Published
July 19, 2022

Location
San Francisco

Article
Soft Deletion Probably Isn't Worth It

Find me on Twitter at **@brandur**.

Soft Deletion Probably Isn't Worth It

Contents

Anyone who's seen a couple different production database environments is likely familiar with the "soft deletion" pattern – instead of deleting data directly via `DELETE` statement, tables get an extra `deleted_at` timestamp and deletion is performed with an update statement instead:

```
UPDATE foo SET deleted_at = now() WHERE id = $1;
```

The concept behind soft deletion is to make deletion safer, and reversible. Once a record's been hit by a hard `DELETE`, it may technically still be recoverable by digging down into the storage layer, but suffice it to say that it's really hard to get back. Theoretically with soft deletion, you just set `deleted_at` back to `NULL` and you're done:

```
-- and like magic, it's back!!
UPDATE foo SET deleted_at = NULL WHERE id = $1;
```

Downsides: Code leakage

But this technique has some major downsides. The first is that soft deletion logic bleeds out into all parts of your code. All our selects look something like this:

```
SELECT *
FROM customer
WHERE id = @id
    AND deleted_at IS NULL;
```

And forgetting that extra predicate on `deleted_at` can have dangerous consequences as it accidentally returns data that's no longer meant to be seen.

Some ORMs or ORM plugins make this easier by automatically chaining the extra `deleted_at` clause onto every query (see `acts_as_paranoid` for example), but just because it's hidden doesn't necessarily make things better. If an operator ever queries the database directly they're even more likely to forget `deleted_at` because normally the ORM does the work for them.

Losing foreign keys

Another consequence of soft deletion is that foreign keys are effectively lost.

The major benefit of foreign keys is that they guarantee referential integrity. For example, say you have customers in one table that may refer to a number of invoices in another. Without foreign keys, you could delete a customer, but forget to remove its invoices, thereby leaving a bunch of orphaned invoices that reference a customer that's gone.

With foreign keys, trying to remove that customer without removing the invoices first is an error:

```
ERROR:  update or delete on table "customer" violates
    foreign key constraint "invoice_customer_id_fkey" on table "invoice"

DETAIL:  Key (id)=(64977e2b-40cc-4261-8879-1c1e6243699b) is still
    referenced from table "invoice".
```

As with other relational database features like predefined schemas, types, and check constraints, the database is helping to keep data valid.

But with soft deletion, this goes out the window. A customer may be soft deleted with its `deleted_at` flag set, but we're now back to being able to forget to do the same for its invoices. Their foreign keys are still valid because the customer record is technically still there, but there's no equivalent check that the invoices are also soft deleted, so you can be left with your customer being "deleted", but its invoices still live.

Pruning data is hard

The last few years have seen major advances in terms of consumer data protection like the roll out of GDPR in Europe. As such, it's generally frowned upon for data to be retained infinitely, which by default would be the case for soft deleted rows.

So you may eventually find yourself writing a hard deletion process which looks at soft deleted records beyond a certain horizon and permanently deletes them from the database.

But the same foreign keys that soft deletion rendered mostly useless now make this job more difficult because a record can't be removed without also making sure that all its dependencies are removed as well (`ON DELETE CASCADE` could do this automatically, but use of cascade is fairly dangerous and not recommended for higher fidelity data).

Luckily, you can still do this in systems that support CTEs like Postgres, but you end up with some pretty elaborate queries. Here's a snippet from one that I wrote recently which keeps all foreign keys satisfied by removing everything as part of a single operation:

```
WITH team_deleted AS (
    DELETE FROM team
    WHERE (
        team.archived_at IS NOT NULL
        AND team.archived_at < @archived_at_horizon::timestamptz
    )
    RETURNING *
),

--
-- team resources
--
cluster_deleted AS (
    DELETE FROM cluster
    WHERE team_id IN (
        SELECT id FROM team_deleted
    )
    OR (
        archived_at IS NOT NULL
        AND archived_at < @archived_at_horizon::timestamptz
    )
    RETURNING *
),
invoice_deleted AS (
    DELETE FROM invoice
    WHERE team_id IN (
        SELECT id FROM team_deleted
    )
    OR (
        archived_at IS NOT NULL
        AND archived_at < @archived_at_horizon::timestamptz
    )
    RETURNING *
),

--
-- cluster + team resources
--
subscription_deleted AS (
    DELETE FROM subscription
    WHERE cluster_id IN (
        SELECT id FROM cluster_deleted
    ) OR team_id IN (
        SELECT id FROM team_deleted
    )
    RETURNING *
)

SELECT 'cluster', array_agg(id) FROM cluster_deleted
UNION ALL
SELECT 'invoice', array_agg(id) FROM invoice_deleted
UNION ALL
SELECT 'subscription', array_agg(id) FROM subscription_deleted
UNION ALL
SELECT 'team', array_agg(id) FROM team_deleted;
```

