Деревья в SQL

Деревом называется упорядоченный граф, в котором от каждого узла к вершине ведет только один путь. С помощью дерева можно представить структуру каталогов и файлов на жестком диске, иерархию подразделений и сотрудников компании или административно-территориальное строение государства.

Оглавление

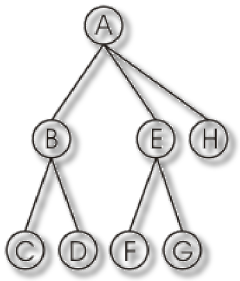

Рис. 1 Дерево.

На рис. 1 приведен пример дерева. Вершина A называется корнем дерева. Узлы, которые не имеют поддеревьев — C, D, F, G и H — называются листьями. Дуги дерева показывают отношения родитель-потомок между его узлами. Дерево называется n-мерным, если каждый его узел имеет не более чем n потомков. Частный случай n-мерного дерева — двоичное дерево, когда каждый узел содержит не более двух потомков.

Извлечение списка всех узлов некоторого поддерева, заданного его вершиной, является одной из самых распространенных задач, решаемых в прикладных программах. Например, чтобы определить обороты по бухгалтерскому счету, необходимо просуммировать проводки по нему и по всем его субсчетам. В складской программе может понадобиться список товаров, принадлежащих некоторой товарной группе и ее подгруппам, причем, таких подгрупп может быть неограниченное количество и сами они могут иметь вложенные подгруппы и т.д. Хотя, теоретически такая задача решается элементарно с помощью рекурсивного обхода дерева[1], на практике мы можем столкнуться с рядом ограничений, когда применение процедурного подхода неприемлемо. В таком случае, следует применять организацию древовидных данных, которая бы позволяла организовать т.н. нерекурсивный обход, т.е. извлечение всех узлов заданного поддерева без обращения к рекурсивным процедурам.

Целью настоящей статьи является рассмотрение четырех возможных вариантов хранения древовидных структур в реляционной базе данных. Для каждого из рассматриваемых вариантов мы приведем пример запроса на извлечение узлов заданного поддерева, а также рассмотрим такие элементарные операции, как: добавление узла в существующее дерево или вершины нового дерева, перемещение или удаление поддерева.

Все, приведенные в настоящей статье SQL запросы, создавались и тестировались для серверов Firebird (http://firebirdsql.org) и Yaffil, клонов с открытым исходным кодом хорошо известного в Беларуси и за ее пределами сервера баз данных Borland Interbase.

Ссылка на родителя

Пожалуй, самый простой способ представления дерева в реляционной базе — это помещение матрицы смежности в таблицу. Каждая запись такой таблицы соответствует узлу дерева и хранит его уникальный идентификатор и ссылку на родительский узел.

Ниже приведен DDL запрос для создания таблицы TEST1 (щелкните по тексту запроса для загрузки его текста в браузер):

```
CREATE TABLE test1 (
  id INTEGER NOT NULL,
  parent INTEGER,
  PRIMARY KEY (id),
  FOREIGN KEY (parent) REFERENCES test1 (id)
    ON UPDATE CASCADE
    ON DELETE CASCADE
)
```

Обратите внимание на правило ON DELETE CASCADE, заданное для внешнего ключа по полю PARENT. Благодаря нему, при удалении узла будет автоматически удалено и все поддерево, для которого этот узел является корнем.

Манипуляции с данными дерева

Добавление узла в дерево осуществляется с помощью простой команды вставки записи, например:

```
INSERT INTO test1 (id, parent)
VALUES (<идентификатор узла>, <идентификатор родителя>);
```

Для перемещения узла к другому родителю следует выполнить команду UPDATE:

```
UPDATE test1
SET parent = <идентификатор нового родителя>
WHERE id = <идентификатор перемещаемого узла>;
```

Удаление узла и всего поддерева осуществляется с помощью команды DELETE:

```
DELETE FROM test1
WHERE id = <идентификатор удаляемого узла>;
```

Извлечение узлов поддерева

Очевидно, что при такой организации таблицы, с помощью одного SQL запроса можно извлечь только определенное число дочерних уровней. Ниже приведен запрос, который извлекает идентификаторы всех потомков для узла, заданного параметром P:

```
SELECT id
FROM
  test1
WHERE
  parent = :P
```

Если мы хотим узнать список потомков для всех потомков узла P, то запрос следует переписать следующим образом:

```
SELECT t1.id
FROM
  test1 t1 JOIN test1 t2
    ON t1.parent = t2.id
WHERE
  t2.parent = :P
```

Для продвижения вглубь дерева придется добавлять все больше и больше таблиц в самообъединение.

