Finding foreign keys missing indexes

April 4, 2025 · 7 min · 1468 words · Robin Kåveland | Suggest Changes

▶ Table of Contents

Last week I was made aware that we had some foreign keys not backed by indexes in the system we're developing at work. Foreign keys in postgres must be backed by an index only on the side they refer *to*, not necessarily the side they refer *from*. Here's an example:

```
create table author(
    id bigint generated always as identity primary key,
    name text not null
);

create table book(
    id bigint generated always as identity primary key,
    author bigint not null references author(id),
    title text not null
);
```

In this example, there's a foreign key from the `book` table to the `author` table. Since `author` refers to a primary key in the `author` table, inserts into `book` can validate very quickly. There's no index on the `author` column in the `book` table though. The consequence of this is that `delete` on `author` must check every single row in `book` to check if it's safe to actually delete. The really annoying part of this is that the scan does not show up in query plans:

```
postgres=# insert into author(name) values ('Robin Hobb');
INSERT 0 1
postgres=# insert into book(author, title) values (1, 'Fool''s Quest');
INSERT 0 1
postgres=# explain analyze delete from author where id = 1337;
QUERY PLAN
------------------------------------------------------------------------------
Delete on author  (cost=0.15..8.17 rows=0 width=0) (actual time=0.180..0.181 rows=0
    ->  Index Scan using author_pkey on author  (cost=0.15..8.17 rows=1 width=6) (ac
```

```
     Index Cond: (id = 1337)
     Planning Time: 0.265 ms
     Execution Time: 0.321 ms
     (5 rows)
```

This query plan looks fine at a glance. It does not show the extra work that is necessary to validate referential integrity. It's easy to miss this problem when the load in production is sky-high and things aren't working. Since it can be challenging to predict how systems grow and change over time, it is often better to use some extra disk space and a few microseconds on insertion time to maintain an index. You'll never know the index is there, but at some poorly timed day in the future, you'll notice if it's not. If the index is never needed because the table never grows big, it's also an inexpensive index that stays small.

Working with `pg_catalog` to identify missing indexes

pg_catalog is a rich collection of system tables and views that we can inspect to learn things about the current database. We can easily retrieve the foreign keys in the current database by looking in pg_constraint:

```
postgres=# select conname, conrelid, conkey from pg_constraint where contype = 'f';
     conname      | conrelid | conkey
------------------+----------+--------
 book_author_fkey |   204852 | {2}
(1 row)
```

Here, `contype = 'f'` asks for foreign keys, `conname` is the foreign key name, `conrelid` is the id of the table that "owns" the foreign key constraint and `conkey` is a collection of columns the constraint uses. We can look up the columns in pg_attribute:

```
postgres=# select attnum, attname from pg_attribute where attrelid = 204852;
 attnum | attname
--------+----------
     -6 | tableoid
     -5 | cmax
     -4 | xmax
     -3 | cmin
     -2 | xmin
     -1 | ctid
      1 | id
      2 | author
      3 | title
```

```
(9 rows)
```

The `attnum` column goes with the `conkey` column, and there's a bunch of hidden columns with negative identifiers for the table too. As an aside, we can select those if we want to:

```
postgres=# select tableoid, xmin from book;
 tableoid | xmin
----------+-------
   204852 | 11038
(1 row)
```

But we normally don't need to know about them. Anyway, it is easy enough to collect all the column names used by a foreign key by inspecting these two system catalogs. For index information, we can consult pg_index. Let's look up the index information for the book table:

```
postgres=# select indexrelid, indkey
postgres-# from pg_index where indrelid = 204852 and indisvalid and indpred is null;
indexrelid | indkey
------------+--------
     204857 | 1
(1 row)
```

`indexrelid` is the system id of the index itself (helpful for finding its name in the `pg_class` catalog). The `indkey` column is a collection of columns the index refers to. We do some filtering here, if `indpred is not null` it means that the index is a partial index which may not be good enough for a foreign key.

Composite foreign keys

Note how `pg_constraint.conkey` and `pg_index.indkey` are both collections of column identifiers. That's because both foreign keys and indexes can contain many columns. Let's complicate matters a little bit to illustrate, by pretending we have some natural keys that consist of many parts.

```
create table country(
    id bigint generated always as identity primary key,
    country_code text not null
);

create table organization(
```

```
    name text not null,
    country bigint not null,

    primary key(name, country),
    foreign key(country) references country(id)
);

create table account(
    country bigint not null,
    organization text not null,
    email_id bigint not null,
    name text not null,

    primary key(country, organization, email_id),
    foreign key(country) references country(id),
    foreign key(country, organization) references organization(country, name)
);
```

