Расширяемость

Слабоструктурированные данные

Авторы: Егор Рогов, Павел Лузанов

Обратная связь
Отзывы, замечания и предложения направляйте по адресу:
edu@postgrespro.ru

Темы

Что такое слабоструктурированные данные

Применение в реляционных базах данных

Необходимые операции

Тип данных для XML: xml

Типы данных для JSON: json и jsonb

Индексирование документов JSON

Что это такое

Наиболее известные примеры: XML и JSON

Схема данных определяется самим документом

- а не хранится отдельно от данных
- документ содержит разметку для выделения элементов и определения иерархии

Данные не соответствуют реляционной модели

- конкретный документ можно представить в виде таблиц
- документы общего вида тоже можно, но практически — неудобно

Слабоструктурированные (semi-structured) данные — это данные, представленные не в реляционном (табличном) виде. Обычно это текстовый документ, но в отличие от текста на естественном языке, документ содержит разметку, определяющую структурные единицы.

Широко известными примерами форматов (языков разметки) являются XML и JSON, в последнее время также популярен YAML.

Структура (схема) документа определяется его собственной разметкой, а не хранится где-то отдельно от него. (Это не исключает возможности определить структуру отдельно — например, для XML есть языки DTD, XMLSchema).

Разметка как правило позволяет определить вложенные элементы, образующие иерархию.

Конечно, любую конкретную иерархию можно представить и в табличном виде. Можно уложить в таблицы и произвольную иерархию без жесткой схемы, но результат будет лишен типизации данных, возможности создания ограничений целостности, с ним будет крайне неудобно работать в терминах запросов SQL. Поэтому с некоторыми видами данных действительно удобнее и логичнее работать именно в таком, слабоструктурированном, виде.

Применение

Postgres PROFESSIONAL

Интерфейсный формат

- между приложением и внешней системой
- между клиентской и серверной частями приложения

Внутри базы данных как атомарное значение

- только хранение и извлечение
- не нарушает 1NF, достаточно обычных средств SQL

Внутри базы данных как неатомарное значение

- операции с отдельными частями документа
- SQL не достаточно; требуется специальный язык запросов и набор дополнительных операций
- гибкость, когда данные плохо укладываются в реляционную модель

Документы XML или JSON могут применяться как удобный интерфейсный формат между разными компонентами, независимый от конкретной платформы или языка программирования. Например, для взаимодействия с внешними системами, или для обмена данными между клиентской и серверной частями системы. В последнем случае интерфейсный формат позволяет клиенту послать сложный запрос, а серверу выполнить его эффективным образом (в отличие от традиционного подхода, принятого в ORM, когда сервер базы данных нагружается большим количеством мелких запросов, не оставляя возможностей для оптимизации).

Если слабоструктурированные данные хранятся внутри самой базы данных, то все зависит от того, как эти данные используются. Если внутри SQL запросов не делается попыток работать с отдельными частями документов, то такие значения можно считать атомарными с точки зрения СУБД. Тут достаточно обычных средств SQL.

Конечно, более интересен случай, когда серверу необходимо выделять часть документа, фильтровать вывод на основе содержания документов и т. п. Для этого SQL уже не хватает и нужен специализированный язык доступа к слабоструктурированным данным. Если СУБД предоставляет такую возможность, это позволяет комбинировать строгий реляционный подход для данных, имеющих четкую структуру, и подход в духе NoSQL для данных, которые сложно представить в табличном виде.

Операции

Преобразование документа к реляционному виду

- сохранение транспортного документа в базу данных
- использование средств SQL для обработки

Преобразование реляционных данных к виду документа

- выгрузка данных в транспортный формат

Выделение части документа

- специализированный язык запросов

Индексирование документов

- поддержка операций специализированного языка запросов

Чтобы работать со слабоструктурированными документами в реляционной базе данных, понадобятся возможности для преобразования данных из одного вида в другой, поскольку в одних ситуациях удобнее использовать реляционные средства, в других — документоориентированные. Особенно это важно, если такой язык, как XML или JSON, используется в качестве транспортного формата.

Кроме этого, нужны средства для выделения части документа — каких-то отдельных значений, или какого-то подмножества элементов. Такие средства обычно представлены в виде специализированного языка запросов, который учитывает иерархическую структуру документов. Такие языки запросов можно использовать не только для выделения части документа, но и, например, для выбора тех документов, в которых присутствует заданный фрагмент.

Если слабоструктурированные документы используются в реляционной СУБД, встает задача индексирования таких данных. Для этого, как правило, не годятся обычные B-деревья, поскольку требуется поддерживать не операции сравнения документов, а операции, связанные со специализированным языком запросов.

Формат XML

eXtensible Markup Language

универсальный язык разметки, первый стандарт 1998 года

форматы на основе XML: XHTML, FB2, RSS и Atom, SVG, SOAP...

Структура

вложенные элементы

отделяются тегами, могут иметь атрибуты, содержат текст

Инструментарий

описание схемы (DTD, XML Schema), языки запросов XPath и XQuery, язык преобразования XSLT

PostgreSQL: XPath 1.0

XML (eXtensible Markup Language) появился в контексте веба и разрабатывается консорциумом W3C. XML является текстовым документом, содержащим специальную разметку. XML определяет синтаксическую структуру разметки, которая с помощью *тегов* выделяет *элементы* (возможно, вложенные друг в друга), которые могут также иметь *атрибуты*. (Язык XML довольно сложен, в нем имеются также *инструкции обработки*, *комментарии* и другие сущности, но мы не будем на этом останавливаться.)