Еще хуже обстоит дело, если необходимо извлечь список всех узлов поддерева, независимо от уровня вложенности. В этом случае придется использовать оператор UNION для объединения результатов нескольких запросов, возвращающих узлы конкретного уровня. Например, SQL для извлечения всех потомков узла P и всех потомков потомков будет выглядеть следующим образом:

```
SELECT id
FROM
  test1
WHERE
  parent = :P

UNION

SELECT t1.id
FROM
  test1 t1 JOIN test1 t2
    ON t1.parent = t2.id
WHERE
  t2.parent = :P
```

Если высота дерева заранее неизвестна, то для извлечения всех узлов поддерева, для которого заданный узел является вершиной, придется создать рекурсивную процедуру.

Пример такой процедуры приведен ниже:

```
CREATE PROCEDURE RecTest1(AnID INTEGER, Self INTEGER)
  RETURNS (ID INTEGER)
AS
BEGIN
  IF (:Self <> 0) THEN
  BEGIN
    ID = :AnID;
    SUSPEND;
  END

  FOR SELECT id FROM test1
  WHERE parent = :AnID INTO :ID
    DO FOR SELECT id FROM RecTest1(:ID, 1) INTO :ID
      DO SUSPEND;
END;
```

Первым параметром на вход процедуры передается идентификатор вершины поддерева. Второй входящий параметр определяет, будет ли присутствовать сама вершина поддерева в результирующей выборке (0 — нет, любое число, отличное от нуля — да).

Тогда запрос, извлекающий все вложенные уровни для узла с идентификатором, заданным параметром P, независимо от их количества, будет выглядеть следующим образом:

```
SELECT t.*
FROM test1 t
  JOIN RecTest1(:P, 0) r ON t.id = r.id
```

Многие сервера баз данных имеют ограничение на максимальное количество рекурсивных вызовов процедуры. Например, для Firebird (http://firebirdsql.org) это ограничение составляет порядка одной тысячи, чего, впрочем, более чем достаточно для выполнения большинства задач.

Хранение вспомогательных связей

Имея для каждого узла только идентификатор родителя, мы вынуждены прибегать к рекурсии для обхода всего дерева. Избежать этого нежелательного явления можно, если хранить все связи между конкретным узлом и всеми узлами на пути от него к вершине дерева. В таком случае нам потребуется две таблицы. Первая — будет хранить список узлов дерева:

```
CREATE TABLE test2(
  id INTEGER NOT NULL,

  PRIMARY KEY (id)
);
```

Вторая таблица будет хранить связи между каждым узлом дерева и узлами, находящимся на пути от него к вершине, включая и саму вершину дерева. Связь задается тремя числами: идентификатором узла, от которого начинается путь, идентификатором конечного узла на пути и расстоянием — целым числом, равным количеству дуг между ними. Т.е. расстояние между потомком и родителем равно 1, между потомком и родителем родителя равно 2 и т.д. вплоть до самого корня.

```
CREATE TABLE test2_link(
  id_from INTEGER NOT NULL,
  id_to INTEGER NOT NULL,
  distance INTEGER NOT NULL,

  PRIMARY KEY (id_from, id_to),
  FOREIGN KEY (id_from) REFERENCES test2 (id)
    ON UPDATE CASCADE
    ON DELETE CASCADE,
  FOREIGN KEY (id_to) REFERENCES test2 (id)
    ON UPDATE CASCADE
    ON DELETE CASCADE
);
```

Для того чтобы в последствии иметь возможность извлекать одним запросом все узлы поддерева, включая и его вершину, мы будем хранить в таблице test2_link связь от узла к самому себе, присвоив ей расстояние, равное 0.

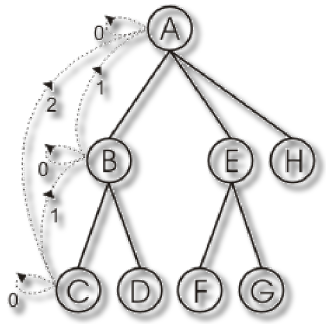

Рис. 2 Связи на пути от узла к вершине дерева.

