Базы данных

Лекция 11 – Индексы

Преподаватель: Поденок Леонид Петрович, 505a-5

+375 17 293 8039 (505a-5)
+375 17 320 7402 (ОИПИ НАНБ)
prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by

Кафедра ЭВМ, 2024

2024.04.16

Оглавление

Индексы

Индексы — это средство увеличения производительности.
Пусть есть таблица:

```
CREATE TABLE tbl (
    id      integer,
    content varchar
);
```

и приложение выполняет много похожих запросов

```
SELECT content FROM test1 WHERE id = constant_value;
```

Если не предпринять специальных мер, сервер будет постоянно сканировать таблицу в поисках `id = constant_value`. Создание индекса по полю `id` меняет положение.

Пример — авторский индекс или тематический в сборнике конференции или в справочнике. Индекс можно создать/удалить командой:

```
CREATE INDEX имя ON имя_таблицы [USING тип] ({имя_столбца|(выражение)});
DROP INDEX имя
```

Создание индекса для вышеуказанной таблицы:

```
CREATE INDEX tbl_id_index ON tbl (id);
```

Индекс обновляется автоматически при всяких изменениях данных в таблице.

Индексы могут быть полезны также при выполнении команд **UPDATE** и **DELETE** с условиями поиска.

Кроме того, они могут применяться в поиске с соединением — индекс, определённый для столбца, участвующего в условии соединения, может значительно ускорить запросы с JOIN.

В целом индексы PostgreSQL можно использовать для оптимизации запросов, содержащих одно или несколько предложений **WHERE** или **JOIN** в следующем виде:

```
indexed-column indexable-operator comparison-value
```

indexed-column — это любой столбец или выражение, для которого был определён индекс.

indexable-operator — это оператор, который является членом класса операторов индекса для индексированного столбца;

comparison-value — любое выражение, которое не является изменчивым (not volatile) и не ссылается на таблицу с этим индексом.

Типы индексов

PostgreSQL поддерживает несколько типов индексов:
B-дерево;
хеш;
-GiST;
SP-GiST;
GIN;
BRIN;
bloom.

Для разных типов индексов применяются разные алгоритмы, ориентированные на определённые типы индексируемых предложений.

Команда CREATE INDEX по умолчанию создаёт индексы-B-деревья, эффективные в большинстве случаев. Выбрать другой тип можно, написав название типа индекса после ключевого слова USING.

Например, создать хеш-индекс можно так:

```
CREATE INDEX name ON table USING HASH (column);
```

B-дерево

Данные хранятся в отсортированной сбалансированной древовидной структуре.

B-деревья лучше всего подходят для поиска на основе диапазона, например, когда приложению необходимо найти все записи с ключами между некоторым начальным и конечным значением.

B-деревья также лучше используют окрестность ссылки. Если приложение может одновременно работать с ключами, расположенными рядом друг с другом, B-деревья работают очень эффективно.

Древовидная структура хранит ключи, которые находятся рядом друг с другом в памяти, поэтому для извлечения ближайших значений обычно не требуется доступ к диску.

Планировщик запросов PostgreSQL может задействовать индекс типа B-дерево, когда индексируемый столбец участвует в сравнении с одним из следующих операторов:

< <= = >= >

При обработке конструкций, представимых как сочетание этих операторов, например BETWEEN и IN, так же может выполняться поиск по индексу типа B-дерева. Кроме того, такие индексы могут использоваться и в условиях IS NULL и IS NOT NULL по индексированным столбцам.

B-деревья могут также применяться для получения данных, отсортированных по порядку. Это не всегда быстрее простого сканирования и сортировки, но иногда бывает полезно.

Есть вопросы с локалью.

Хеши

Хеш-индексы хранят 32-битный хеш-код, полученный из значения индексированного столбца, поэтому хеш-индексы работают только с простыми условиями равенства.

Планировщик запросов может применить хеш-индекс, только если индексируемый столбец участвует в сравнении с оператором **=**.