XML позволяет определить *расширения XML*, определяя конкретный набор элементов и их атрибутов, и наделяя их определенной семантикой. К известным расширениям относятся XHTML (более строгий HTML, совместимый с XML), FB2 (формат электронных книг), RSS и Atom (форматы для новостных лент), SVG (векторная графика), SOAP (формат сообщений веб-сервисов) и другие.

За время существования формата XML сформировался богатый инструментарий для работы с ним, включающий языки описания схемы документов (такие, как DTD, XMLSchema), языки запросов (базовый XPath и более богатый XQuery на его основе), языки преобразования документов XML (XSLT и XSL-FO на его основе).

В реляционных базах работу с XML регламентирует стандарт SQL/XML. PostgreSQL поддерживает только XPath 1.0. Преобразования XSLT доступны в расширениях (xml2 или в сторонних). Индексирование не реализовано (хотя все возможности для этого есть).

https://postgrespro.ru/docs/postgresql/12/datatype-xml

https://postgrespro.ru/docs/postgresql/12/functions-xml

XML: выражения XPath

```
=> CREATE DATABASE ext_semistruct;
```

```
CREATE DATABASE
```

```
=> \c ext_semistruct
```

```
You are now connected to database "ext_semistruct" as user "student".
```

Будем работать с документом, описывающим компоненты компьютера.

- Теги выделяются угловыми скобками. Для каждого открывающего тега (<computer>) есть соответствующий ему закрывающий, имя которого начинается на косую черту (</computer>). Такая пара тегов определяет элемент (computer).
- Атрибуты могут быть перечислены вместе со значениями в открывающем теге.
- Текст внутри тегов составляет текстовый элемент.

```
=> SELECT $xml$
<computer>                                <!-- открывающий тег -->
  <motherboard>
    <!-- текстовый элемент -->
    <cpu>Intel® Core™ i7-7567U</cpu>
    <ram>
      <!-- тег с атрибутом -->
      <dimm size_gb="32">Crucial DDR4-2400 SODIMM</dimm>
    </ram>
  </motherboard>
  <disks>
    <ssd size_gb="512">Intel 760p Series</ssd>
    <hdd size_gb="3000">Toshiba Canvio</hdd>
  </disks>
</computer>                               <!-- закрывающий тег -->
$xml$ AS xml \gset
```

Сначала посмотрим, какие средства есть для получения части XML-документа. Для этого используются выражения языка запросов XPath 1.0. Мы не будем детально рассматривать все возможности XPath, но посмотрим некоторые примеры.

XML состоит из иерархии элементов, поэтому язык описывает перемещения по дереву. Часть документа, соответствующая пути от корня:

```
=> SELECT xpath('/computer/motherboard/ram', :'xml');
```

```
                           xpath
-------------------------------------------------------------
 {"<ram>                                                    +
       <!-- тег с атрибутом -->                             +
       <dimm size_gb=\"32\">Crucial DDR4-2400 SODIMM</dimm>+
     </ram>"}
(1 row)
```

В пути можно указывать не только непосредственные потомки:

```
=> SELECT xpath('/computer//ram', :'xml');
```

```
                           xpath
-------------------------------------------------------------
 {"<ram>                                                    +
       <!-- тег с атрибутом -->                             +
       <dimm size_gb=\"32\">Crucial DDR4-2400 SODIMM</dimm>+
     </ram>"}
(1 row)
```

По дереву элементов можно «двигаться» не только вниз к листьям, но и вверх к корню:

```
=> SELECT xpath('//ram/dimm/..', :'xml');
```

```
                           xpath
-------------------------------------------------------------
 {"<ram>                                                    +
       <!-- тег с атрибутом -->                             +
       <dimm size_gb=\"32\">Crucial DDR4-2400 SODIMM</dimm>+
     </ram>"}
(1 row)
```

Вместо конкретного имени элемента можно указать, что подходит любой:

```
=> SELECT xpath('//disks/*', :'xml');
```

```
                                                 xpath
-------------------------------------------------------------------------------------------
 {"<ssd size_gb=\"512\">Intel 760p Series</ssd>","<hdd size_gb=\"3000\">Toshiba Canvio</hdd>"}
(1 row)
```

Здесь мы получили в массиве две части XML-документа. Можно посчитать количество:

```
=> SELECT xpath('count(//disks/*)', :'xml');
```

```
 xpath
-------
 {2}
(1 row)
```

Можно извлечь не весь элемент, а только его текст:

```
=> SELECT xpath('//cpu/text()', :'xml');
```

```
           xpath
--------------------------
 {"Intel® Core™ i7-7567U"}
(1 row)
```

Атрибуты записываются с помощью «собаки». Найдем значения атрибутов size_gb:

```
=> SELECT xpath('//@size_gb', :'xml');
```

```
     xpath
---------------
 {32,512,3000}
(1 row)
```

Условия фильтрации записываются в квадратных скобках. Найдем все элементы, объем памяти которых начинается от гигабайта:

```
=> SELECT xpath('//*[@size_gb >= 1024]', :'xml');
```

```
                      xpath
-----------------------------------------------
 {"<hdd size_gb=\"3000\">Toshiba Canvio</hdd>"}
(1 row)
```

Сравнение работает, потому что XPath поддерживает числовые типы. Кроме этого, поддерживаются строки и логический тип.

