УДК 681.3

Построение индекса по иерархии записей в реляционной базе данных

Беззубов С.Н., Майоров А.В.
Ярославский государственный университет, 2005

Аннотация

В статье предлагается метод построения индекса по логической иерархии записей в реляционной базе данных. Индекс облегчает выборку записей, лежащих ниже или выше по иерархии, в том числе и на заданном расстоянии от заданной записи.

1. Постановка задачи

Дана реляционная база данных, в которых среди всего множества связей между таблицами можно выделить иерархическую структуру. При этом иерархия не обязательно должна иметь древовидную структуру (то есть у каждой записи может быть больше одного родителя), но в ней не должно быть циклов. Другими словами, записи образуют направленный ациклический граф.

Задача: разработать методику, позволяющую максимально быстро выбирать всех потомков или всех предков заданной записи или группы записей. В том числе и предков или потомков, удаленных от заданной записи на заданное число шагов.

2. Существующие решения

2.1. CONNECT BY в Oracle

Популярная СУБД Oracle уже давно имеет возможность формирования иерархии внутри выборки. Для этого в команду SELECT добавляется конструкция CONNECT BY, указывающая как именно нужно связывать записи между собой. [1] Например:

```
SELECT name
FROM employee
START WITH name = 'John'
CONNECT BY PRIOR employee_id = manager
```

Этот запрос выбирает работника по имени John, его подчиненных, подчиненных его подчиненных и так далее, до тех пор, пока не дойдет до низа иерархической лестницы. Если бы в запросе не было указано, с какой записи нужно начинать рекурсию, то запрос выбрал бы все существующие деревья подчинения.

Очевидно, что данный подход позволяет решить задачу выборки дочерних или родительских записей. Если учесть, что в таком запросе еще можно использовать псевдоколонку LEVEL, то мы можем ограничить выборку и по удаленности от стартовой записи.

Недостатки этого метода:

1. Для получения списка объектов выполняется столько запросов, сколько существует уровней иерархии. Таким образом, скорость работы запроса равна скорости выборки записей по полю связи (в нашем примере – manager) умноженной на количество уровней иерархии.

2. Довольно сложно делать гетерогенные иерархии (т.е. иерархии, содержащие записи из разных таблиц).
3. Данный подход работает только в Oracle.

2.2. Common Table Expressions

В стандарте ANSI SQL-99 была введена конструкция Common Table Expressions (CTE), позволяющая, в частности, делать рекурсивные выборки.

CTE похожи на представления (view), но в отличие от последних они не сохраняются в схеме базы данных. На них можно ссылаться только в том блоке кода, где они декларируются. Причем ссылаться можно и изнутри этого самого выражения, организуя рекурсию. Вот запрос из первого раздела, переписанный при помощи CTE:

```
WITH t( employee, name ) AS (
     SELECT   employee_id, name
     FROM     employee
     WHERE    name = 'John'
     UNION ALL
     SELECT   next.employee_id, next.name
     FROM     employee as next
              INNER JOIN t ON t.employee_id = next.manager
)

SELECT name
FROM t;
```

Конструкция получилась более громоздкая, но работает она примерно так же. Первый SELECT внутри CTE заменяет собой START WITH, второй – CONNECT BY PRIOR. В том случае, если нам понадобится индикатор уровня вложенности, то его реализация также вполне прямолинейна. Более подробно о соответствии между CONNECT BY и CTE можно узнать в статье «Port CONNECT BY to DB2» [2].

Сравним этот подход с предыдущим:

1. Скорость работы должна быть сравнимой с предыдущим вариантом.
2. Гетерогенные иерархии строятся проще. При помощи оператора UNION ALL мы можем объединять несколько выборок, присоединяющих к иерархии разные таблицы, каждый раз используя специфический критерий связи.
3. На данный момент Common Table Expressions поддерживаются основными коммерческими СУБД (DB2, MS SQL Server, Oracle), но не популярными СУБД с открытым кодом (MySQL, Firebird, PostgreSQL и другие).

2.3. Построение иерархии вручную

Если в СУБД нет поддержки рекурсивных запросов, то можно эмулировать их вручную, используя специальную временную таблицу, а потом выбрать из этой таблицы записи, находящиеся вне заданного диапазона расстояний от начальной записи.

Сравнивая этот подход с предыдущими, мы понимаем, что он, в общем-то, очень похож на подход с использованием CTE. При этом он может быть реализован в любой базе данных, поддерживающей хранимые процедуры.

В то же время, ручной метод не позволяет надеяться на ту поддержку со стороны СУБД, которая теоретически могла бы существовать для штатных средств. Например, оптимизация выполнения рекурсивного запроса в зависимости от внешних критериев, кэширование результатов и т.п. Таким образом, скорость выборки при данном подходе будет наименьшей.

