Goal: ~4,000-5,000 words

Hi, I’m Gary Sieling. I work on software products for the Pharma industry. For the past three or four years, I’ve worked on a product which fulfills some regulatory requirements for storing some of the data collected during clinical trials. It’s a fairly heavily used product, written in Java with Postgres as a backend. I’ve done a lot of engineering support for the product, which is what inspired this talk.

So, I can’t show the product but imagine you’re doing phone support for Netflix. Anyone who is a customer has probably noticed that they regularly change the selection of movies available.

To illustrate this, you can see I’ve swapped some in this list. There are various reasons for this – Netflix suddenly thinks you like period pieces, or they get into a contract dispute with one of their vendors.

As a support engineer, this presents a set of communication challenges, which inspire some technical solutions. Customers often ask questions like “why is this different than a week ago,” which is a reasonable question, but very time consuming to answer. I’ve found in our Pharma customers will often demand a lot more research and in-depth answers than our non-pharma clients, because no one wants to somehow find themselves responsible for a fine.

As an engineer, I imagine then a time-travelling debugger, that lets me see how these screens looked in the past. If we store each change in an audit trail in Postgres, we can simulate having a debugger with a long time frame. This lets us to generate timelines of data changes, validate data corrections, and build complex analytics on system usage.

We’re fortunate because the original customers of our system required audit history for each change tht happened and who did it – this lets us build some tooling around it. During the period where we built this software, the software development community has experienced a resurgence in interest in a programming style called functional programming. Functional programming has a few different principals, but one of these is writing code that transforms data from one form to another without side effects, which is a natural fit for SQL.

We’re going to go through a few different examples of things we can do with this data, and what I’ve learned about the design of audit trails and stored procedure coding styles from these things. For inspiration I’d like to look at version control software, shown here – this shows every change happening to a file, and diffs.

Looking at the functionality provided by git shows some things we might want to be able to do with our audit trail, as well as metadata we should store – a transaction Id, author, date, maybe other notes. Git lets you some other interesting things, like rolling back specific changes, tagging specific changes, or going back in time. If the value of tagging here is not obvious, the natural thing to look at would be noting when a software upgrade or schema migration is applied to the database, as this will correspond to when new defects appear.

This is a screenshot of a Wordpress auditing system I found. One of the things that wasn’t intuitively obvious to me looking at this is that you really always need multiple levels of audit trails – there is auditing at the level of a specific database change, and audit trails at the level of showing you specific business operations, which is what is shown here. The logs of business operations correspond to what users actually did, and help the most with communication issues, but whenever they fall apart, you need the second level.

Audit history typically provides a wealth of information, if you can sift through the noise and plan up front what you store. Postgres provides a number of built-in features that support sophisticated analysis, but they only work if you store enough up front.

[todo: maybe this should go later?] Depending on the nature of your system, you’ll have a few choices – if it’s a webapp, you’d want to know the a web server request ID, the user, time, and maybe the server of origin. Some systems are built around showing you historical data, like analytics packages, and some data storage systems only show deltas (this is how version control works).

In the application I support, every data change is captured using triggers, and stored in shadow tables. Each time a record changes, it’s saved, along with context for the operation: the user who executed the operation, the time, and whether the operation was an insert, update, or delete. If possible, it’s really useful to add a numeric revision number for a row, and a request ID that can tie together an operation across tables. [one single table for all audits vs many tables for audits – we have multiple]

Some ‘immutable data’ databases combine present and past state in the same relation, or store diffs rather than state. Our application uses a custom ORM layer to write queries against a fairly complex OLTP database that we generate mostly programmatically.

In our case, the shadow tables provide a natural partitioning – the application is only interested in current state, and support staff are primarily interested in history. Storing deltas would make operations like rollbacks easier to implement, but I do find ours easier to explain.

I like to keep audit triggers as simple as possible, I like to write stored procedures that pre-generate the queries in the simplest form, so there is no additional work required for the database when it comes to executing updates. When I do this in stored procedures, I use the format function in sql, which lets you treat big blocks of multi-line text as templates – this lets you write the query you you want, then stick in the variable blocks.