GiST

GiST-индексы представляют собой не просто разновидность индексов, а инфраструктуру, позволяющую реализовать много разных стратегий индексирования. Как следствие, GiST-индексы могут применяться с разными операторами, в зависимости от стратегии индексирования (класса операторов). Например, стандартный дистрибутив PostgreSQL включает классы операторов GiST для нескольких двумерных типов геометрических данных, что позволяет применять индексы в запросах с операторами:

```
<<    &<    &>    >>    <<|    &<|    |&>    |>>    @>    <@    ~=    &&
```

Эти операторы описаны в Разделе «Геометрические функции и операторы».

GiST-индексы могут оптимизировать поиск «ближайшего соседа», например такой:

```
SELECT * FROM places ORDER BY location <-> point '(101,456)' LIMIT 10;
```

<-> — расстояние между объектами.
Запрос возвращает десять расположений, ближайших к заданной точке.

Предикаты (bool foo(lh, rh))

<< — lh-объект строго слева от rh-объекта?
>> — lh-объект строго справа от rh-объекта?
&< — lh-объект не простирается правее rh-объекта?
&> — lh-объект не простирается левее rh-объекта?
<<| — lh-объект строго ниже rh-объекта?
|>> — lh-объект строго выше rh-объекта?
&<| — lh-объект не простирается выше rh-объекта?
|&> — lh-объект не простирается ниже rh-объекта?
@> — левый объект содержит правый?
<@ — левый объект содержится в правом?
~= — объекты совпадают?
&& — объекты пересекаются?

SP-GiST

Индексы SP-GiST, как и GiST, предоставляют инфраструктуру, поддерживающую различные типы поиска.

SP-GiST позволяет организовывать на диске самые разные несбалансированные структуры данных, такие как деревья квадрантов (quadtree), k-мерные и префиксные деревья.

Например, стандартный дистрибутив PostgreSQL включает классы операторов SP-GiST для точек в двумерном пространстве, что позволяет применять индексы в запросах с операторами:

```
<<    >>    ~=    <@    <<|    |>>
```

Индексы SP-GiST, как и GiST, поддерживают поиск ближайших соседей.

Для SP-GiST есть операторы, поддерживающие упорядочивание по расстоянию.

GIN

GIN-индексы представляют собой «инвертированные индексы», в которых могут содержаться значения с несколькими ключами, например массивы.

Инвертированный индекс содержит отдельный элемент для значения каждого компонента, и может эффективно работать в запросах, проверяющих присутствие определённых значений компонентов.

Подобно GiST и SP-GiST, индексы GIN могут поддерживать различные определённые пользователем стратегии и в зависимости от них могут применяться с разными операторами.

Составные индексы

Это индексы, созданные по нескольким столбцам таблицы. Например:

```
CREATE TABLE dev (
  major int,
  minor int,
  name  varchar
);
```

Допустим, что в таблице хранится содержимое каталога /dev и часто выполняются запросы вида:

```
SELECT name FROM dev WHERE major = константа AND minor = константа;
```

В этом случае имеет смысл определить индекс, покрывающий оба столбца major и minor:

```
CREATE INDEX dev_majmin_idx ON dev (major, minor);
```

В настоящее время составными могут быть только индексы типов B-дерево, GiST, GIN и BRIN.

Число столбцов в индексе ограничивается 32, включая столбцы INCLUDE (можно изменить при компиляции).

B-дерево

Составной индекс типа B-дерево может применяться в условиях с любым подмножеством столбцов индекса, но наиболее эффективен он при ограничениях по ведущим (расположенным левее) столбцам.

Сканируемая область индекса определяется условиями равенства с ведущими столбцами и условиями неравенства с первым столбцом, не участвующим в условии равенства.

Например, если есть индекс по столбцам (a, b, c) и условие

```
WHERE a = 5 AND b >= 42 AND c < 77
```

Индекс будет сканироваться от первой записи a = 5 и b = 42 до последней с a = 5. Записи индекса, в которых c >= 77, не будут учитываться, но тем не менее будут просканированы.