Для дерева, показанного на Рис. 2, таблица test2_link будет хранить следующие данные:

id_from	id_to	distance
A	A	0
B	B	0
C	C	0
D	D	0
E	E	0
F	F	0
G	G	0
H	H	0
B	A	1
C	B	1
D	B	1
C	A	2
D	A	2
E	A	1
F	E	1
G	E	1
F	A	2
G	A	2
H	A	1

Извлечение узлов поддерева

При такой организации данных, запрос, извлекающий все узлы поддерева с заданной вершиной, будет выглядеть следующим образом:

```
SELECT
  t.*
FROM
  test2 t JOIN test2_link l
    ON l.id_from = t.id
WHERE
  l.id_to = :P AND l.distance > 0
```

Обратите внимание на последнее условие — l.distance > 0. Заменив строгое неравенство нестрогим, мы добьемся того, что сама вершина поддерева будет включена в результирующую выборку.

Манипуляции с данными дерева

Хотя, нам и удалось избежать рекурсии при извлечении данных, но только ценой существенного усложнения вставки и модификации записей. Во-первых, для добавления узла придется выполнить две вставки: в таблицу узлов дерева и в таблицу связей. Например, для того чтобы добавить дочерний узел с идентификатором 2 к узлу с идентификатором 1, придется выполнить две следующие команды:

```
INSERT INTO test2 (id) VALUES (2);
INSERT INTO test2_link (id_from, id_to, distance)
  VALUES (2, 1, 1);
```

Но это еще не все. Нам понадобятся два триггера. Один, будет срабатывать после вставки записи в таблицу test2, и добавлять в test2_link циклическую связь с расстоянием 0:

```
CREATE TRIGGER test2_ai FOR test2
  AFTER INSERT
  POSITION 32000
AS
BEGIN
  INSERT INTO test2_link (id_from, id_to, distance)
    VALUES (NEW.id, NEW.id, 0);
END;
```

Второй триггер будет срабатывать после вставки записи в таблицу test2_link. Его задача: создать связи между добавленным узлом и нижележащими узлами (если добавляется не одиночный узел, а поддерево) и всеми узлами, лежащими выше родителя:

```
CREATE TRIGGER test2_link_ai FOR test2_link
  AFTER INSERT
  POSITION 32000
AS
  DECLARE VARIABLE AnID INTEGER;
  DECLARE VARIABLE ADistance INTEGER;
BEGIN
  IF (NEW.distance = 1) THEN
  BEGIN
    INSERT INTO test2_link (id_from, id_to, distance)
      SELECT down.id_from, up.id_to,
        down.distance + up.distance + 1
      FROM test2_link up, test2_link down
      WHERE up.id_from = NEW.id_to
        AND down.id_to = NEW.id_from
        AND down.distance + up.distance > 0;
  END
END;
```

Теперь рассмотрим перемещение узла (поддерева) от одного родителя к другому. Данную операцию мы разложим на две элементарные операции: выделение поддерева в самостоятельное дерево, т.е. вершина поддерева не будет иметь родителя, с помощью команды:

```
DELETE FROM test2_link
WHERE
  id_from = <ИД перемещаемого поддерева или узла>
  AND id_to = <Идентификатор родителя>
  AND distance = 1
```

и добавление поддерева (одиночного узла) к новому родителю:

```
INSERT INTO test2_link (id_from, id_to, distance)
  VALUES (
  <ИД вершины перемещаемого поддерева или узла >,
  <ИД нового родителя>,
  1);
```

Поскольку триггер на вставку записи мы уже создали выше, займемся триггером, который при удалении связи с расстоянием 1 разорвет (удалит из таблицы) все связи между узлами поддерева и узлами, лежащими на пути от родительского узла к вершине дерева:

```
CREATE TRIGGER test2_link_bd FOR test2_link
  BEFORE DELETE
  POSITION 32000
AS
  DECLARE VARIABLE AnIDFrom INTEGER;
  DECLARE VARIABLE AnIDTo INTEGER;
BEGIN
  IF (OLD.distance = 1) THEN
  BEGIN
    FOR
      SELECT down.id_from, up.id_to
      FROM test2_link down
        JOIN test2_link up
        ON up.id_from = OLD.id_from
          AND up.distance + down.distance > 1
          AND down.id_to = OLD.id_from
      INTO :AnIDFrom, :AnIDTo
      DO
        DELETE FROM test2_link
        WHERE id_from = :AnIDFrom
          AND id_to = :AnIDTo;
  END
END;
```

Обратите внимание, что операция удаления узла не требует особой заботы с нашей стороны — внешние ключи в таблице test2_link настроены так, что в случае удаления узла из таблицы test2, все связанные записи из таблицы test2_link удалятся автоматически. Т.е. для удаления узла

(поддерева) достаточно выполнить команду:

```
DELETE FROM test2 WHERE id = <Идентификатор узла>
```