Выражения XPath применяются не только в функции xpath, но и в других. Например, можно проверить, содержит ли XML-документ указанный фрагмент:

```
=> SELECT xmlexists(
    '//disks/*[starts-with(text(),''Toshiba'')]'
    PASSING :'xml'
);
```

```
 xmlexists
-----------
 t
(1 row)
```

```
=> SELECT xmlexists(
    '//disks/*[starts-with(text(),''Seagate'')]'
    PASSING :'xml'
);
```

```
 xmlexists
-----------
 f
(1 row)
```

XML: преобразование в реляционный вид и обратно

Для того чтобы преобразовать XML-документ к реляционному (табличному) виду, используется функция xmltable.

Пусть у нас имеется таблица для дисковых накопителей:

```
=> CREATE TABLE disks (
    drive_type text,
    name text,
    capacity integer
);
```

```
CREATE TABLE
```

Сначала выделим нужную часть документа:

```
=> SELECT xpath('//disks/*', :'xml');
```

```
                                                 xpath
-------------------------------------------------------------------------------------------
 {"<ssd size_gb=\"512\">Intel 760p Series</ssd>","<hdd size_gb=\"3000\">Toshiba Canvio</hdd>"}
(1 row)
```

Теперь можно написать вызов функции xmltable. В нем мы указываем выражение XPath, сам документ, а также описываем, как получать значения для столбцов таблицы с помощью дополнительных XPath-выражений:

```
=> SELECT * FROM xmltable(
    '//disks/*'
    PASSING :'xml'
    COLUMNS
        drive_type  text PATH 'name()',
        name        text PATH 'text()',
        capacity integer PATH '@size_gb * 1024'
);
```

```
 drive_type |       name        | capacity
------------+-------------------+----------
 ssd        | Intel 760p Series |   524288
 hdd        | Toshiba Canvio    |  3072000
(2 rows)
```

Обратите внимание:

- выражения XPath для столбцов отсчитываются не от корня документа, а от текущего контекста (элемента, найденного основным выражением XPath);
- в выражениях можно использовать некоторые арифметические операции.

Результат такого запроса можно непосредственно вставить в таблицу:

```
=> INSERT INTO disks(drive_type, name, capacity)
SELECT * FROM xmltable(
    '//disks/*'
    PASSING :'xml'
    COLUMNS
        drive_type  text PATH 'name()',
        name        text PATH 'text()',
        capacity integer PATH '@size_gb * 1024'
);
```

```
INSERT 0 2
```

Для создания документов XML имеется довольно много функций, с помощью которых можно собрать документ «по кусочкам». Но есть также функции, позволяющие выгрузить в XML целую таблицу, или результат запроса, или даже всю базу данных в фиксированном формате.

```
=> SELECT table_to_xml(
    tbl => 'disks',
    nulls => true,
    tableforest => false,
    targetns => '' -- пространство имен
);
```

```
                          table_to_xml
-------------------------------------------------------------------
 <disks xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">+
                                                                +
 <row>                                                          +
   <drive_type>ssd</drive_type>                                 +
   <name>Intel 760p Series</name>                               +
   <capacity>524288</capacity>                                  +
 </row>                                                         +
                                                                +
 <row>                                                          +
   <drive_type>hdd</drive_type>                                 +
   <name>Toshiba Canvio</name>                                  +
   <capacity>3072000</capacity>                                 +
 </row>                                                         +
                                                                +
 </disks>                                                       +

(1 row)
```

Формат JSON

JavaScript Object Notation

- простой язык разметки, стандарт 1999 года
- появился в JavaScript, но распространен повсеместно

Структура

- объекты (пары «ключ: значение»), массивы значений
- значения текстовые, числовые, даты, логические

Инструментарий

- PostgreSQL: язык запросов JSONPath,
  неполная пока поддержка стандарта SQL/JSON,
  индексирование

Формат JSON — простое подмножество языка JavaScript, описывающее объект, состоящий из пар «ключ: значение», или массив значений. Значения могут быть следующих типов: текстовые, числовые, даты и логические (а также могут быть объектами или массивами, т. е. документ имеет иерархическую структуру).

Несмотря на то, что JSON — «родной» формат для JavaScript, он получил огромное распространение благодаря своей простоте и удобству, и повсеместно используется в современном интернете.

Изначально PostgreSQL предоставлял собственные операции для работы с JSON; при необходимости более серьезного инструментария можно было воспользоваться расширением jsQuery. В настоящее время в PostgreSQL реализована часть стандарта SQL/JSON:2016, включая язык запросов JSONPath.

Имеется и индексирование документов JSON. Это позволяет комбинировать реляционные возможности PostgreSQL с возможностями, предоставляемыми NoSQL-базами.

https://postgrespro.ru/docs/postgresql/12/datatype-json

https://postgrespro.ru/docs/postgresql/12/functions-json

https://github.com/postgrespro/jsquery

Типы данных для JSON

Postgres PROFESSIONAL

Json

- хранение в виде обычного текста + синтаксическая проверка
- сохраняется исходное форматирование
- необходимость повторного разбора при каждом запросе

Jsonb

- внутренний формат, исключающий повторный разбор
- поддержка SQL/JSON (язык запросов JSONPath)
- поддержка индексирования

Имеются два типа данных для представления документов JSON: json и jsonb.