GiST

Составной индекс GiST может применяться в условиях с любым подмножеством столбцов индекса. Условия с дополнительными столбцами ограничивают записи, возвращаемые индексом, но в первую очередь сканируемая область индекса определяется ограничением первого столбца. GiST-индекс будет относительно малоэффективен, когда первый его столбец содержит только несколько различающихся значений, даже если дополнительные столбцы дают множество различных значений.

GIN

Составной индекс GIN может применяться в условиях с любым подмножеством столбцов индекса. В отличие от индексов GiST или B-деревьев, эффективность поиска по нему не меняется в зависимости от того, какие из его столбцов используются в условиях запроса.

Каждый столбец должен использоваться с операторами, соответствующими типу индекса.

Составные индексы следует использовать обдуманно. В большинстве случаев индекс по одному столбцу будет работать достаточно хорошо и сэкономит время и место.

Индексы по более чем трём столбцам вряд ли будут полезными, если только таблица не используется крайне однообразно.

Индексы и предложения ORDER BY

Обычно индекс используется при поиске строк для выдачи в результате запроса.

Однако, индексы также могут применяться для сортировки строк в определённом порядке. Это позволяет, не выполняя сортировку дополнительно, учесть в запросе предложение ORDER BY. Из всех типов индексов, которые поддерживает PostgreSQL, сортировать данные могут только B-деревья — индексы других типов возвращают строки в неопределённом, зависящем от реализации порядке.

Планировщик может выполнить указание ORDER BY, либо просканировав существующий индекс, подходящий этому указанию, либо просканировав таблицу в физическом порядке и выполнив сортировку явно.

Для запроса, требующего сканирования большой части таблицы, явная сортировка скорее всего будет быстрее, чем применение индекса, так как при последовательном чтении она потребует меньше операций ввода/вывода.

ORDER BY в сочетании с LIMIT n — при явной сортировке, чтобы выбрать первые n строк, потребуется обработать все данные, но при наличии индекса по столбцам в ORDER BY, первые n строк можно получить сразу, не просматривая остальные вовсе.

По умолчанию элементы B-дерева хранятся в порядке возрастания, при этом значения NULL идут в конце (для упорядочивания равных записей используется табличный столбец TID). Это означает, что при прямом сканировании индекса по столбцу `col` порядок оказывается соответствующим указанию `ORDER BY col` (или указав явно, `ORDER BY col ASC NULLS LAST`).

Индекс также может сканироваться в обратную сторону, и тогда порядок соответствует указанию `ORDER BY col DESC` (или указав явно, `ORDER BY col DESC NULLS FIRST`, так как для `ORDER BY DESC` подразумевается `NULLS FIRST`).

При создании индекса порядок сортировки элементов B-дерева можно изменить, добавив уточнения ASC, DESC, NULLS FIRST *и/или* NULLS LAST; например:

```
CREATE INDEX tbl_info_nulls_low ON tbl (info NULLS FIRST);
CREATE INDEX tbl_desc_index ON tbl (id DESC NULLS LAST);
```

Объединение нескольких индексов

При простом сканировании индекса могут обрабатываться только те предложения в запросе, в которых применяются операторы его класса и объединяет их оператор AND.

Например, для индекса (a, b) условие запроса

```
WHERE a = 5 AND b = 6
```

сможет использовать этот индекс, а запрос

```
WHERE a = 5 OR b = 6
```

нет.

Тем не менее, PostgreSQL способен объединять несколько индексов, в том числе многократно применять один индекс, а также охватывать также случаи, когда сканирования одного индекса недостаточно.

Система реализует условия AND и OR за несколько проходов индекса. Например, запрос WHERE col = 42 OR col = 47 OR col = 53 OR col = 99 можно разбивается на четыре сканирования индекса по col и сканированию для каждой части условия. Затем результаты сканирований будут логически сложены (OR) вместе и дадут конечный результат.

Другой пример — если есть отдельные индексы по cola и colb, запрос WHERE cola = 5 AND colb = 6 можно выполнить, применив индексы для соответствующих частей запроса, а затем вычислив логическое произведение (AND) для найденных строк, которое и станет конечным результатом.