В качестве последнего штриха можно создать два триггера, которые бы гарантировали логическую целостность данных и соответствие их нашей модели. Один триггер запрещает редактирование записей в таблице test2_link, а второй — удаляет узел, если из таблицы связей удаляется запись с расстоянием 0:

```
CREATE EXCEPTION tree_e_incorrect_operation
  ‘Incorrect operation’;

CREATE TRIGGER test2_link_au FOR test2_link
  BEFORE UPDATE
  POSITION 32000
AS
BEGIN
  EXCEPTION tree_e_incorrect_operation;
END;

CREATE TRIGGER test2_link_ad FOR test2_link
  AFTER DELETE
  POSITION 32000
AS
BEGIN
  IF (OLD.distance = 0) THEN
  BEGIN
    DELETE FROM test2 WHERE id = OLD.id_from;
  END
END;
```

Составной путь

Не обязательно для хранения списка узлов на пути к корню дерева использовать отдельную таблицу. Можно объединить упорядоченный от корня список идентификаторов в строку, используя произвольный символ-разделитель[2], и хранить ее непосредственно в записи таблицы узлов, например, так:

```
CREATE TABLE test3 (
  id INTEGER NOT NULL,
  parent INTEGER,
  path VARCHAR(60) NOT NULL,

  PRIMARY KEY (id),
  FOREIGN KEY (parent) REFERENCES test3 (id)
    ON UPDATE CASCADE
    ON DELETE CASCADE
);
```

Поскольку, для извлечения записей мы будем использовать оператор STARTING WITH, следующий индекс просто необходим для увеличения быстродействия:

```
CREATE INDEX test3_x_path ON test3 (path);
```

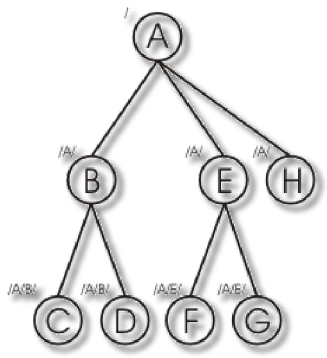

Рис. 3 Составные пути для каждого из узлов дерева.

Если в качестве символа-разделителя выбран слэш, то для нашего исходного дерева, отображенного на Рис. 3, таблица test3 будет заполнена следующим образом:

id	parent	path
A	NULL	/
B	A	/A/
C	B	/A/B/

D	B	/A/B/
E	A	/A/
F	E	/A/E/
G	E	/A/E/
H	A	/A/

Извлечение узлов поддерева

Запрос на извлечение всех узлов поддерева с вершиной P будет выглядеть следующим образом:

```
SELECT *
FROM test3
WHERE path STARTING WITH
  (SELECT path FROM test3 WHERE id = :P)
```

Если саму вершину поддерева требуется исключить, добавим соответствующее условие в запрос:

```
SELECT *
FROM test3
WHERE path STARTING WITH
  (SELECT path FROM test3 WHERE id = :P)
  AND id <> :P
```

Манипуляции с данными дерева

Как и в самом простом случае организации древовидных данных, рассмотреном нами выше, для добавления, перемещения или удаления узла достаточно выполнить соответствующие SQL команды: INSERT, UPDATE или DELETE.

За автоматическое формирование строки пути будут отвечать следующие триггеры:

```
CREATE TRIGGER test3_bi FOR test3
  BEFORE INSERT
  POSITION 32000
AS
  DECLARE VARIABLE P INTEGER;
BEGIN
  NEW.path = ‘’;
  P = NEW.parent;
  WHILE (NOT :P IS NULL) DO
  BEGIN
    NEW.path = CAST(:P AS VARCHAR(60)) || ‘/’
      || NEW.path;
    SELECT parent
    FROM test3
    WHERE id = :P
    INTO :P;
  END
  NEW.path = ‘/’ || NEW.path;
END;

CREATE EXCEPTION tree_e_incorrect_operation2
  ‘Incorrect operation’;

CREATE TRIGGER test3_bu FOR test3
  BEFORE UPDATE
  POSITION 32000
AS
  DECLARE VARIABLE P INTEGER;
  DECLARE VARIABLE T INTEGER;
  DECLARE VARIABLE NP VARCHAR(60);
  DECLARE VARIABLE OP VARCHAR(60);
BEGIN
  IF((NEW.parent <> OLD.parent)
    OR (NEW.parent IS NULL AND NOT OLD.parent IS NULL)
    OR (NOT NEW.parent IS NULL
      AND OLD.parent IS NULL)) THEN
  BEGIN
    NEW.path = ‘’;
    P = NEW.parent;
    WHILE (NOT :P IS NULL) DO
    BEGIN
      NEW.path = CAST(:P AS VARCHAR(60)) || '/'
        || NEW.path;
      SELECT parent
      FROM test3
      WHERE id = :P
      INTO :P;
    END
    NEW.path = ‘/’ || NEW.path;

    /* Проверим, не произошло ли зацикливания? */
    IF (POSITION(('/' ||
      CAST(NEW.id AS VARCHAR(60)) || '/')
      IN NEW.path) > 0) THEN
      EXCEPTION tree_e_incorrect_operation2;

    NP = NEW.path ||
      CAST(NEW.id AS VARCHAR(60)) || '/';
    OP = OLD.path ||
      CAST(OLD.id AS VARCHAR(60)) || '/';
    T = LENGTH(OP) + 1;

    UPDATE test3
    SET path = :NP || SUBSTRING(path FROM :T FOR 60)
    WHERE path STARTING WITH :OP;
  END
END;
```

