученной вершины загрузить ее атрибуты из файла атрибутов вершин. Это потребует одной дисковой операции на каждую проверяемую вершину, что не всегда является приемлемым. Альтернативой данному подходу является хранение непосредственных значений атрибутов вместо значений хэш-функции в записях индекса в качестве первого поля. Данный способ не требует фильтрации, однако он способен увеличить объем индекса в несколько раз.

Операции модификации атрибутов уже добавленной вершины требуют внесения соответствующих изменений в индекс.

В случае добавления новых атрибутов следует добавить в индекс дополнительную запись на каждый атрибут, при удалении некоторого атрибута необходимо добавить в индекс запись с флагом удаления. При последующем чтении записей, соответствующих некоторому атрибуту, следует читать полученный список с конца. Встретив запись об удалении некоторого атрибута у вершины с некоторым идентификатором, мы должны игнорировать предыдущие записи о его наличии у вершины с тем же идентификатором. Операция изменения атрибута сводится к последовательному применению операций удаления и добавления атрибута.

Поиск путей и кластеризация списков смежности

Рассмотрим задачу нахождения пути между множествами вершин S_1 и S_2 за минимальное время. При этом желательно найти путь как можно меньшей длины. Обозначим через

V_1 множество вершин, ассоциированных с *i*-м логическим файлом. Количество связей между вершинами из данного множества намного меньше общего числа связей в графе (при количестве логических файлов, превышающем несколько сотен тысяч). Поэтому разумно требовать, чтобы ограничение графа на множество V_i могло быть полностью размещено в оперативной памяти. В противном случае логический файл можно разбить на несколько файлов меньшего размера. Отметим также, что такого рода графы могут быть построены за один проход по логическому файлу.

Предлагается следующий алгоритм, решающий поставленную задачу поиска путей. На каждом шаге предлагаемого алгоритма образуются множества S_1^n, S_2^n, а также некоторые структуры данных D_1^n, D_2^n, сопоставляющие каждой вершине $s \in S_j^n$ некоторую вершину предок $D_j^n(s)$. В качестве такого рода структур могут быть использованы обычные массивы. При этом $S_j^1 = S_j$ и $D_j^1(s) = s$, что означает, что все вершины из S_j^1 не имеют предка.

Опишем *n*-й шаг алгоритма.

1. В случае если множества S_1^n, S_2^n пересекаются – кратчайший путь найден, так как при наличии общей точки с помощью структур D_1^n, D_2^n можно построить путь от нее до множеств S_1^n, S_2^n соответственно.

2. Проверяем существование таких *i*, для которых $S_1^n \bigcap V_i$ и $S_2^n \bigcap V_i$ одновременно не пусты, и при этом одно из множеств $S_1^n \bigcap V_i$, $S_2^n \bigcap V_i$ должно быть строго больше, чем $S_1^{n-1} \bigcap V_i$ и $S_2^{n-1} \bigcap V_i$ соответственно.

3. Для каждого найденного *i* загружаем *i*-й логический файл в память, строим по нему граф G_i и ищем в нем кратчайшие пути между множествами $S_1^n \bigcap V_i$, $S_2^n \bigcap V_i$ с помощью обычного алгоритма поиска в ширину. Каждый такой путь очевидным образом дает некоторый путь между множествами S_1^n, S_2^n в исходном графе, который с помощью структур D_1^n, D_2^n продолжается до некоторого пути между S_1 и S_2.

Если такие пути существуют, то добавляем их в список найденных путей. Полученные таким образом пути необязательно являются кратчайшими, поэтому алгоритм не может быть остановлен в случае их успешного обнаружения.

4. Добавляем в множество S_1^{n+1} все вершины из множества S_1^n и смежные с ними вершины. Для каждой вершины, не содержащейся в S_1^n, запоминаем ту вершину, из которой она была достигнута в структуре D_1^{n+1}. Таким же образом строим множество S_2^{n+1}, после чего снова переходим к пункту 1.

Отметим, что по мере нахождения некоторого набора путей они могут предоставляться пользователю, который может принять решение о прекращении дальнейшей работы алгоритма.

Точные оценки количества операций, требуемых данным алгоритмом, существенным образом зависят от того, насколько плотно связанными являются графы G_i. Идеальным вариантом являются полные графы, в которых каждая пара вершин связана между собой. Отметим, что полноту графов G_i можно существенно повысить на этапе построения графа с помощью некоторого процесса, называемого далее кластеризацией.

Кластеризация проводится на этапе построения графа. В процессе нее происходит перемещение списка смежности данной вершины в другой логический файл. Целью данной операции является увеличение связности графов G_i.

При добавлении вершины к ней автоматически привязывается наименее загруженный логический файл. Это происходит потому, что без наличия каких-либо связей невозможно определить тот файл, с вершинами которого данная вершина имеет максимальное число связей. В дальнейшем по мере накопления связей происходит перемещение всего списка смежности в тот логический файл, с которым у данной вершины имеется максимальное число связей.

Для определения моментов переноса определяются некоторые пороговые значения $a_N = 2^N$. При достижении счетчиком количества данной вершины очередного порогового значения список смежности этой вершины перемещается в наиболее подходящий логический файл. Для этого он должен быть полностью прочитан. В процессе чтения вычисляется количество уникальных вершин в каждом логическом файле, смежным с данным. Выбирается файл с максимальным значением этого числа. Именно в этот файл и перемещается список смежностей вершины. Счетчик количества уникальных связей для этого логического файла запоминается в дескрипторе вершины.