Первый появился раньше и, по сути, просто хранит документ в виде текстовой строки (при этом, конечно, проверяется, что строка является корректным документом JSON). Но при любом обращении к части документа JSON, его приходится заново разбирать. Это вызывает большие накладные расходы. Кроме того, для формата json не реализован язык JSONPath и не работает индексирование. Поэтому json применяется редко.

Обычно используется тип jsonb (b — binary). Этот формат сохраняет однажды разобранную иерархию элементов, что позволяет эффективно работать с документом. Следует учитывать, что исходный вид документа при этом не сохраняется: нарушается порядок следования элементов, пропадают отступы и дублирующиеся ключи.

https://postgrespro.ru/docs/postgresql/12/datatype-json#DATATYPE-JSONPATH

JSON: типы данных json и jsonb

Документ, описывающий компоненты компьютера, может выглядеть в JSON так:

```
=> SELECT $js$
{ "motherboard": {
    "cpu": "Intel® Core™ i7-7567U",
    "ram": [
      { "type": "dimm",
        "size_gb": 32,
        "model": "Crucial DDR4-2400 SODIMM"
      }
    ]
  },
  "disks": [
    { "type": "ssd",
      "size_gb": 512,
      "model": "Intel 760p Series"
    },
    { "type": "hdd",
      "size_gb": 3000,
      "model": "Toshiba Canvio"
    }
  ]
}
$js$ AS json \gset
```

Формат json хранит документ как обычный текст:

```
=> SELECT :'json'::json;
```

```
                    json                     
---------------------------------------------
                                            +
 { "motherboard": {                         +
     "cpu": "Intel® Core™ i7-7567U",        +
     "ram": [                               +
       { "type": "dimm",                    +
         "size_gb": 32,                     +
         "model": "Crucial DDR4-2400 SODIMM"+
       }                                    +
     ]                                      +
   },                                       +
   "disks": [                               +
     { "type": "ssd",                       +
       "size_gb": 512,                      +
       "model": "Intel 760p Series"         +
     },                                     +
     { "type": "hdd",                       +
       "size_gb": 3000,                     +
       "model": "Toshiba Canvio"            +
     }                                      +
   ]                                        +
 }                                          +
 
(1 row)
```

В jsonb документ разбирается и записывается во внутреннем формате, сохраняющем структуру разбора. Из-за этого при выводе документ составляется заново в эквивалентном, но ином виде:

```
=> SELECT :'json'::jsonb;
```

```
                                                                                                                                  jsonb
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 {"disks": [{"type": "ssd", "model": "Intel 760p Series", "size_gb": 512}, {"type": "hdd", "model": "Toshiba Canvio", "size_gb": 3000}], "motherboard": {"cpu": "Intel® Core™ i7-7567U",
(1 row)
```

Чтобы вывести документ в человекочитаемом виде, можно использовать специальную функцию:

```
=> SELECT jsonb_pretty(:'json'::jsonb);
```

```
                     jsonb_pretty                     
------------------------------------------------------
 {                                                   +
     "disks": [                                      +
         {                                           +
             "type": "ssd",                          +
             "model": "Intel 760p Series",           +
             "size_gb": 512                          +
         },                                          +
         {                                           +
             "type": "hdd",                          +
             "model": "Toshiba Canvio",              +
             "size_gb": 3000                         +
         }                                           +
     ],                                              +
     "motherboard": {                                +
         "cpu": "Intel® Core™ i7-7567U",             +
         "ram": [                                    +
             {                                       +
                 "type": "dimm",                     +
                 "model": "Crucial DDR4-2400 SODIMM",+
                 "size_gb": 32                       +
             }                                       +
         ]                                           +
     }                                               +
 }
(1 row)
```

Дальше мы будем работать с типом jsonb, который предоставляет больше возможностей.

JSON: выражения JSONPath и другие средства

Для получения части JSON-документа стандарт SQL:2016 определил язык запросов JSONPath. Вот некоторые примеры.

Так же, как и XPath, JSONPath позволяет спускаться по дереву элементов. Часть документа, соответствующая пути от корня:

```
=> SELECT jsonb_pretty(jsonb_path_query(:'json', '$.motherboard.ram'));
```

```
                 jsonb_pretty                 
----------------------------------------------
 [                                           +
     {                                       +
         "type": "dimm",                     +
         "model": "Crucial DDR4-2400 SODIMM",+
         "size_gb": 32                       +
     }                                       +
 ]
(1 row)
```

Элементы массива указываются в квадратных скобках:

```
=> SELECT jsonb_pretty(jsonb_path_query(:'json', '$.disks[0]'));
```

```
           jsonb_pretty
-----------------------------------
 {                                 +
     "type": "ssd",                +
     "model": "Intel 760p Series",+
     "size_gb": 512                +
 }
(1 row)
```

Можно получить и все элементы сразу:

```
=> SELECT jsonb_pretty(jsonb_path_query(:'json', '$.disks[*]'));

           jsonb_pretty
-----------------------------------
 {                                 +
     "type": "ssd",                +
     "model": "Intel 760p Series",+
     "size_gb": 512                +
 }
 {                                 +
     "type": "hdd",                +
     "model": "Toshiba Canvio",    +
     "size_gb": 3000               +
 }
(2 rows)
```

Условия фильтрации записываются в скобках после вопросительного знака. Символ @ обозначает текущий путь.