Обратите внимание, что функции LENGTH и POSITION являются встроенными для сборки Yaffil 887 и выше. При использовании сервера Firebird вам придется подключить библиотеку fbudf.dll.

Хранение составного пути от корня дерева к текущему узлу накладывает ограничение на максимальную глубину дерева, которая приблизительно равна L div (K + 1), где L — длина строкового поля для хранения пути, а K — средняя длина строкового представления ключа записи[3].

Интервальное дерево

До сих пор мы пытались работать с древовидной структурой, как с графом. Проблемы, которые у нас возникали: использование рекурсии, дополнительной таблицы или громоздкого строкового поля были вызваны тем, что язык SQL по своей природе — это язык, предназначенный для работы с множествами, а не с графами.

Для того чтобы представить древовидную структуру в виде вложенных множеств[4] необходимо для каждого узла ввести два целочисленных параметра: левую и правую границу, так чтобы выполнялись следующие условия:

1. Правая граница узла больше, либо равна левой;
2. Левые границы потомков больше чем левая граница родительского узла;
3. Правые границы потомков меньше, либо равны правой границе родительского узла;
4. Интервалы, определенные левыми и правыми границами узлов, имеющих одного родителя, не должны пересекаться.

Следующий рисунок призван проиллюстрировать представление дерева в виде вложенных множеств:

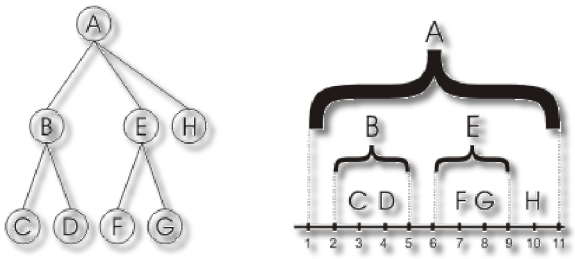

Рис. 4 Представление дерева в виде вложенных множеств.

Для хранения интервальной древовидной структуры создадим следующую таблицу:

```
CREATE TABLE test4 (
  id INTEGER NOT NULL,
  parent INTEGER,
  lb INTEGER NOT NULL,
  rb INTEGER NOT NULL,

  PRIMARY KEY (id),
  FOREIGN KEY (parent) REFERENCES test4 (id)
    ON UPDATE CASCADE
    ON DELETE CASCADE,
  CHECK (lb <= rb)
);
```

Для ускорения доступа к данным нам потребуются два индекса:

```
CREATE INDEX test4_x_lb ON test4 (lb);
CREATE DESCENDING INDEX test4_x_rb ON test4 (rb);
```

Извлечение узлов поддерева

Запрос, извлекающий все узлы поддерева, вершина которого имеет идентификатор P, будет выглядеть следующим образом:

```
SELECT t.*
FROM test4 t JOIN test4 t2
  ON t.lb > t2.lb AND t.rb <= t2.rb
WHERE
  t2.id = :P
```

Заменив строгое неравенство по левой границе на не строгое, мы добавим саму вершину поддерева в результирующую выборку.

Манипуляции с данными дерева

По сравнению с ранее рассмотренными структурами, интервальное дерево требует наибольших усилий при добавлении нового узла или при переносе узла из одной ветви в другую. Помимо двух триггеров нам понадобится еще и хранимая процедура.