Уникальные индексы

Индексы также могут обеспечивать уникальность значения в столбце или уникальность сочетания значений в нескольких столбцах. Если индекс создаётся как уникальный, в таблицу нельзя будет добавить несколько строк с одинаковыми значениями ключа индекса.

```
CREATE UNIQUE INDEX name ON table (column [, ...])
       [ NULLS [ NOT ] DISTINCT ];
```

В настоящее время обеспечение уникальности поддерживают только индексы типа B-дерево.

По умолчанию значения NULL в уникальном столбце считаются не равными друг другу, так что в таком столбце может быть несколько значений NULL. Параметр NULLS NOT DISTINCT изменяет это правило, так что значения NULL в индексе считаются равными друг другу.

Составной уникальный индекс не принимает только те строки, в которых все индексируемые столбцы содержат одинаковые значения.

Когда для таблицы определяется ограничение уникальности или первичный ключ, PostgreSQL автоматически создаёт уникальный индекс по всем столбцам, составляющим это ограничение или первичный ключ (индекс может быть составным).

Такой индекс и является механизмом, который обеспечивает выполнение ограничения.

Для уникальных столбцов не нужно вручную создавать отдельные индексы — они просто продублируют индексы, созданные автоматически.

Индексы по выражениям

Индекс можно создать не только по столбцу таблицы, но и по функции или скалярному выражению с одним или несколькими столбцами таблицы.

Это позволяет быстро находить данные в таблице по результатам вычислений.

Например, для сравнений без учёта регистра символов часто используется функция `lower()`:

```
SELECT * FROM table WHERE lower(col) = 'value';
```

Этот запрос сможет использовать индекс, определённый для результата функции `lower(col1)` так:

```
CREATE INDEX table_lower_col_idx ON table (lower(col));
```

Если мы объявим этот индекс уникальным (UNIQUE), он не даст добавить строки, в которых значения `col` различаются только регистром, как и те, в которых значения `col` действительно одинаковые.

Таким образом, индексы по выражениям можно использовать для обеспечения ограничений, которые нельзя записать как простые ограничения уникальности.

Если же часто выполняются запросы вида:

```
SELECT * FROM people
WHERE (first_name || ' ' || last_name) = 'John Smith';
```

тогда, имеет смысл создать следующий индекс:

```
CREATE INDEX people_names ON people ((first_name || ' ' || last_name));
```

Синтаксис команды CREATE INDEX обычно требует заключать индексные выражения в скобки, как показано во втором примере. Если же выражение представляет собой просто вызов функции, как в первом примере, дополнительные скобки можно опустить.

Поддержка индексируемых выражений обходится довольно дорого, так как эти выражения должны вычисляться при каждом добавлении строки и при каждом изменении без оптимизации (non-HOT update).

Однако при поиске по индексу индексируемое выражение не вычисляется повторно, так как его результат уже сохранён в индексе. В рассмотренных выше случаях система видит запрос как

```
WHERE indexedcolumn = 'constatnt'
```

поэтому поиск выполняется так же быстро, как и с простым индексом.

Таким образом, индексы по выражениям могут быть полезны, когда скорость извлечения данных гораздо важнее скорости добавления и изменения.

Частичные индексы

Частичный индекс — это индекс, который строится по подмножеству строк таблицы, определяемому условным выражением. Это выражение называется предикатом частичного индекса. Такой индекс содержит записи только для строк, удовлетворяющих предикату.

Частичные индексы в ряде ситуаций они могут быть очень полезны:

1) они позволяют избежать индексирования распространённых значений.

Поскольку при поиске распространённого значения (такого, которое содержится в значительном проценте всех строк) индекс всё равно не будет использоваться, хранить эти строки в индексе нет смысла. Исключив их из индекса, можно уменьшить его размер, а значит и ускорить запросы, использующие этот индекс. Это также может ускорить операции изменения данных в таблице, так как индекс будет обновляться не всегда.