Now we get these foreign keys:

```
postgres=# select conname, conrelid, conkey from pg_constraint where contype = 'f';
              conname               | conrelid | conkey
------------------------------------+----------+--------
 book_author_fkey                   |   204852 | {2}
 organization_country_fkey          |   204890 | {2}
 account_country_fkey               |   204902 | {1}
 account_country_organization_fkey |   204902 | {1,2}
(4 rows)
```

The interesting part here is really that `account_country_fkey` and `account_country_organization_fk` can be backed by the same index since they share a prefix (`{1}`). `pg_index` looks like this for the `account` table:

```
postgres=# select indexrelid, indkey
from pg_index where indrelid = 204902 and indisvalid and indpred is null;
 indexrelid | indkey
------------+--------
     204907 | 1 2 3
(1 row)
```

So the need for an index here is actually covered already. That's because the foreign key can use an index as long as it is a prefix of that index. The organization table is different:

```
postgres=# select indexrelid, indkey
from pg_index where indrelid = 204890 and indisvalid and indpred is null;
 indexrelid | indkey
------------+--------
```

```
     204895 | 1 2
(1 row)
```

This needs an index, because the index does not start with the same column as the foreign key. To summarize, we currently lack these indexes:

- The `author` column of the `book` table
- The `country` column of the `organization` table

There's just one little thing we need to know to write the query that connects `pg_constraint` and `pg_index` in the right way now – `pg_index.indkey` and `pg_constraint.conkey` are both 0-indexed collections. This runs counter to intuition since arrays are normally 1-indexed in postgres.

Connecting `pg_constraint` and `pg_index`

We want to do an anti-join – we want to keep the `pg_constraint` columns that are not matched by any row in `pg_index`. Let's break the query down with CTEs so it's manageable to look at:

```
with fks as (
    select conname, conrelid, conkey :: smallint[]
    from pg_constraint
    where contype = 'f'
), indexes as (
    select indrelid, indkey :: smallint[]
    from pg_index
    where indpred is null and indisvalid
)
select
    fks.conname,
    fks.conrelid,
    fks.conkey
from fks left join indexes on
    fks.conrelid = indexes.indrelid and
      -- foreign key must be prefix of index and both arrays are 0-indexed
      fks.conkey = indexes.indkey[0:array_length(fks.conkey, 1) - 1]
where indexes.indrelid is null; -- no index found
```

Running this gives us the expected two foreign keys:

```
          conname          | conrelid | conkey
---------------------------+----------+--------
```

```
 organization_country_fkey |   204890 | {2}
 book_author_fkey          |   204852 | {2}
(2 rows)
```

Drawing the rest of the FK-ing owl

But this leaves a lot of work left for whoever is using the output. We can use `pg_class` to fetch the table name, `pg_namespace` to fetch the schema name and `pg_attributes` to get the column names to make it a little nicer to look at the output. Let's get to work:

```
with fks as (
    select conname, conrelid, conkey :: smallint[]
    from pg_constraint
    where contype = 'f'
), indexes as (
    select indrelid, indkey :: smallint[]
    from pg_index
    where indpred is null and indisvalid
), fks_without_index as (
    select fks.conname, fks.conrelid, fks.conkey
    from fks
    left join indexes on
        fks.conrelid = indexes.indrelid and
        -- foreign key must be prefix of index and both arrays are 0-indexed
        fks.conkey = indexes.indkey[0:array_length(fks.conkey, 1) - 1]
    where indexes.indrelid is null -- no index found
)
select
    pg_namespace.nspname as schema_name,
    pg_class.relname as table_name,
    fk.conname as fk,
    array_agg(
      pg_attribute.attname
      -- order the column names by their position in the foreign key
      order by array_position(fk.conkey, pg_attribute.attnum)
    ) as columns
from fks_without_index fk
    join pg_class on fk.conrelid = pg_class.oid
    join pg_namespace on pg_class.relnamespace = pg_namespace.oid
    join pg_attribute on fk.conrelid = pg_attribute.attrelid
        and pg_attribute.attnum = any(fk.conkey)
group by schema_name, table_name, fk;
```

This gives us a nice output that is straightforward to look at, and we could also easily generate `create index` statements for the columns now:

```
 schema_name |  table_name  |            fk              |  columns
-------------+--------------+----------------------------+-----------
 public      | book         | book_author_fkey           | {author}
 public      | organization | organization_country_fkey | {country}
(2 rows)
```

Put it in CI and create the indexes it suggests, and it'll save you trouble and stress in the future.

Postgres Sql

« PREV
Running containers on no-ops linux in 2025

NEXT »
Constraint propagation: Mutual recursion for fun and profit