3. Цель работы

Как мы увидели, штатные методы решения нашей задачи сопряжены с итеративной выборкой связанных записей. Хотя при каждой из этих выборок может быть использован индекс по полю связи, это все равно будет столько выборок по индексу, сколько существует уровней иерархии.

В идеале хотелось бы, чтобы необходимый нам запрос выполнялся за одну выборку по одному индексу. Этого можно добиться, если у нас будет таблица, хранящая все пути между любыми связанными записями базы данных:

RelationMap(child, parent, distance)

Здесь:

- child – идентификатор дочерней записи,
- parent – идентификатор родительской записи,
- distance – расстояние между этими записями.
- В таблице должен быть построен индекс, включающий в себя все три поля.

С такой таблицей, поставленная нами задача решается простейшим запросом, приводящим к одной выборке по индексу без каких-либо рекурсивных итераций.

Очевидно, что эта таблица должна всегда содержать актуальные данные о связях между записями, то есть она должна обновляться вместе с изменением связей между записями, и это обновление должно занимать как можно меньше времени.

4. Разбор задачи на примере

Рассмотрим задачу на примере следующего направленного ациклического графа объектов, представленного на рис. 1. Расположенные выше объекты будем считать родительскими, ниже – дочерними. Пути внутри графа ведут от дочерних объектов к родительским; длина пути равна количеству ребер, входящих в этот путь.

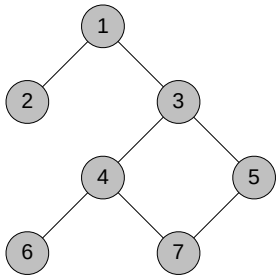

Рис. 1

Отметим, что объект 7 и объект 3 связывают два пути с одинаковой длиной. Один путь ведет через объект 4, другой – через объект 5. По этой же причине есть два одинаковых пути из 7 в 1.

Добавим новый элемент в иерархию и рассмотрим это на фрагменте дерева (рис. 2).

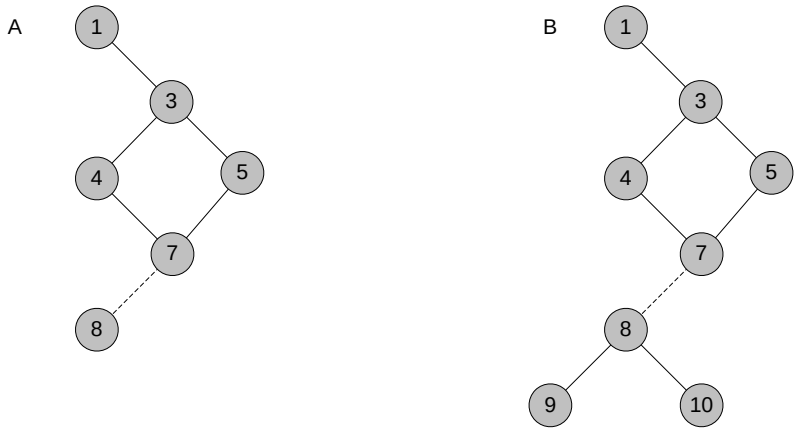

Рис. 2

В варианте А мы добавляем связь между элементами 7 и 8. Очевидно, что при этом у нас появляются следующие новые пути:

1. Прямой путь из 8 в 7 (длина 1).
2. Пути из объекта 8 через объект 7 во все объекты, в которые можно попасть из объекта 7. При этом длина всех путей будет на единицу больше, чем из объекта 7.

В варианте В мы также добавляем связь между объектами 7 и 8, но при этом у нас уже есть независимый фрагмент графа, состоящий из объектов 8, 9, 10. Здесь добавляются следующие пути:

1. Прямой путь из 8 в 7 (длина 1).
2. Из 8 ко всем объектам, принадлежащим множеству предков объекта 7, с длиной увеличенной на единицу.
3. Из 7 ко всем объектам, принадлежащих множеству потомков объекта 8, с длиной увеличенной на единицу.
4. Из всех объектов, до которых есть пути из 7, мы теперь также можем попасть в объекты 9 и 10, и наоборот. При этом длина пути между потомком объекта 8 и предком объекта 7 будет равна сумме длин путей до 8 и до 7 соответственно плюс 1.

Отдельно следует рассмотреть вопрос количества одинаковых путей, которые возникнут в результате соединения фрагментов графа (рис 2, В). Если из объекта 7 в объект x можно попасть cx путями с длиной dx, а из объекта y в объект 8 – cy путями с длиной dy, то из x в y можно будет попасть $cx \cdot cy$ путями с длиной $dx+dy+1$.