The unabridged version of this is five times as long and includes a full 30 separate tables. It's cool that this works, but is so overly elaborate as to be a liability.

And even with liberal testing, this kind of query can still end up being a reliability problem because in case a new dependency is added in the future but an update to the query is forgotten, it'll suddenly start failing after a year's (or whatever the hard delete horizon is) delay.

Does undelete really work?

Once again, soft deletion is theoretically a hedge against accidental data loss. As a last argument against it, I'd ask you to consider, realistically, whether undeletion is something that's ever actually done.

When I worked at Heroku, we used soft deletion.

When I worked at Stripe, we used soft deletion.

At my job right now, we use soft deletion.

As far as I'm aware, never *once*, in ten plus years, did anyone at any of these places ever actually use soft deletion to undelete something. [1]

The biggest reason for this is that almost always, **data deletion also has non-data side effects**. Calls may have been made to foreign systems to archive records there, objects may have been removed in blob stores, or servers spun down. The process can't simply be reversed by setting `NULL` on `deleted_at` – equivalent undos need to exist for all those other operations too, and they rarely do.

We had a couple cases at Heroku where an important user deleted an app by accident and wanted to recover it. We had soft deletion, and theoretically other delete side effects could've been reversed, but we still made the call not to try because no one had ever done it before, and trying to do it in an emergency was exactly the wrong time to figure it out – we'd almost certainly get something wrong and leave the user in a bad state. Instead, we rolled forward by creating a new app, and helping them copy environment and data from the deleted app to it. So even where soft deletion was theoretically most useful, we still didn't use it.

Alternative: A deleted records table

Although I've never seen an undelete work in practice, soft deletion wasn't completely useless because we would occasionally use it to refer to deleted data – usually a manual process where someone wanted to see to a deleted object for purposes of assisting with a support ticket or trying to squash a bug.

And while I'd argue against the traditional soft deletion pattern due to the downsides listed above, luckily there's a compromise.

Instead of keeping deleted data in the same tables from which it was deleted, there can be a new relation specifically for storing all deleted data, and with a flexible `jsonb` column so that it can capture the properties of any other table:

```
CREATE TABLE deleted_record (
    id uuid PRIMARY KEY DEFAULT gen_ulid(),
    deleted_at timestamptz NOT NULL default now(),
    original_table varchar(200) NOT NULL,
    original_id uuid NOT NULL,
    data jsonb NOT NULL
);
```

A deletion then becomes this:

```
WITH deleted AS (
    DELETE FROM customer
    WHERE id = @id
    RETURNING *
)
INSERT INTO deleted_record
                (original_table, original_id, data)
SELECT 'foo', id, to_jsonb(deleted.*)
FROM deleted
RETURNING *;
```

This does have a downside compared to `deleted_at` – the process of selecting columns into `jsonb` isn't easily reversible. While it's possible to do so, it would likely involve building one-off queries and manual intervention. But again, that might be okay – consider how often you're really going to be trying to undelete data.

This technique solves all the problems outlined above:

- Queries for normal, non-deleted data no longer need to include `deleted_at IS NULL` everywhere.
- Foreign keys still work. Attempting to remove a record without also getting its dependencies is an error.
- Hard deleting old records for regulatory requirements gets really, really easy: `DELETE FROM deleted_record WHERE deleted_at < now() - '1 year'::interval`.

Deleted data is a little harder to get at, but not by much, and is still kept around in case someone needs to look at it.

[1] An ex-Stripe colleague just emailed me to say that at least in the old days, we would occasionally undelete customer records for users who'd gotten themselves into real trouble, but its use was discouraged, and rare. (Thanks OB!)

Article
Soft Deletion Probably Isn't Worth It

Published
July 19, 2022

Location
San Francisco

Find me on Twitter at **@brandur**.

Did I make a mistake? Please consider sending a pull request.