Пример

Предположим, в базе данных хранится журнал обращений к корпоративному сайту компании. Большая часть обращений будет происходить из диапазона IP-адресов этой компании, а остальные могут быть откуда угодно, например, к нему могут подключаться внешние сотрудники с динамическими IP. Если при поиске по IP интересуют внешние подключения, IP-диапазон внутренней сети компании можно не включать в индекс.

```
CREATE TABLE access_log (
    url varchar,
    client_ip inet,
    ...
);
```

Создать частичный индекс для нашего примера можно так:

```
CREATE INDEX access_log_client_ip_idx ON access_log (client_ip)
    WHERE NOT (client_ip > inet '192.168.100.0' AND
               client_ip < inet '192.168.100.255');
```

Типичный запрос, использующий этот индекс:

```
SELECT * FROM access_log
WHERE url = '/index.html' AND client_ip = inet '212.78.10.32';
```

Указанный IP-адрес попадает в частичный индекс.

Следующий запрос не будет использовать частичный индекс, так как в нём IP-адрес не попадает в диапазон индекса:

```
SELECT * FROM access_log
WHERE url = '/index.html' AND client_ip = inet '192.168.100.23';
```

При таком определении частичного индекса необходимо, чтобы распространённые значения были известны заранее, так что такие индексы лучше использовать, когда распределение данных не меняется. Тем не менее, такие индексы можно пересоздавать время от времени, подстраиваясь под новое распределение, это значительно усложняет поддержку.

2) частичные индексы могут быть полезны тем, что позволяют исключить из индекса значения, которые обычно не представляют интереса.

Пример

Есть таблица, в которой хранятся и оплаченные, и неоплаченные счета, и при этом неоплаченные счета составляют только небольшую часть всей таблицы, но представляют наибольший интерес, производительность запросов можно увеличить, создав индекс только по неоплаченным счетам:

```
CREATE INDEX orders_unbilled_index ON orders (order_nr)
    WHERE billed is not true;
```

Этот индекс будет применяться, например в таком запросе:

```
SELECT * FROM orders WHERE billed is not true;
SELECT * FROM orders WHERE billed is not true AND amount > 5000.00;
```

Если неоплаченных счетов сравнительно мало, с таким частичным индексом можно выиграть при поиске неоплаченного счёта

В следующем запросе этот индекс использоваться не будет:

```
SELECT * FROM orders WHERE order_nr = 3501;
```

Счёт с номером 3501 может оказаться, как в числе неоплаченных, так и оплаченных.

3) Настройка частичного уникального индекса.

Предположим, что у нас есть таблица с результатами теста.

```
CREATE TABLE tests (
    subject text,
    target  text,
    success boolean,
    ...
);
```

Необходимо, чтобы для каждого сочетания предмета (subject) и целевой темы (target) была только одна запись об успешном результате, а неудачных попыток могло быть много.

Добиться этого можно следующим образом:

```
CREATE UNIQUE INDEX tests_success_constraint ON tests (subject, target)
    WHERE success;
```

Это подход будет особенно эффективным, когда неудачных попыток будет намного больше, чем удачных.

4) Не применяйте частичные индексы в качестве замены секционированию

Иногда возникает желание создать множество неперекрывающихся частичных индексов, например:

```
CREATE INDEX mytable_cat_1 ON mytable (data) WHERE category = 1;
CREATE INDEX mytable_cat_2 ON mytable (data) WHERE category = 2;
CREATE INDEX mytable_cat_3 ON mytable (data) WHERE category = 3;
...
CREATE INDEX mytable_cat_N ON mytable (data) WHERE category = N;
```

Но так делать не следует! Почти наверняка лучше использовать один составной индекс, объявленный так:

```
CREATE INDEX mytable_cat_data ON mytable (category, data);
```

При поиске в большем индексе может потребоваться опуститься на несколько уровней ниже, чем при поиске в меньшем частичном, но это почти гарантированно будет дешевле, чем выбрать при планировании из всех частичных индексов подходящий.

Сложность с выбором индекса объясняется тем, что система не знает, как взаимосвязаны частичные индексы, и ей придётся проверять каждый из них, чтобы понять, соответствует ли он текущему запросу.