Таким образом, мы видим, что при добавлении связи между некоторыми объектами *parent* (родитель) и *child* (ребенок) появляется следующий набор путей:

для всех
$x \in descendants(child)$,
$dx \in distances(x,child)$,
$y \in ancestors(parent)$,
$dy \in distances(parent,y)$
добавляется $count(x,child,dx)*count(parent,y,dy)$ путей с длиной $dx+dy+1$, ведущих из x в y.

Здесь:

- *ancestors(x)* – множество объектов, до которых из объекта x есть путь вверх по иерархии. Другими словами – множество предков. Считаем, что множество предков объекта x содержит и сам этот объект.
- *descendants(x)* – множество объектов, до которых из объекта x есть путь вниз по иерархии, т.е. множество потомков. Также включает в себя сам объект x.

- *distances(x,y)* – множество различных длин путей между связанными объектами *x* и *y*. *distances(x,x)* ≡ *{0}*.
- *count(x,y,distance)* – количество различных путей из *x* в *y*, имеющих длину *distance*.

Очевидно, что при удалении связи между объектами, мы будем должны сделать такие же выборки, как и при вставке, сформировать множество путей затрагиваемых этой связью, и вычесть его из всего множества существующих путей.

5. Решение для реляционной модели

Разобранный нами выше пример может быть смоделирован следующей системой таблиц (рис. 3).

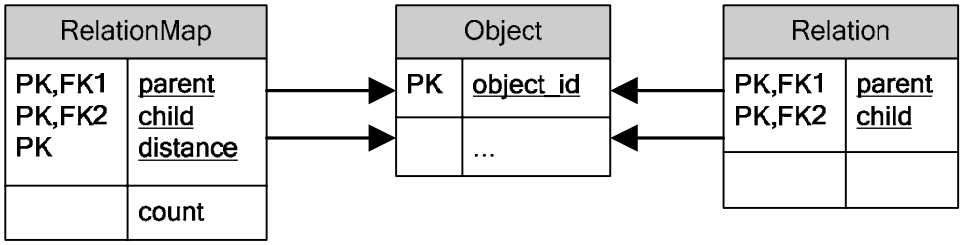

Рис. 3

Здесь таблица Object содержит объекты системы, идентифицируемые целочисленным ключом, а записи таблицы Relation связывают эти объекты попарно. При этом в каждой паре выделяются родительский и дочерний объекты. Таблица RelationMap содержит карту путей между объектами.

Заметим, что, так как тройка parent, child, distance в таблице RelationMap является первичным ключом, мы не можем сохранять одинаковые пути в виде отдельных записей. Для их учета мы ввели дополнительное поле count.

В нашей модели только таблица Relation отвечает за формирование графа объектов, поэтому для поддержания карты связей в актуальном состоянии нам нужно будет отслеживать изменения именно в этой таблице. При этом нас интересует только вставка и удаление записи, так как, во-первых, изменение первичного ключа является плохой практикой, а, во-вторых, любое изменение можно выразить в виде пары операций удаление-вставка.

Таким образом, нам необходимо создать триггеры, которые будут срабатывать по вставке или удалению записей таблицы Relation и обновлять таблицу RelationMap соответственно произошедшим изменениям.

Так как в любой момент времени таблица RelationMap содержит полную карту связей внутри графа, то мы можем использовать ее при построении множества дополнительных путей, возникающих при создании новой связи. Следующая выборка дает нам множество путей, появляющихся при добавлении связи между объектами с идентификаторами @iParent и @iChild (в нотации MS SQL Server):

```
SELECT    descendants.child,
          ancestors.parent,
          ancestors.distance + descendants.distance + 1,
          sum( ancestors.count * descendants.count )
FROM      (
          SELECT  *
          FROM    RelationMap
          WHERE   child = @iParent
          UNION
          SELECT  @iParent, @iParent, 0, 1
```

```
          ) AS ancestors,
          (
          SELECT  *
          FROM    RelationMap
          WHERE   parent = @iChild
          UNION
          SELECT  @iChild, @iChild, 0, 1
          ) AS descendants
GROUP BY  descendants.object,
          ancestors.ancestor,
          ancestors.distance + descendatds.distance + 1
```

Принципы работы запроса:

1. Выбираются все пути, в которых родительский объект вставляемой связи выступает в качестве потомка. К выборке добавляется путь от этого объекта к самому себе с длиной 0 и количеством повторений 1.
2. Выбираются все пути, в которых дочерний объект выступает в качестве предка. Также добавляется путь к самому себе.
3. Делается декартово произведение этих двух выборок, и получается выборка всех возможных путей от объектов второго множества к объектам первого. Длины путей складываются, количество повторений перемножается.
4. Так как в результате складывания длин путей, мы можем получить новые наборы путей с идентичной длиной, мы группируем набор записей и суммируем количество повторений внутри групп.