Начнем рассмотрение с процедуры EL_TEST4, которая отыскивает и возвращает левую границу для потомка заданного узла (входящий параметр PARENT). Если интервал родителя исчерпан, то он будет расширен на величину не меньшую, чем заданную через входящий параметр DELTA. При расширении интервала, будут расширены все вышележащие интервалы, а также сдвинуты все интервалы, находящиеся правее. При сдвиге и расширении из обработки исключаются интервалы, попадающие в интервал заданный границами LB2, RB2. Исключение интервалов помогает избежать зацикливания при перемещении поддерева от одного родителя к другому.

```
CREATE PROCEDURE EL_TEST4 (
  PARENT INTEGER,
  DELTA INTEGER,
  LB2 INTEGER,
  RB2 INTEGER)
RETURNS (
  LEFTBORDER INTEGER)
AS
  DECLARE VARIABLE R INTEGER;
  DECLARE VARIABLE L INTEGER;
  DECLARE VARIABLE R2 INTEGER;
  DECLARE VARIABLE MKey INTEGER;
  DECLARE VARIABLE MultiDelta INTEGER;
BEGIN
  /* Получаем границы родителя */
  SELECT rb, lb
  FROM test4
  WHERE id = :Parent
  INTO :R, :L;

  /* Получаем крайнюю правую границу потомков, */
  /* родителя, если они существуют */
  R2 = NULL;
  SELECT MAX(rb) FROM test4 WHERE parent = :Parent
    INTO :R2;

  /* Если потомков нет берем левую границу */
  /* родителя */
  IF (:R2 IS NULL) THEN
    R2 = :L;

  /* проверяем: хватит ли места для размещения */
  /* еще одного потомка в интервале родителя */
  IF (:R - :R2 < :Delta) THEN
  BEGIN
    /* Если места не достаточно -- раздвигаем */
    /* интервал */
    MultiDelta = :R - :L + 100;

    /* Проверяем удовлетворяет ли нас новый */
    /* диапазон */
    IF (:Delta > :MultiDelta) THEN
      MultiDelta = :Delta;

    /* Сдвигаем правую границу родителей */
    IF (:LB2 > -1) THEN
      UPDATE test4 SET rb = rb + :MultiDelta
      WHERE lb <= :L AND rb >= :R
        AND NOT (lb >= :LB2 AND rb <= :RB2);
    ELSE
      UPDATE test4 SET rb = rb + :MultiDelta
      WHERE lb <= :L AND rb >= :R;

    /* Сдвигаем обе границы интервалов, лежащих*/
    /* правее, от родительского интервала */
    IF (:LB2 > -1) THEN
      UPDATE test4
        SET lb = lb + :MultiDelta,
        rb = rb + :MultiDelta
        WHERE lb > :R
          AND NOT (lb >= :LB2 AND rb <= :RB2);
    ELSE
      UPDATE test4
        SET lb = lb + :MultiDelta,
        rb = rb + :MultiDelta
        WHERE lb > :R;
  END

  /* возвращаем найденную границу */
  LeftBorder = :R2 + 1;
END;
```

Триггер TEST4_BI вызывается после вставки записи и присваивает ее левую и правую границы:

```
CREATE TRIGGER test4_bi FOR test4
  BEFORE INSERT
  POSITION 32000
AS
BEGIN
  /* проверим: добавляется корень нового дерева */
  /* или потомок к существующему узлу? */
  IF (NEW.parent IS NULL) THEN
  BEGIN
  /* если добавляется новое дерево, то */
  /* расположим его левую границу за максимальной, */
  /* известной нам, правой границей из существующих*/
  /* записей в таблице. */
    NEW.lb = NULL;
    SELECT MAX(rb) FROM test4 INTO NEW.lb;

    IF (NEW.lb IS NULL) THEN
    /* добавляется самый первый элемент в таблицу */
    /* в нашем примере, мы используем целые числа */
    /* >0 для хранения границ интервалов. */
      NEW.lb = 1;
    ELSE
      NEW.lb = NEW.lb + 1;
  END ELSE
  BEGIN
    /* вызовем процедуру, которая отыщет возможную */
    /* левую границу для потомка. Если интервал */
    /* родителя исчерпан, то он будет расширен. */
    EXECUTE PROCEDURE el_test4 (NEW.parent, 1, -1, -1)
      RETURNING_VALUES NEW.lb;
  END

  NEW.rb = NEW.lb;
END;
```