Найдем диски, объем памяти которых начинается от гигабайта:

```
=> SELECT jsonb_pretty(
    jsonb_path_query(:'json', '$.disks ? (@.size_gb > 1000)')
);

          jsonb_pretty
-------------------------------
 {                             +
     "type": "hdd",            +
     "model": "Toshiba Canvio",+
     "size_gb": 3000           +
 }
(1 row)
```

Условия являются частью пути, который можно продолжить дальше. Выберем только модель:

```
=> SELECT jsonb_pretty(
    jsonb_path_query(:'json', '$.disks ? (@.size_gb > 1000).model')
);

   jsonb_pretty
------------------
 "Toshiba Canvio"
(1 row)
```

В пути может быть и несколько условий:

```
=> SELECT jsonb_pretty(
    jsonb_path_query(
        :'json',
        '$.disks ? (@.size_gb > 128).model ? (@ starts with "Intel")'
    )
);

    jsonb_pretty
--------------------
 "Intel 760p Series"
(1 row)
```

Кроме средств JSONPath, можно применять и «традиционную» стрелочную нотацию.

Переходим к ключу motherboard, затем к ключу ram, затем берем первый (нулевой) элемент массива:

```
=> SELECT jsonb_pretty( (:'json'::jsonb)->'motherboard'->'ram'->0 );

               jsonb_pretty
-----------------------------------------
 {                                       +
     "type": "dimm",                     +
     "model": "Crucial DDR4-2400 SODIMM",+
     "size_gb": 32                       +
 }
(1 row)
```

Двойная стрелка возвращает не jsonb, а текстовое представление:

```
=> SELECT (:'json'::jsonb)->'motherboard'->'ram'->0->>'model';

         ?column?
--------------------------
 Crucial DDR4-2400 SODIMM
(1 row)
```

Необходимые фильтрации в таком случае придется выполнять уже на уровне SQL.

JSON: преобразование в реляционный вид и обратно

Стандарт определяет функцию jsontable, но она пока не реализована в PostgreSQL. Разумеется, можно выйти из положения и теми средствами, которые существуют. Сначала выделим все диски:

```
=> TRUNCATE TABLE disks;

TRUNCATE TABLE

=> WITH dsk(d) AS (
    SELECT jsonb_path_query(:'json', '$.disks[*]')
)
SELECT d FROM dsk;

                               d
-----------------------------------------------------------------
 {"type": "ssd", "model": "Intel 760p Series", "size_gb": 512}
 {"type": "hdd", "model": "Toshiba Canvio", "size_gb": 3000}
(2 rows)
```

На основе этого запроса несложно сделать вставку в таблицу:

```
=> INSERT INTO disks(drive_type, name, capacity)
WITH dsk(d) AS (
    SELECT jsonb_path_query(:'json', '$.disks[*]')
)
SELECT d->>'type', d->>'model', (d->>'size_gb')::integer FROM dsk;

INSERT 0 2
```

Для обратного преобразования удобно воспользоваться функцией row_to_json:

```
=> SELECT row_to_json(disks) FROM disks;

                          row_to_json
---------------------------------------------------------------
 {"drive_type":"ssd","name":"Intel 760p Series","capacity":512}
 {"drive_type":"hdd","name":"Toshiba Canvio","capacity":3000}
(2 rows)
```

Соединить строки в общий JSON-массив можно, например, так:

```
=> SELECT json_agg(disks) FROM disks;

                             json_agg
---------------------------------------------------------------------
 [{"drive_type":"ssd","name":"Intel 760p Series","capacity":512}, +
  {"drive_type":"hdd","name":"Toshiba Canvio","capacity":3000}]
(1 row)
```

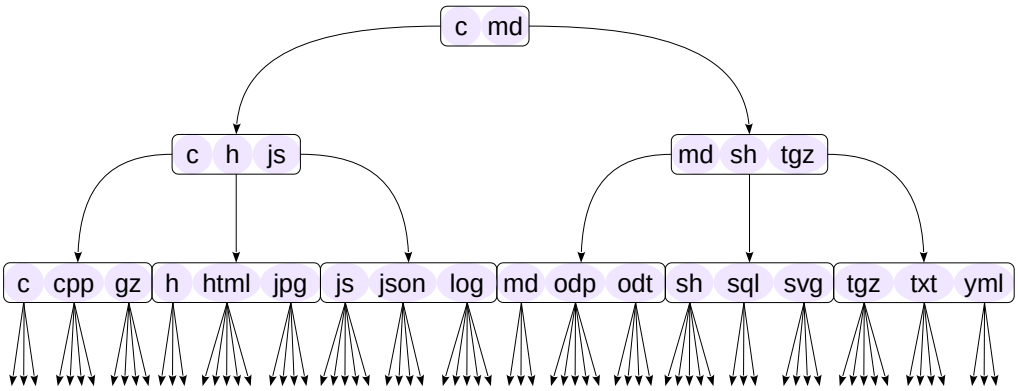
Здесь отдельные строки преобразуются в объекты JSON автоматически.

Метод доступа GIN

Инвертированный индекс

метод применим для сложносоставных значений (массив, текст)

API метода доступа определяет, как выделять элементы значения



11

Идея метода доступа GIN (general inverted index) основана на том, что для сложносоставных значений имеет смысл индексировать элементы значений, а не все значение целиком.

Представьте предметный указатель в конце книги. Страницы книги — сложносоставные значения (текст), а указатель позволяет ответить на вопрос «на каких страницах встречается такой-то термин?».

