Deferrable SQL Constraints in Depth

August 27, 2017 **Newsletter ↳**

One strength of relational databases is their constant vigilance over data correctness. The user can declare constraints that their data must obey, and then leave it to the database to enforce the rules. This saves a lot of procedural application code and potential errors.

Automatic constraint enforcement is a powerful feature, and should be leveraged whenever possible. However there are times when it is convenient – and even necessary – to temporarily defer enforcement.

Table of Contents

Constraint Checking Granularities

As I mentioned in a previous article (https://begriffs.com/posts/2017-08-01-practical-guide-sql-isolation.html), transactions are the units of consistency, thus the SQL standard does not allow a transaction to commit with any constraint violations. While we can defer constraint checking to some degree, the buck stops at commit.

To be more precise, PostgreSQL supports three enforcement granularities:

1. **By row.** In this case a statement which updates multiple rows will fail immediately, possibly partway through its operation, when one of the rows violates a constraint.
2. **By statement.** In this case a statement is allowed to make its full set of changes before the database checks for violations.
3. **By transaction.** Any statement inside a transaction is free to violate constraints. However at commit time the constraints will be checked, and the transaction will fail if any constraints do not hold.

By default PostgreSQL checks each constraint with granularity type one or two, depending on the constraint's type. In the following table the NOT DEFERRABLE column lists the default for each constraint.

	NOT DEFERRABLE	INITIALLY IMMEDIATE	INITIALLY DEFERRED
CHECK	row	row	row
NOT NULL	row	row	row
UNIQUE	row	statement	transaction
PRIMARY KEY	row	statement	transaction
FOREIGN KEY	statement	statement	transaction
EXCLUDE	statement	statement	transaction

To change a constraint's validation granularity to the value in another column, we have to

explicitly declare a constraint as deferrable. Notice that some types of constraints won't budge. There's no way to defer CHECK and NOT NULL any later than per-row. (This is PostgreSQL behavior which violates the SQL standard.)

Before considering when/why to use deferrable constraints, let's see how they work in general. The first step is marking a constraint deferrable like this:

```
ALTER TABLE foo
  ADD CONSTRAINT foo_bar_fk
  FOREIGN KEY (bar_id) REFERENCES bar (id)
  DEFERRABLE INITIALLY IMMEDIATE; -- the magic line
```

Deferrable constraints give transactions flexibility. Any transaction can *choose* to defer checking `foo_bar_fk` to the end:

```
BEGIN;
-- defer the constraint
SET CONSTRAINTS foo_bar_fk DEFERRED;

-- ...
-- now we can do whatever we want to bar_id
-- ...

COMMIT; -- but it better be correct at this point
```

Additionally, we can approach it the other way and mark the constraint `DEFERRABLE INITIALLY DEFERRED`. In this mode the constraint will be checked per-transaction by default. If a transaction wants to *not* defer it, the transaction must say `SET CONSTRAINTS constraint_name IMMEDIATE`.

Without an explicit BEGIN command each statement runs in its own single-statement transaction where there is no difference between initially immediate or deferred. Attempting to defer constraints outside of a transaction will do nothing and warn, `WARNING: 25P01: SET CONSTRAINTS can only be used in transaction blocks`.

There’s one other important caveat. As the table above shows, declaring UNIQUE or PRIMARY KEY constraint `DEFERRABLE INITIALLY IMMEDIATE` actually changes it from per-row to per-statement checking. Even if a transaction does not choose to defer those constraints, their granularity is still subtly altered.

What’s the practical difference between per-row and per-statement checking? It’s best illustrated with an example.

```
CREATE TABLE snowflakes ( i int UNIQUE );

INSERT INTO snowflakes VALUES (1), (2), (3);
UPDATE snowflakes SET i = i + 1;
```

The UNIQUE constraint here is not deferrable, so by default an UPDATE statement will enforce it per-row and will raise an error.

```
ERROR:  23505: duplicate key value violates unique constraint "snowflake
s_i_key"
DETAIL:  Key (i)=(2) already exists.
```

If PostgreSQL had waited for all the rows to be updated (i.e. per-statement checking) thre would be no problem. The rows would be uniformly incremented and would end up unique again. However, as it is, PostgreSQL checks for violations immediately after incrementing i=1 to i=2, and at that moment the table contains 2, 2, 3.