Сканирование только индекса и покрывающие индексы

Все индексы в PostgreSQL являются вторичными – каждый индекс хранится вне области основных данных таблицы, которая в терминологии PostgreSQL называется кучей таблицы. Это значит, что при обычном сканировании индекса для извлечения каждой строки необходимо прочитать данные и из индекса, и из кучи. Элементы индекса, соответствующие заданному условию WHERE, обычно находятся в индексе рядом, а вот строки таблицы могут располагаться в куче произвольным образом. Таким образом, обращение к куче при поиске по индексу влечёт множество операций произвольного чтения кучи, которые могут обойтись дорого.

Чтобы решить эту проблему с производительностью, PostgreSQL поддерживает сканирование только индекса, при котором результат запроса может быть получен из самого индекса, без обращения к куче. Основная идея такого сканирования в том, чтобы выдавать значения непосредственно из элемента индекса, и не обращаться к соответствующей записи в куче. Для применения этого метода есть два фундаментальных ограничения:

1) Тип индекса должен поддерживать сканирование только индекса:

- индексы типа B-дерево поддерживают его всегда.
- индексы GiST и SP-GiST могут поддерживать его с одними классами операторов и не поддерживать с другими;
- остальные индексы такое сканирование не поддерживают.

Суть состоит в том, что индекс должен физически хранить или каким-то образом восстанавливать исходное значение данных для каждого элемента индекса. В качестве контрпримера, индексы GIN неспособны поддерживать сканирование только индекса, так как в элементах индекса обычно хранится только часть исходного значения данных.

2) Запрос должен обращаться только к столбцам, сохранённым в индексе.

Например, если в таблице построен индекс по столбцам x и y, и в ней есть также столбец z, то следующие запросы будут использовать сканирование только индекса:

```
SELECT x, y FROM tab WHERE x = 'key';
SELECT x FROM tab WHERE x = 'key' AND y < 42;
```

а эти запросы не будут:

```
SELECT x, z FROM tab WHERE x = 'key';
SELECT x FROM tab WHERE x = 'key' AND z < 42;
```

Если два этих фундаментальных ограничения выполняются, то все данные, требуемые для выполнения запроса, содержатся в индексе, так что сканирование только по индексу физически возможно.

В PostgreSQL существует и ещё одно требование, связанное с изолированностью транзакций — необходимо убедиться, что все извлеченные строки «видны» в снимке MVCC запроса[1].

Информация о видимости хранится не в элементах индекса, а только в куче, поэтому на первый взгляд может показаться, что для получения данных каждой строки всё равно необходимо обращаться к куче. И это в самом деле так, если в таблице недавно произошли изменения. Однако для редко меняющихся данных есть возможность обойти эту проблему.

---

1) MVCC (MultiVersion Concurrency Control) — один из механизмов СУБД для обеспечения параллельного доступа к базам данных, заключающийся в предоставлении каждому пользователю так называемого «снимка» базы, обладающего тем свойством, что вносимые пользователем изменения невидимы другим пользователям до момента фиксации транзакции.

PostgreSQL отслеживает для каждой страницы в куче таблицы, являются ли все строки в этой странице достаточно старыми, чтобы их видели все текущие и будущие транзакции. Это отражается в битах в карте видимости таблицы. Процедура сканирования только индекса, найдя потенциально подходящую запись в индексе, проверяет бит в карте видимости для соответствующей страницы в куче. Если он установлен, значит эта строка видна, и данные могут быть возвращены сразу. В противном случае придётся посетить запись строки в куче и проверить, видима ли она, так что никакого выигрыша по сравнению с обычным сканированием индекса не будет. И даже в благоприятном случае обращение к кучи не исключается совсем, а заменяется обращением к карте видимости; но так как карта видимости на четыре порядка меньше соответствующей ей области кучи, для работы с ней требуется много меньше операций физического ввода/вывода. В большинстве ситуаций карта видимости просто всё время находится в памяти.