Для хранения элементов в GIN используется обычное B-дерево, поэтому элементы должны принадлежать к сортируемому типу данных. Основные отличия от B-дерева состоят в следующем:

- Когда нужно проиндексировать новое значение, это значение разбивается на элементы и индексируются сразу все элементы. Поэтому в индекс добавляется не один элемент, а сразу несколько (обычно много).

- Каждый элемент индекса ссылается на множество табличных строк.

- Хотя элементы и организованы в B-дерево, классы операторов GIN не поддерживают операции сравнения «больше», «меньше».

Таким образом, GIN оптимизирован для других условий использования, нежели B-дерево. (Но имеется расширение btree_gin, реализующее классы операторов GIN для обычных типов данных.)

https://postgrespro.ru/docs/postgresql/12/gin

https://postgrespro.ru/docs/postgresql/12/datatype-json#JSON-INDEXING

Метод доступа GIN для индексирования JSON

Пусть теперь таблица с дисками будет содержать JSON-документы.

```
=> DROP TABLE disks;
```

```
DROP TABLE
```

```
=> CREATE TABLE disks(
    id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    disk jsonb
);
```

```
CREATE TABLE
```

Заполним ее разными моделями, от 10 до 1000 Гб:

```
=> INSERT INTO disks(disk)
WITH rnd(r) AS (
    SELECT (10+random()*990)::integer FROM generate_series(1,100000)
),
dsk(type, model, capacity, plates) AS (
    SELECT 'hdd', 'NoName '||r||' GB', r, (1 + random()*9)::integer
    FROM rnd
)
SELECT row_to_json(dsk) FROM dsk;
```

```
INSERT 0 100000
```

```
=> ANALYZE disks;
```

```
ANALYZE
```

Вот что получилось:

```
=> SELECT * FROM disks LIMIT 3;
```

```
 id |                                  disk
----+-------------------------------------------------------------------------
  1 | {"type": "hdd", "model": "NoName 827 GB", "plates": 10, "capacity": 827}
  2 | {"type": "hdd", "model": "NoName 91 GB", "plates": 4, "capacity": 91}
  3 | {"type": "hdd", "model": "NoName 244 GB", "plates": 6, "capacity": 244}
(3 rows)
```

Сколько всего моделей имеют емкость 10 Гб и сколько времени займет поиск?

```
=> \timing on
```

```
Timing is on.
```

Оператор @? проверяет, есть ли в документе JSON заданный путь.

```
=> SELECT count(*) FROM disks WHERE disk @? '$ ? (@.capacity == 10)';
```

```
 count
-------
    47
(1 row)

Time: 52,755 ms
```

```
=> \timing off
```

```
Timing is off.
```

Как выполняется этот запрос?

```
=> EXPLAIN (costs off)
SELECT count(*) FROM disks WHERE disk @? '$ ? (@.capacity == 10)';
```

```
                          QUERY PLAN
---------------------------------------------------------------
 Aggregate
   ->  Seq Scan on disks
         Filter: (disk @? '$?(@."capacity" == 10)'::jsonpath)
(3 rows)
```

Конечно, используется полное сканирование таблицы — у нас нет подходящего индекса.

Документы JSONB можно индексировать с помощью метода GIN. Для этого есть два доступных класса операторов:

```
=> SELECT opcname, opcdefault
FROM pg_opclass
WHERE opcmethod = (SELECT oid FROM pg_am WHERE amname = 'gin')
AND opcintype = 'jsonb'::regtype;
```

```
    opcname     | opcdefault
----------------+------------
 jsonb_ops      | t
 jsonb_path_ops | f
(2 rows)
```

Класс по умолчанию, jsonb_ops, более универсален, но менее эффективен. Этот класс операторов помещает в индекс все ключи и значения. Из-за этого поиск получается неточным: значение 10 может быть найдено не только в контексте емкости (ключ capacity), но и как число пластин (ключ plates). Зато такой индекс поддерживает и другие операции с JSONB.

Попробуем.

```
=> CREATE INDEX disks_json_idx ON disks USING gin(disk);
```

```
CREATE INDEX
```

```
=> \timing on
```

```
Timing is on.
```

```
=> SELECT count(*) FROM disks WHERE disk @? '$ ? (@.capacity == 10)';
```

```
 count
-------
    47
(1 row)
```

```
Time: 7,651 ms
```

```
=> \timing off
```

```
Timing is off.
```

Доступ, тем не менее, ускоряется.

```
=> EXPLAIN (costs off)
SELECT count(*) FROM disks WHERE disk @? '$ ? (@.capacity == 10)';
```

```
                              QUERY PLAN
-----------------------------------------------------------------------
 Aggregate
   ->  Bitmap Heap Scan on disks
         Recheck Cond: (disk @? '$?(@."capacity" == 10)'::jsonpath)
         ->  Bitmap Index Scan on disks_json_idx
               Index Cond: (disk @? '$?(@."capacity" == 10)'::jsonpath)
(5 rows)
```

Другой класс операторов, jsonb_path_ops, помещает в индекс значения вместе с путем, который к ним ведет. За счет этого поиск становится более точным, хотя поддерживаются не все операции.

Проверим и этот способ:

```
=> CREATE INDEX disks_json_path_idx
ON disks USING gin(disk jsonb_path_ops);
```

```
CREATE INDEX
```

```
=> \timing on
```

```
Timing is on.
```

```
=> SELECT count(*) FROM disks WHERE disk @? '$ ? (@.capacity == 10)';
```

```
 count
-------
    47
(1 row)
```

```
Time: 0,663 ms
```

```
=> \timing off
```

```
Timing is off.
```

Так гораздо лучше.