Checking constraints per row is fragile and depends on the physical ordering of the rows. If we had inserted items in the opposite order (`INSERT INTO snowflakes VALUES (3), (2), (1)`) then the update would have worked.

To recap, declaring a constraint deferrable allows transactions to defer validation until commit time. Confusingly it also changes the semantics of some constraints even when no transaction chooses to defer them.

Why Defer?

Cyclic Foreign Keys

Sometimes you have to. The classic example is creating two items related by cyclic consistency checks. Consider,

```
CREATE TABLE husbands (
  id int PRIMARY KEY,
  wife_id int NOT NULL
);

CREATE TABLE wives (
  id int PRIMARY KEY,
  husband_id int NOT NULL
);

ALTER TABLE husbands ADD CONSTRAINT h_w_fk
  FOREIGN KEY (wife_id) REFERENCES Wives;

ALTER TABLE wives ADD CONSTRAINT w_h_fk
  FOREIGN KEY (husband_id) REFERENCES husbands;
```

Creating rows in either requires a row in the other to already exist. There's no way to get started because foreign keys are checked per statement and inserting into two tables requires two statements. What we *can* do is to defer the constraint.

```
ALTER TABLE husbands ALTER CONSTRAINT h_w_fk
  DEFERRABLE INITIALLY DEFERRED;

ALTER TABLE wives ALTER CONSTRAINT w_h_fk
  DEFERRABLE INITIALLY DEFERRED;
```

Because the cyclic constraints pretty much need to be deferred, we set this as their default behavior in a transaction. We can now run:

```
BEGIN;
INSERT INTO husbands (id, wife_id) values (1, 1);
INSERT INTO wives (id, husband_id) values (1, 1);
COMMIT;
-- and they lived happily ever after
```

What a neat and tidy textbook example. Only there's a **dirty little secret.** PostgreSQL has an alternative to deferrable foreign keys for making this example work! Check this out:

```
-- We won't need to defer them
ALTER TABLE husbands ALTER CONSTRAINT h_w_fk
  NOT DEFERRABLE;
ALTER TABLE wives ALTER CONSTRAINT w_h_fk
  NOT DEFERRABLE;

-- Instead do the two inserts as a single statement
WITH wife AS (
    INSERT INTO wives (id, husband_id)
      VALUES (2, 2)
  )
  INSERT INTO husbands (id, wife_id)
    VALUES (2, 2);
```

Common table expressions (CTEs (https://www.postgresql.org/docs/current/static/queries-with.html)) allow multiple queries to be considered as the same statement when checking constraints. Since foreign keys are checked per statement this approach works fine.

Although we found a workaround for deferring cyclic foreign keys, they're still a good way to start thinking about deferrable constraints.

Reallocating Items, One per Group

In this example, we have a set of school classes each with exactly one assigned teacher. Suppose we would like to switch the teachers of two classes. The constraints make it hard to do.

```
CREATE TABLE classes (
  id int PRIMARY KEY,
  teacher_id int UNIQUE NOT NULL
);

INSERT INTO classes VALUES (1, 1), (2, 2);
```

The CTE trick will not work for swapping teachers because a non-deferrable uniqueness constraint is checked by row rather than by statement.

```
-- fails unless deferrable since PostgreSQL
-- checks per row, not per statement
WITH swap AS (
    UPDATE classes
       SET teacher_id = 2
     WHERE id = 1
  )
UPDATE classes
   SET teacher_id = 1
 WHERE id = 2;

ERROR:  23505: duplicate key value violates unique constraint "classes_t
eacher_id_key"
DETAIL:  Key (teacher_id)=(1) already exists.
```