Триггер TEST4_BU изменяет интервалы узла при его перемещении к другому родителю:

```
CREATE EXCEPTION tree_e_invalid_parent
  'Invalid parent';

CREATE TRIGGER test4_bu FOR test4
  BEFORE UPDATE
  POSITION 32000
AS
  DECLARE VARIABLE OldDelta INTEGER;
  DECLARE VARIABLE L INTEGER;
  DECLARE VARIABLE R INTEGER;
  DECLARE VARIABLE NewL INTEGER;
BEGIN
  /* Проверяем факт изменения PARENT */
  IF ((NEW.parent <> OLD.parent)
    OR (OLD.parent IS NULL AND NOT NEW.parent IS NULL)
    OR (NEW.parent IS NULL AND NOT OLD.parent IS NULL))
  THEN
  BEGIN
    /* Делаем проверку на зацикливание */
    IF (EXISTS(SELECT * FROM test4
      WHERE id = NEW.parent
      AND lb >= OLD.lb AND rb <= OLD.rb)) THEN
    BEGIN
      EXCEPTION tree_e_invalid_parent;
    END ELSE
    BEGIN
      IF (NEW.parent IS NULL) THEN
      BEGIN
        /* Получаем крайнюю правую границу */
        SELECT MAX(rb)
        FROM test4
        WHERE parent IS NULL
        INTO :NewL;
        /* Предполагается, что существует */
        /* хотя бы один элемент с нулл парентом */
        NewL = :NewL + 1;
      END ELSE
      BEGIN
        /* Получаем значение новой левой границы */
        EXECUTE PROCEDURE el_test4 (NEW.parent,
          OLD.rb - OLD.lb + 1, OLD.lb, OLD.rb)
        RETURNING_VALUES :NewL;
      END

      /* Определяем величину сдвига. */
      /* +1 выполняется в процедуре */
      OldDelta = :NewL - OLD.lb;
      /* Сдвигаем границы основной ветви */
      NEW.lb = OLD.lb + :OldDelta;
      NEW.rb = OLD.rb + :OldDelta;
      /* Сдвигаем границы детей */
      UPDATE test4
        SET lb = lb + :OldDelta, rb = rb + :OldDelta
        WHERE lb > OLD.lb AND rb <= OLD.rb;
    END
  END
END;
```

Удаление всех вложенных элементов дерева при удалении заданного узла гарантируется правилом ON DELETE CASCADE на внешней ссылке по полю PARENT, и мы не будем предпринимать никаких дополнительных действий по сжиманию интервалов в этом случае.

Интервальные деревья широко используются в Гедымине, технологической платформе с открытым кодом для разработки экономических приложений. Узнать более подробно о ней можно на сайте компании Golden Software of Belarus, Ltd по адресу http://gsbelarus.com (http://www.gsbelarus.com).

Заключение

Может возникнуть вполне резонный вопрос: существует ли среди всех предложенных вариантов организации древовидных структур в SQL наиболее оптимальный, пригодный для применения во всех приложениях? К сожалению, ответ на данный вопрос отрицательный. Каждая, из предложенных структур, имеет свои сильные и слабые стороны, которые должны оцениваться применительно к требованиям конкретной задачи. Например:

1. Каково общее количество записей в таблице? Очевидно, что при небольшом размере дерева различия в производительности при применении того, либо иного метода будут минимальны и несущественны.
2. Насколько часто будут редактироваться данные дерева? Интервальное дерево позволяет достичь наилучших результатов при извлечении данных, но одновременно весьма требовательно к вычислительным ресурсам при вставке новой записи или перемещении узлов.
3. Какова максимальная глубина вложенности дерева? При глубине в 10 уровней и выше применение второй и третьей методик (см. выше) приведет к значительному росту базы данных из-за необходимости хранения информации обо всех связях между узлами на пути к

вершине.

4. Что более критично для поставленной задачи: скорость извлечения информации или размер, занимаемый данными на диске?

В любом случае, выбор остается за проектировщиком базы данных, которому, надеемся, настоящая статья послужит хорошим подспорьем.

Для вашего удобства все SQL скрипты, использованные в настоящей статье собраны в один файл и доступны тут (http://gsbelarus.com/gs/images/gs/2006/tree/sql.txt).

На английском языке: Trees in SQL. Part 1 (https://dev.to/andreik/trees-in-sql-4fp), Trees in SQL. Part 2 (https://dev.to/andreik/trees-in-sql-part-2-15pa).

Андрей Киреев, andreik(at)gsbelarus.com

[1] Различают рекурсивные обходы вширь и вглубь. В первом случае, сначала обрабатываются все потомки заданного узла, а затем для каждого из них процедура вызывается рекурсивно. Во втором — сначала для каждого потомка вызывается рекурсивная процедура и только за тем обрабатываются непосредственно узлы-потомки заданного узла. Если рекурсивная процедура построена таким образом, что сначала обрабатывает вершину, а затем рекурсивно вызывает сама себя для каждого из потомков, то можно говорить о комбинированном методе обхода дерева.

[2] Разумеется, символ-разделитель должен быть подобран таким образом, чтобы не встречаться в возможном строковом представлении идентификатора узла.