Получившуюся выборку нужно добавить к таблице RelationMap (таблица @tSubTree – временное хранилище результатов предыдущей выборки):

```
UPDATE    RelationMap
SET       count = om.count + st.count
FROM      @tSubTree st
INNER     JOIN RelationMap om
ON        st.child = om.child
AND       st.parent = om.parent
AND       st.distance = om.distance

INSERT    INTO RelationMap
SELECT    st.child, st.parent, st.distance, st.count
FROM      @tSubTree st
LEFT      JOIN RelationMap om
ON        st.child = om.child
AND       st.parent = om.parent
AND       st.distance = om.distance
WHERE     om.child IS NULL
```

Здесь мы сначала добавляем количество повторений к уже существующим путям, а затем вставляем те пути, которых ранее не было.

Необходимо отметить, что инкрементальный характер алгоритма построения множества путей, возникающих при создании новой связи, не позволяет надежно вставлять несколько связей параллельно. Каждая вставка должна быть просчитана последовательно, при этом порядок выполнения вставок не важен.

Карта путей помогает также избежать появления циклов в графе объектов. Для этого, в начале триггера, обрабатывающего вставку новой связи нужно проверить, не существует ли уже путей, ведущих от @iParent к @iChild (т.е. в обратном направлении).

Аналогично вставке, процесс удаления связи также полностью опирается на содержимое таблицы RelationMap. Мы делаем точно такую же выборку путей, как и при вставке, а затем «вычитаем» ее из таблицы RelationMap:

```
UPDATE    RelationMap
SET       count = om.count - st.count
FROM      @tSubTree st
INNER     JOIN RelationMap om
ON        st.child = om.child
AND       st.parent = om.parent
AND       st.distance = om.distance

DELETE
FROM      RelationMap
WHERE     count = 0
```

Так как в данном случае, все пути уже должны существовать в таблице RelationMap, мы вычитаем количество повторений путей в выборке из общего количества повторений этих путей, а затем удаляем записи, где количество повторений равно 0.

Аналогично вставке, удаление связей также должно выполняться последовательно.

6. Расширение до гетерогенного случая

Мы рассмотрели упрощенный случай, когда имеется только одна таблица связи, а все объекты имеют сквозную нумерацию. Возможны модели, когда иерархия строится по-другому. Например, следующим образом: в таблице A есть поля, ссылающиеся на таблицы B и C, и связь многие-ко-многим к таблице D (хранится в таблице AD). При этом считается, что записи таблицы A подчинены соответствующим записям из B и C, а записи таблицы D находятся в подчиненном положении по отношению к связанным с ними записям таблицы A.

Основной сложностью в этом случае является построение такой таблицы RelationMap, которая могла бы хранить ссылки на объекты разных типов. Например, если все таблицы имеют целочисленный первичный ключ, то таблица RelationMap может быть такой: parent, parentType, child, childType, distance, count. Если ключи таблиц имеют разные типы, то можно, в принципе, использовать строковые ссылки. Хотя, конечно, производительность от этого пострадает.

После того, как построена обобщающая таблица с путями, мы можем поддерживать ее практически тем же способом – при помощи триггеров на таблицах со связями. В нашем примере нужно будет создать триггеры на таблицах A (отлеживать связи в B и C) и AD (отслеживать связи из D в A).

7. Характеристики метода

Достоинства описанного метода:

1. Выборки, описанные в постановке задачи, делаются за один запрос и с использованием одного индекса. Синтаксис выборок очень прост и подразумевает только связь целевой таблицы с картой путей.
2. Обновление карты путей происходит по мере изменения данных. Алгоритм обновления состоит из набора простых операций с самой картой: выборка по индексу, группировка, обновление таблицы.
3. Метод может быть реализован для любой базы данных, в которой поддерживаются триггеры.
4. В том случае, если в таблице RelationMap создать кластерный индекс по всем полям (clustered covering index), то СУБД (по крайней мере – MS SQL Server) будет хранить данные записей в самом индексе. В этом случае мы, фактически, получаем просто некий иерархический индекс, распространяющийся на несколько таблиц.

Недостатки подхода:

1. Если граф содержит много уровней иерархии, то таблица путей может получиться достаточно большой.
2. Хотя алгоритм обновления таблицы путей прост и требует всего двух выборок по хорошему индексу, он не бесплатен (что, впрочем, очевидно).

Хочется заметить, что все перечисленные недостатки относятся также и к обычным индексам базы данных – они занимают место и на их построение требуется время. В то же время, выигрыш от использования индексов многократно превышает затраты. Конечно, в том случае, если индексы правильно применены. Это справедливо и для нашего метода.

8. Литература

1. David C. Kreines. Oracle SQL: The Essential Reference // ISBN: 1-56592-697-8, O'Reilly, 2000.
2. Serge Rielau. Port CONNECT BY to DB2 // http://www-128.ibm.com/developerworks/db2/library/techarticle/dm-0510rielau/, 13 Oct 2005.