Еще один вариант — построить индекс на основе B-дерева по выражению. Вот так:

```
=> CREATE INDEX disks_btree_idx ON disks( (disk->>'capacity') );
```

```
CREATE INDEX
```

```
=> \timing on
```

```
Timing is on.
```

```
=> SELECT count(*) FROM disks WHERE disk->>'capacity' = '10';
```

```
 count
-------
    47
(1 row)
```

```
Time: 0,717 ms
```

```
=> \timing off
```

```
Timing is off.
```

Но такой способ, конечно, менее универсален — под каждый запрос потребуется создавать отдельный индекс.

Сравним размер индексов (для сравнения выводится и размер таблицы):

```
=> SELECT indexname,
    pg_size_pretty(pg_total_relation_size(indexname::regclass))
FROM pg_indexes
WHERE tablename = 'disks'
UNION ALL
SELECT 'disks', pg_size_pretty(pg_table_size('disks'::regclass));
```

```
      indexname      | pg_size_pretty
---------------------+----------------
 disks_pkey          | 2208 kB
 disks_json_idx      | 1904 kB
 disks_json_path_idx | 1408 kB
 disks_btree_idx     | 2224 kB
 disks               | 13 MB
(5 rows)
```

Итоги

PostgreSQL поддерживает слабоструктурированные типы

Частичная поддержка XML

- формат теряет популярность

Полноценная поддержка JSON

- направление активно развивается

Индексирование на основе метода доступа GIN

Практика

1. Часть сведений о книгах (ISBN, аннотация и другие) выводятся приложением только на отдельной странице. Сейчас эти данные хранятся в таблице books, но не передаются приложению.
   Измените функцию, реализующую API приложения, так, чтобы сведения передавались в виде одного объекта JSON. Проверьте реализацию в приложении.
2. Многочисленные столбцы, хранящие дополнительные сведения о книгах, никак не используются в серверной части приложения. Замените их на один столбец, в котором сведения будут храниться в формате JSON. Сделайте это прозрачно для приложения.

1. Сведения должны передаваться в объекте JSON следующего формата:

```
{ "ISBN": isbn,
  "Аннотация": abstract,
  "Издательство": publisher,
  "Год выпуска": year,
  "Гарнитура": typeface,
  "Издание": edition,
  "Серия": series
}
```

2. Действуйте в предположении, что во время замены столбцов приложение работает и должно корректно выполнять запросы пользователей. Не забудьте, что вместе с заменой столбцов на один общий необходимо исправить и интерфейсную функцию приложения.

Кроме того, считайте, что одновременно с работой приложения могут изменяться как данные о существующих книгах, так и добавляться новые книги.

```
student$ psql bookstore2
```

Дополнительные сведения о книгах

Приложение рассчитывает получить дополнительные сведения о книгах в столбце additional таблицы, возвращаемой вызовом webapi.get_catalog, который, в свою очередь, обращается за данными к public.get_catalog. Для формирования дополнительных эта функция вызывает заглушку, которая всегда возвращает пустой документ:

```
=> \sf get_additional
```

```
CREATE OR REPLACE FUNCTION public.get_additional(book books)
 RETURNS jsonb
 LANGUAGE sql
 IMMUTABLE
AS $function$
    SELECT NULL::jsonb;
$function$
```

Изменим функцию следующим образом:

```
=> CREATE OR REPLACE FUNCTION public.get_additional(book books)
RETURNS jsonb
AS $$
    SELECT jsonb_build_object(
        'ISBN',         book.isbn,
        'Аннотация',    book.abstract,
        'Издательство', book.publisher,
        'Год выпуска',  book.year,
        'Гарнитура',    book.typeface,
        'Издание',      book.edition,
        'Серия',        book.series
    );
$$ LANGUAGE sql STABLE;
```

```
CREATE FUNCTION
```

Замена нескольких столбцов на один формата JSON

Добавим к таблице books необходимый столбец:

```
=> ALTER TABLE books ADD additional jsonb;
```

```
ALTER TABLE
```

Если данные о книгах могут изменяться в процессе наших действий, мы можем создать триггер, который будет переносить изменения в новый столбец:

```
=> CREATE FUNCTION fill_additional() RETURNS trigger
AS $$
BEGIN
    NEW.additional := jsonb_build_object(
        'ISBN',         NEW.isbn,
        'Аннотация',    NEW.abstract,
        'Издательство', NEW.publisher,
        'Год выпуска',  NEW.year,
        'Гарнитура',    NEW.typeface,
        'Издание',      NEW.edition,
        'Серия',        NEW.series
    );
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

```
=> CREATE TRIGGER books_additional
BEFORE INSERT OR UPDATE OF
    isbn, abstract, publisher, typeface, edition, series
ON books
FOR EACH ROW
EXECUTE FUNCTION fill_additional();
```

```
CREATE TRIGGER
```

Теперь перенесем в новый столбец исторические данные:

```
=> UPDATE books
SET additional = get_additional(books)

WHERE additional IS NULL;

UPDATE 96
```

Начиная с этого момента старые столбцы менять нельзя. Если бы у нас была процедура загрузки новых поступлений, ее следовало бы переделать на заполнение нового столбца.

Переключим приложение на использование нового столбца:

```
=> CREATE OR REPLACE FUNCTION public.get_additional(book books)
RETURNS jsonb
AS $$
    SELECT book.additional;
$$ LANGUAGE sql STABLE;