To swap teachers without deferring the uniqueness constraint on `teacher_id`, we would need to assign a temporary teacher to a class.

```
-- first assign a temporary teacher to free
-- teacher 1 to assignment to chass 2
UPDATE classes SET teacher_id = 999 WHERE id = 1;
UPDATE classes SET teacher_id = 1   WHERE id = 2;
UPDATE classes SET teacher_id = 2   WHERE id = 1;
```

Using a temporary teacher is a bit of a hack. It’s more natural to create the table with a deferrable constraint instead.

```
CREATE TABLE classes (
  id int PRIMARY KEY,
  teacher_id int NOT NULL UNIQUE
    DEFERRABLE INITIALLY IMMEDIATE
);
```

Then we can do a simple swap:

```
BEGIN;
SET CONSTRAINTS classes_teacher_id_key DEFERRED;

UPDATE classes SET teacher_id = 1 WHERE id = 2;
UPDATE classes SET teacher_id = 2 WHERE id = 1;
COMMIT;
```

Also a CTE approach with no explicit transaction will now work because a deferrable uniqueness constraint switches to per-statement checking.

Renumbering a List

One might model the position of todos todos in an ordered list using an integer position column:

```
CREATE TABLE todos (
  list_id int,
  position int,
  task text,
  PRIMARY KEY (list_id, position)
);

INSERT INTO todos VALUES
  (1, 1, 'write grocery list'),
  (1, 2, 'go to store'),
  (1, 3, 'buy items');
```

Each todo’s position is unique per list because of the compound primary key constraint.

Suppose we suddenly remember a new item for the top of the list, like “plan menus.” If we want the new item to have position 1 then we’ll need to shift the others down before inserting it. If you recall, this is the same basic problem as our earlier “snowflakes” example.

Marking the primary key as deferrable would fix the problem:

```
CREATE TABLE todos (
  list_id int,
  position int,
  task text,
  PRIMARY KEY (list_id, position)
    DEFERRABLE INITIALLY IMMEDIATE
);
```

This would have the effect of making PostgreSQL use per-statement semantics, with no actual `SET ... DEFERRED` needed in a transaction.

```
UPDATE todos
SET position = position + 1
WHERE list_id = 1;

INSERT INTO todos VALUES
  (1, 1, 'plan menus');
```

However the best solution isn’t deferrable constraints, it’s better data modeling. Re-orderable position shouldn’t be stored as integers but as fractions (rational numbers). That way there is always room to find a number between any other two. See my article User-defined Order in SQL (../posts/2018-03-20-user-defined-order.html). Correct modeling in this case will elegantly remove the need for deferring constraints.

Data Ingestion

Deferring constraints while loading data can make the process more convenient. For instance, deferred foreign keys allow tables to be loaded in any order, children before

parents. This can accommodate data arriving out of order, from the network perhaps.

Some articles online claim that deferred constraints allow more efficient bulk inserts. By my measurement this appears to be a **myth**. At commit time the same number of elements must be checked for consistency, and the query time balances out. To reproduce:

```
CREATE TABLE parent (
  id int PRIMARY KEY,
  name text NOT NULL
);

CREATE TABLE child (
  id int PRIMARY KEY,
  parent_id int REFERENCES parent
    DEFERRABLE INITIALLY IMMEDIATE,
  name text
);

INSERT INTO parent
SELECT
  generate_series(1,1000000) AS id,
  md5(random()::text) as name;
```

Now we insert five million child elements and measure the time to populate the data and check foreign keys. With a stock installation of PostgreSQL 9.6.3 on my wimpy laptop:

```
INSERT INTO child
SELECT
  generate_series(1,5000000) AS id,
  generate_series(1,1000000) AS parent_id,
  md5(random()::text) as name;

--
-- Time: 89064.987 ms
```