Операции объединения и пересечения

Необходимо объединить хранилища $G_1 = (V_1, E_1)$ и $G_2 = (V_2, E_2)$ в результирующее хранилище $G_3 = (V_3, E_3)$. При этом задаются некоторые предикаты P_1, P_2 на множестве пар $V_1 \times V_2$ и $E_1 \times E_2$, определяющие условия, при выполнении которых должны отождествляться вершины и связи из разных графов. При отождествлении вершин и связей, происходит объединение списков атрибутов с последующим устранением дубликатов.

Установим следующие два требования к хранилищам, необходимые для успешного выполнения операции объединения:

1) списки вершин обоих хранилищ со всеми их атрибутами должны одновременно помещаться в оперативной памяти;

2) объединенное хранилище не должно содержать вершины, все связи которой вместе с атрибутами не помещались бы в оперативную память.

Алгоритм объединения двух хранилищ включает несколько этапов.

1. Сначала необходимо объединить множества вершин V_1 и V_2 и поместить их в результирующее хранилище. Для этого оба списка помещаются в оперативную память, после чего производится их объединение в единый список с помощью предиката P_1. При объединении нескольких вершин в одну объединяются списки их атрибутов. Список $V_1 \bigcup V_2$ заносится в результирующее хранилище. При этом формируются два массива I_1, I_2, определяющие соответствие между индексами вершин объединяемых хранилищ и их индексами в результирующем хранилище.

2. Необходимо прочитать все логические файлы каждого из объединяемых хранилищ и добавить каждую прочитанную запись о наличии связи в новое хранилище. Для каждой прочитанной связи будут известны индексы ее вершин в соответствующем хранилище G_i. Индексы тех вершин, в которые они перешли, могут быть вычислены с помощью массивов I_1, I_2. Таким образом, эта связь может быть добавлена в хранилище G_3.

3. Производится удаление дубликатов связей в хранилище G_3. Для этого необходимо по очереди прочитать в оперативную память каждый логический файл хранилища G_3, отсортировать все связи по полям $\nu_1(l)$, $\nu_2(l)$. После этого следует попарно сравнить все связи с совпадающими значениями данных полей с помощью предиката P_2. Для каждой группы совпавших связей следует оставить только один экземпляр.

После данной процедуры необходимо перезаписать логический файл. Если какой-либо логический файл не помещается в оперативную память, он разбивается на несколько файлов меньшего размера.

Пересечение графов выполняется схожим с объединением способом. За исключением того, что в список вершин результирующего хранилища попадают только те вершины, которые присутствуют в обоих хранилищах G_1 и G_2. При этом в хранилище G_3 попадают только связи l, для которых вершины $\nu_1(l)$, $\nu_2(l)$ попали в G_3. Таким образом, после пересечения графов по вершинам происходит их объединение по связям с последующим устранением дубликатов. Требования к графам, участвующим в данной операции, аналогичны требованиям операции объединения.

Заключение

Предложена модель хранения графов и необходимые для ее реализации структуры данных и базовые алгоритмы. Эта модель ориентирована на хранение и операции с графами, размеры которых превышают объемы оперативной памяти. При этом количество хранимых ребер потенциально ограничено лишь объемом наличного дискового пространства.

Рассматриваемый объем данных делает невозможным использование стандартного алгоритма поиска в ширину для нахождения кратчайшего пути.

Более сложные алгоритмы предполагают создание и поддержание дополнительных структур данных, размер которых потенциально может превосходить размер самого графа в несколько раз.

Предложенная схема кластеризации списков смежности не использует дополнительных структур данных, позволяя при этом значительно сократить количество дисковых операций, требуемых для нахождения какого-либо пути между двумя группами вершин, по сравнению с тривиальным алгоритмом поиска в ширину.

Данные выводы подтверждены вычислительными экспериментами авторов на графах, имеющих порядка 100 миллионов вершин и нескольких миллиардов связей.

Список литературы

1. *Angles R.* A Comparison of Current Graph Database Models // Proceedings of the 2012 IEEE 28th International Conference on Data Engineering Workshops, ICDEW '12, IEEE Computer Society. Washington, DC, USA, 2012. P. 171–177.

2. *Angles R., Gutierrez C.* Survey of Graph Database Models // ACM Comput. Surv. 2008. Vol. 40 (1). P. 1:1–1:39.

3. *Shalini Batra, Charu Tyagi.* Comparative Analysis of Relational and Graph Databases // International Journal of Soft Computing and Engineering (IJSCE). 2012. Vol. 2. Is. 2. P. 509–512.

4. *Shrinivas S. G. et al.* Applications of Graph Theory in Computer Science an Overview // International Journal of Engineering Science and Technology. 2010. Vol. 9. P. 4610–4621.

Материал поступил в редколлегию 18.10.2013

I. V. Polyakov, A. A. Chepovskiy, A. M. Chepovskiy

SOCIAL NETWORKS STORING AND PROCESSING

In this paper special data structure for big social graph storing and operating is presented. We discuss mainly graph paths searching, obtaining subgrapths and addition of new edges and vertices.

Keywords: graph warehouse, graph algorithms, big data.