[3] В нашем примере мы используем строковое представление целого числа в десятичной системе исчисления. Написав несложную UDF (User Defined Function — внешняя функция в сервере Interbase/Firebird) можно кодировать число в строку по произвольному базису. Например, используя в качестве цифр 254 символа можно закодировать четырех миллиардное число всего четырьмя знаками: 254^4 =4 162 314 256.

[4] Представление деревьев в виде вложенных множеств подробно рассмотрено Joe Celko в журнале DBMS Online. Стоит заметить, что в настоящей статье предложен вариант организации интервальных деревьев, когда связь родитель-потомок задается в первую очередь через отношение полей id-parent. Границы интервалов, поля lb и rb, играют скорее вспомогательную роль и используются при необходимости организовать нерекурсивный обход дерева. В варианте, предложенном Joe Celko, границы интервалов, и только они, определяют иерархию узлов в дереве.

07.01.2006

Новости

- Дайджест новостей компании за май 2025 г. (/pw/articles/news/daidzhest-novostei-kompanii-za-mai-2025-g/)
- Интеграция Гедымин с кассой Зв1! (/pw/articles/news/integratciia-gedymin-s-kassoi-3v1/)
- Golden Software примет участие в Tibo 2025 (/pw/articles/news/golden-software-primet-uchastie-v-tibo-2025/)
- Инструкция по миграции на Firebird 5 (/pw/articles/news/instruktciia-po-migratcii-na-firebird-5/)
- Дайджест новостей компании за апрель 2025 г. (/pw/articles/news/daidzhest-novostei-kompanii-za-aprel-2025-g/)
- Итоги участия в выставке HoReCa 2025 (/pw/articles/news/itogi-uchastiia-v-vystavke-horeca-2025/)
- Приглашаем на наш стенд на выставке HoReCa 2025! (/pw/articles/news/priglashaem-na-nash-stend-na-vystavke-horeca-2025/)
- Маркировка шин: как Гедымин помогает бизнесу соответствовать новым требованиям (/pw/articles/news/markirovka-shin-kak-gedymin-pomogaet-biznesu-sootvetstvovat-novym-trebovaniiam/)
- Дайджест новостей компании за март 2025 г. (/pw/articles/news/daidzhest-novostei-kompanii-za-mart-2025-g/)
- QR-коды в Гедымине: еще больше удобства для бизнеса (/pw/articles/news/qr-kody-v-gedymine-eshche-bol-she-udobstva-dlia-biznesa/)
- С Днем клиента! (/pw/articles/news/den-klienta/)
- Golden Software примет участие в Expo HORECA Minsk 2025 (/pw/articles/news/ooo-ampersant-primet-uchastie-v-expo-horeca-minsk-2025/)
- Итоги конференции "АгроУправление-2025" (/pw/articles/news/itogi-konferentcii-agroupravlenie-2025/)
- Дайджест новостей компании за февраль 2025 г. (/pw/articles/news/daidzhest-novostei-kompanii-za-fevral-2025-g/)
- Выдающиеся результаты Рассветовской средней школы благодаря спонсорсту ООО "Амперсант" (/pw/articles/news/vydaiushchiesia-rezul-taty-rassvetovskoi-srednei-shkoly-blagodaria-sponsorstu-ooo-ampersant/)
- Дайджест новостей компании за январь 2025 г. (/pw/articles/news/daidzhest-novostei-kompanii-za-ianvar-2025-g/)
- С Новым 2025 годом! (/pw/articles/news/s-novym-2025-godom/)
- Дайджест новостей компании за декабрь 2024 г. (/pw/articles/news/daidzhest-novostei-kompanii-za-dekabr-2024-g/)
- ООО «Амперсант» приняло участие в семинаре-презентации цифровых решений для белорусского рынка в ПВТ (/pw/articles/news/ooo-ampersant-prinialo-uchastie-v-seminare-prezentatcii-tcifrovykh-reshenii-dlia-belorusskogo-rynka-v-pvt/)
- Дайджест новостей компании за ноябрь 2024 г. (/pw/articles/news/daidzhest-novostei-kompanii-za-noiabr-2024-g/)

Показать все... (/pw/articles/news/)

Платформа (/pw/#platform)

Узнать больше (/pw/front-page/platform/story/)

Скачать установку (/pw/downloads/)

Документация (http://gsbelarus.com/gs/wiki)

Исходный код (http://gsbelarus.com/gs/wiki/index.php?title=Компиляция_платформы_Гедымин)

Статьи (/pw/articles/post/)

Что нового? (/pw/front-page/about/whats-up/)

Решения (/pw/#solutions)