Trying again with the foreign key constraint deferred:

```
BEGIN;
SET CONSTRAINTS child_parent_id_fkey DEFERRED;

INSERT INTO child
SELECT
  generate_series(1,5000000) AS id,
  generate_series(1,1000000) AS parent_id,
  md5(random()::text) as name;
--
-- Time: 40828.810 ms

COMMIT;
--
-- Time: 47211.533 ms

-- Total: 88040.343 ms
```

What time we save in the insert we lose again in the commit.The only way to make the ingest faster is to temporarily *disable* the foreign key (assuming we know ahead of time that the data to be loaded is correct). That's generally a bad idea because we may end up taking more time to debug inconsistency down the line than it takes to do consistency checks during ingestion.

Reasons Not to Defer

Deferrable constraints seem great, right? Marking them deferrable simply gives us more options, so why not mark them all that way?

Query Planner Performance Penalty

The database query planner uses facts about the data to choose execution strategies that are correct and efficient. Its first duty is returning correct results, and it will use optimizations only when database constraints guarantee correctness. By definition deferrable constraints are not guaranteed to hold at all times, so they inhibit the planner.

To learn more about this I asked Andrew Gierth, RhodiumToad (http://blog.rhodiumtoad.org.uk/), on IRC who explained, “The planner can determine that a set of conditions on a table forces uniqueness of the result. If there’s a compatible unique, non-deferrable index it can elide an otherwise necessary sort/unique or hashagg step. It can’t do that for deferrable constraints, because there might be actual duplicate values present.”

He outlined two optimizations, one in PostgreSQL 9, and one in version 10. The older feature is join elimination:

```
CREATE TABLE foo (
  a integer UNIQUE,
  b integer UNIQUE DEFERRABLE
);

EXPLAIN
SELECT t1.*
FROM foo t1
LEFT JOIN foo t2
  ON (t1.a = t2.a);
```

Notice the join disappears, and it’s a simple sequential scan:

```
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on foo t1  (cost=0.00..32.60 rows=2260 width=8)
```

Whereas joining on `b`, which has a deferrable uniqueness constraint, does not eliminate the join.

```
EXPLAIN
SELECT t1.*
FROM foo t1
LEFT JOIN foo t2
  ON (t1.b = t2.b);
```

Result:

```
                              QUERY PLAN
----------------------------------------------------------------------
 Hash Left Join  (cost=60.85..124.53 rows=2260 width=8)
   Hash Cond: (t1.b = t2.b)
   -> Seq Scan on foo t1  (cost=0.00..32.60 rows=2260 width=8)
   -> Hash  (cost=32.60..32.60 rows=2260 width=4)
       -> Seq Scan on foo t2  (cost=0.00..32.60 rows=2260 width=4)
```

PostgreSQL 10 provides another optimization that turns the usual semi-join of an IN subquery into a regular join when the column of the subquery is guaranteed unique.

```
EXPLAIN
SELECT *
FROM foo
WHERE a IN (
  SELECT a FROM foo
);
```

It detects that `a` is unique:

```
                              QUERY PLAN
----------------------------------------------------------------------
-
 Hash Join  (cost=60.85..121.97 rows=2260 width=8)
   Hash Cond: (foo.a = foo_1.a)
   ->  Seq Scan on foo  (cost=0.00..32.60 rows=2260 width=8)
   ->  Hash  (cost=32.60..32.60 rows=2260 width=4)
         ->  Seq Scan on foo foo_1  (cost=0.00..32.60 rows=2260 width=4)
```

Let’s try it on `b`.

```
EXPLAIN
SELECT *
FROM foo
WHERE b IN (
  SELECT b FROM foo
);
```

The deferrable constraint prevents the optimization:

```
                                QUERY PLAN
-------------------------------------------------------------------------
-
 Hash Semi Join  (cost=60.85..124.53 rows=2260 width=8)
   Hash Cond: (foo.b = foo_1.b)
   ->  Seq Scan on foo  (cost=0.00..32.60 rows=2260 width=8)
   ->  Hash  (cost=32.60..32.60 rows=2260 width=4)
         ->  Seq Scan on foo foo_1  (cost=0.00..32.60 rows=2260 width=4)
```

In addition to helping the planner, non-deferrable constraints are required for target columns of foreign keys. In order to point a foreign key at a column with a uniqueness or primary key constraint, that constraint must be non-deferrable or else the database throws an error.