CREATE FUNCTION
```

Теперь можно удалить временный триггер:

```
=> DROP TRIGGER books_additional ON books;

DROP TRIGGER

=> DROP FUNCTION fill_additional;

DROP FUNCTION
```

И можно удалить ненужные теперь столбцы таблицы:

```
=> ALTER TABLE books
    DROP isbn,
    DROP abstract,
    DROP publisher,
    DROP year,
    DROP typeface,
    DROP edition,
    DROP series;

ALTER TABLE
```

Удаление столбцов происходит быстро, поскольку данные фактически не удаляются, а только становятся невидимыми. Поэтому размер таблицы не уменьшится до тех пор, пока строки не будет физически перезаписаны, например, при обновлениях.

Практика

1. В базе данных имеются таблицы пользователей и их заказов, связанные отношением «один ко многим».
   Напишите функцию, по идентификатору пользователя возвращающую документ JSON, содержащий все сведения о пользователе вместе с массивом его заказов.
2. В пользовательском интерфейсе необходимо выводить названия городов, хранящиеся в базе данных, на выбранном языке. Один из способов — хранить все переводы названий в объекте JSON в виде пар «код языка — название».
   Реализуйте такой способ. Для удобства создайте представление, показывающее название городов на языке, заданном в конфигурационном параметре сервера.

1. Таблицы могут быть следующими:

```
CREATE TABLE users(
  id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
  name text);

CREATE TABLE orders(
  id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
  user_id integer REFERENCES users(id),
  amount numeric);
```

Результат должен иметь вид:

```
{ "user_id": …,
  "name": …,
  "orders": [ { "order_id": …, "amount": … }, … ]
}
```

Для того чтобы сформировать объект JSON из нескольких полей, воспользуйтесь функцией jsonb_build_object.

2. Название пользовательского конфигурационного параметра должно содержать точку в своем имени, например, «translation.lang».

Получить текущее значение параметра можно функцией current_setting.

1. Формирование JSON

```
=> CREATE DATABASE ext_semistruct;
```

```
CREATE DATABASE
```

```
=> \c ext_semistruct
```

```
You are now connected to database "ext_semistruct" as user "student".
```

Создадим таблицы и тестовые данные:

```
=> CREATE TABLE users(
    id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    name text
);
```

```
CREATE TABLE
```

```
=> CREATE TABLE orders(
    id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    user_id integer REFERENCES users(id),
    amount numeric
);
```

```
CREATE TABLE
```

```
=> INSERT INTO users(name) VALUES ('alice'), ('bob'), ('charlie');
```

```
INSERT 0 3
```

```
=> INSERT INTO orders(amount, user_id)
    SELECT round( (random()*1000)::numeric, 2), u.id
    FROM users u, generate_series(1,3);
```

```
INSERT 0 9
```

Функция использует jsonb_build_object для создания объекта и jsonb_agg для создания массива:

```
=> CREATE FUNCTION get_users_w_orders(user_id integer) RETURNS jsonb
AS $$
SELECT jsonb_build_object(
    'user_id', u.id,
    'name',    u.name,
    'orders',  jsonb_agg(jsonb_build_object(
        'order_id', o.id,
        'amount',   o.amount
    ))
)
FROM users u
    JOIN orders o ON o.user_id = u.id
WHERE u.id = get_users_w_orders.user_id
GROUP BY u.id;
$$ LANGUAGE sql STABLE;
```

```
CREATE FUNCTION
```

Проверим:

```
=> SELECT jsonb_pretty(get_users_w_orders(1));
```

```
         jsonb_pretty
------------------------------
 {                            +
     "name": "alice",         +
     "orders": [              +
         {                    +
             "amount": 203.50,+
             "order_id": 1    +
         },                   +
         {                    +
             "amount": 867.77,+
             "order_id": 4    +
         },                   +
         {                    +
             "amount": 110.01,+
             "order_id": 7    +
         }                    +
     ],                       +
     "user_id": 1             +
 }
(1 row)
```

2. Хранение переводов

Таблица с городами:

```
=> CREATE TABLE cities_ml(
    id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    data jsonb
);
```

```
CREATE TABLE
```

```
=> INSERT INTO cities_ml(data) VALUES
    ('{ "ru": "Москва",
        "en": "Moscow",
        "it": "Mosca" }'),
    ('{ "ru": "Санкт-Петербург",
        "en": "Saint Petersburg",
        "it": "San Pietroburgo" }');
```

```
INSERT 0 2
```

Представление может быть устроено следующим образом:

```
=> CREATE VIEW cities AS
SELECT c.id,
       c.data ->> current_setting(
                      'translation.lang', /* missing_ok */true
                  ) AS name
FROM cities_ml c;
```

```
CREATE VIEW
```

Проверим результат с разными значениями параметра:

```
=> SET translation.lang = 'ru';
```

```
SET
```

```
=> SELECT * FROM cities;
```

```
 id |      name
----+-----------------
  1 | Москва
  2 | Санкт-Петербург
(2 rows)
```

```
=> SET translation.lang = 'en';
```

```
SET
```

```
=> SELECT * FROM cities;
```

```
 id |       name
----+------------------
  1 | Moscow
  2 | Saint Petersburg
(2 rows)
```

Таким образом приложение может установить язык один раз в начале сеанса, и затем запросы не будут зависеть от выбора пользователя.