Таким образом, тогда как сканирование только по индексу возможно лишь при выполнении двух фундаментальных требований, оно даст выигрыш, только если для значительной части страниц в куче таблицы установлены биты полной видимости. Но таблицы, в которых меняется лишь небольшая часть строк, встречаются достаточно часто, и этот тип сканирования весьма полезен на практике.

Чтобы эффективно использовать возможность сканирования только индекса, следует создавать покрывающие индексы. Такие индексы специально предназначены для включения столбцов, которые требуются в определённых часто выполняемых запросах. Так как в запросах обычно нужно получить не только столбцы, по которым выполняется поиск, PostgreSQL позволяет создать индекс, в котором некоторые столбцы будут просто «дополнительной нагрузкой», но не войдут в поисковый ключ. Это реализуется предложением INCLUDE, в котором перечисляются дополнительные столбцы.

Например, если часто выполняется запрос вида

```
SELECT y FROM tab WHERE x = 'key';
```

его можно ускорить, создав индекс только по x. Однако такой индекс:

```
CREATE INDEX tab_x_y ON tab(x) INCLUDE (y);
```

может удовлетворить такие запросы при сканировании только индекса, так как значение y можно получить из индекса, не обращаясь к данным в куче.

Поскольку столбец y не является частью поискового ключа, он не обязательно должен иметь тип данных, воспринимаемый данным индексом — включаемый столбец просто сохраняется внутри индекса и никак не обрабатывается механизмом индексации. Кроме того, в случае с уникальным индексом, например:

```
CREATE UNIQUE INDEX tab_x_y ON tab(x) INCLUDE (y);
```

условие уникальности распространяется только на столбец x, а не на x и y в совокупности.

Можно также добавить в ограничения UNIQUE и PRIMARY KEY предложение INCLUDE, что позволяет определить такой индекс альтернативным образом.

```
CREATE [ UNIQUE ] INDEX
    [ CONCURRENTLY ] [ [ IF NOT EXISTS ] name ] ON
    [ ONLY ] table_name [ USING method ]
    ( { column_name | ( expression ) } [ COLLATE collation ]
        [ opclass [ ( opclass_parameter = value [, ... ] ) ] ]
        [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [, ...] )
    [ INCLUDE ( column_name [, ...] ) ]
    [ NULLS [ NOT ] DISTINCT ]
    [ WITH ( storage_parameter [= value] [, ... ] ) ]
    [ TABLESPACE tablespace_name ]
    [ WHERE predicate ]
```

INCLUDE

Необязательное предложение INCLUDE определяет список столбцов, которые будут включены в индекс как *неключевые* столбцы.

Неключевой столбец не может использоваться в процессе поиска при сканировании индекса.

Неключевой столбец игнорируется при рассмотрении каких-либо ограничений уникальности или исключения, налагаемых данным индексом. Сканирование только по индексу может вернуть содержимое неключевых столбцов без необходимости посещения индексируемой таблицы, поскольку они доступны непосредственно из записи индекса. Таким образом, добавление неключевых столбцов позволяет использовать сканирование только по индексу для запросов, которые в противном случае не могли бы их использовать.

Следует проявлять осторожность при добавлении в индекс неключевых широких столбцов. Если кортеж индекса превышает максимальный размер, разрешенный для данного типа индекса, вставка данных не удастся.

В любом случае неключевые столбцы дублируют данные из индексируемой таблицы и увеличивают размер индекса, что потенциально замедляет поиск.

Столбцы, перечисленные в предложении INCLUDE, не требуют соответствующих классов операторов. Предложение может включать столбцы, типы данных которых не имеют классов операторов, определенных для данного метода доступа.

Выражения со включенными столбцами не поддерживаются, поскольку их нельзя использовать при сканировании только индекса.

В настоящее время эту функцию поддерживают методы доступа к индексам B-tree, GiST и SP-GiST. В этих индексах значения столбцов, перечисленных в предложении INCLUDE, включаются в конечные кортежи, соответствующие кортежам кучи, но не включаются в записи индекса верхнего уровня, используемые для навигации по дереву.