Harder Troubleshooting

Getting errors only after a bunch of statements finish makes troubleshooting harder. The resulting error does not pinpoint which statement caused the problem. You may or may not be able to determine the statement based on the message.

```
CREATE TABLE u (
  i int UNIQUE DEFERRABLE INITIALLY IMMEDIATE
);

BEGIN;
SET CONSTRAINTS u_i_key DEFERRED;

INSERT INTO u (i) VALUES (1), (2);
-- ... other statments
INSERT intu u (i) VALUES (2), (3);
-- ... other statements
INSERT intu u (i) VALUES (3), (4);

COMMIT;
```

At commit it says merely,

```
ERROR:  23505: duplicate key value violates unique constraint "u_i_key"
DETAIL:  Key (i)=(2) already exists.
```

In this case the message provides enough detail to trace it back to the offending statement, but it would be more difficult if the statements lacked literal values.

Not only might errors at commit time be confusing for people, they can mess up object relational mappers (ORMs). The mappers are often designed for simplistic database access and may not all be programmed with enough knowledge of the SQL spec to properly handle transaction-level constraint failures.

Also any work performed after a deferred constraint violation will ultimately be lost when the transaction rolls back. Deferred constraints can waste processing power.

Misleading Self-Documentation

Finally there's the matter of expressing our intentions through the database schema. In a perfect world there would be no performance penalties for using deferrable constraints and all constraints would all be be deferrable (initially immediate).

It’s an unfortunate implementation detail that we’re forced to choose manually between performance and flexibility. But given that the distinction exists, marking a constraint deferrable documents that deferring it is a worthwhile or necessary action. This is not the case for the vast majority of database constraints and labelling all as deferrable would hide the distinction.

Deferring by Column

Here’s one last trick for fun.

The `SET CONSTRAINTS` command acts on constraints by their names. However it can be convenient to defer eligible constraints by column instead. PostgreSQL’s information schema allows us to look up constraints by column.

```
-- A handy view

CREATE VIEW deferrables AS
SELECT table_schema, table_name, column_name,
       conname, contype
FROM
  pg_constraint,
  information_schema.constraint_column_usage
WHERE constraint_name = conname
  AND condeferrable = TRUE;

-- Defer all constraints for column

CREATE FUNCTION defer_col_constraints(
    t_name information_schema.sql_identifier,
    c_name name
  ) RETURNS void AS $$
DECLARE
  names text;
BEGIN
  names := (
    SELECT array_to_string(array_agg(conname), ', ')
    FROM deferrables
    WHERE table_name = $1
    AND column_name = $2
  );

  EXECUTE format(
    'SET CONSTRAINTS %s DEFERRED',
    names
  );
END;
$$ LANGUAGE plpgsql;
```

In our example loading parent and child data we could use this function to disable the child's foreign key constraint like so:

```
BEGIN;
SELECT defer_col_constraints('child', 'parent_id');

-- ...
COMMIT;
```

Related Posts

- PostgreSQL Domain Integrity In Depth (2017-10-21-sql-domain-integrity.html)
- Practical Guide to SQL Transaction Isolation (2017-08-01-practical-guide-sql-isolation.html)
- Faster PostgreSQL Counting (2016-10-12-count-performance.html)
- Relocatable PostgreSQL Builds (2016-05-21-relocatable-postgresql.html)
- Postgres Adores a Vacuum (2016-04-19-postgres-adores-vacuum.html)
- A Tour of PostgREST (2016-03-20-postgrest-tour.html)
- Database migrations without merge conflicts (2014-04-30-database-migrations-without-merge.html)