Assuming you control the code for the application accessing your database, there are a couple ways to inject context into a trigger: you can build a temp table that has request context, or use the application\_name context parameter. Application\_name is supposed to be used to identify an application vs. pgadmin or psql, but we’ve used it successfully to specify our application’s concept of users.

What we’ve built so far lets us build a very simple tool to replicate the ‘blame’ functionality of version control software. [I’ve written a more generic version sample procedures which are available on my github page]. This is notable because it uses named windows – you can tell Postgres to segment the data in the table by ID, sorted by date, then find out if a row changed.

Once you do this, you can stick the result in a common table expression – that’s the ‘with’ block at the top. Then, do the same operation again, but rank rows by change, only if they actually changed.

With these together, we can ‘simply’ filter to the row that changed, to find out who set a specific value.

Note that we haven’t done this per column – you can see it gets incredibly complex pretty fast.

Let’s consider a more involved example – rolling back a change.

We’ve found that we require infrequent corrections to data, where application defects to lead to values that shouldn’t exist, and “fixing” this carefully is key to unwinding the problem.

We can build a simple data correction that undoes a specific change in history, across tables, for instance a specific database transaction. To do this we have to find the change in the audit history tables, find out which columns changed, then change them back.

We need to provide the developer with the SQL for this. If we’re updating multiple tables, you need to do it in the order that supports foreign keys – if you’re missing some, the output of this would require manual intervention.

We also need to generate a report of what changed, to provide evidences that it doesn’t change more than necessary.

The full source to this solution is in my attached github page. Because this is somewhat complex, I’ve written the queries for single columns, then extracted pieces and formatted them – this shows the result for the query to target what to undo. The ‘where’ clause to this is generated ahead of time, depending whether you have a transaction id/user/time.

This creates a problem – this code is concise and easy to create, but unreadable. I like to put comments at the beginning of each query, partly to identify what they are, but it’s super useful when these show up in your logs, because anyone can interpret them. I’ve also introduced functionality to take a piece of SQL and replicate it once per column in a table.

I don’t have time to go through every piece of the rollback, but since I’ve written a lot of things like this, I’d like to mention how to test something like this. We make scrubbed copies of our production data for developers to test on, which lets you make sure it really works with the application. It’s important to remove sensitive data, and also email addresses so you can’t accidently sent out mass emails from your desktops. This means you can write a really simple report for the before/after part of a data correction – the numbers should match very closely in production.

The final analysis tool I want to discuss is in some ways the most complex. Let’s get back to the original question – what did our application show a week ago? I want to write a query in the form shown- Postgres has a time interval time, so if your data had effective time ranges, you could query for what was effective at a specific time.

To work this out, we’re going to write a view that shows the full information available about a movie over time – if our application has views that show present state for a movie already, we want it to be the same as those, so all we have to do is add a where clause to filter the time window.

This shows generating the effective date ranges for when a movie record was ‘effective’. The tsrange function creates the interval from the time of a change and the next change – the last argument shows that it is closed on one end and open on the other. If we are the most recent change, we say it’s effective until infinity.

In audit trail data, it’s possible to go back and add an ‘end date’ column, but I don’t like the idea of any modification of audit history.

If there is only one row, we say it’s effective forever.

We do this work for each table we care about and put each effective range in the join condition, then add a filter to force all the ranges to overlap (this is where the infinite ranges are really useful).

Now we can filter each time window to match the time specified. This is good, but inconvenient to hit each column.

Instead, we can write out this query in a stored procedure again, and use the stored procedure like a view, as shown.

For the full solution to this, we need to generate these – we have views like this with dozens of tables. No support person will generate something this long on their own, so by exposing some functionality we can give them a tool that enables them to do things they couldn’t otherwise.

If you’re interested in the time window joins, there is a mathematical formalism that describes time window operators called Allen’s interval operators. This defines a set of 13 mathematical operations which define what is possible**. [todo picture of the timeline thing]**

Having seen this much, it’s worth discussing performance.

We don’t index the audit data, because it’s rarely used. I’ve had problems where I had to wait 30 to 40 minutes to triage a production issue over a weekend, and it’s really painful.

If you make a mistake you lose an hour. The ideal situations to this are either to be allowed ad hoc access to add and drop indexes, which would ideally be carefully controlled so you don’t screw up a production system late at night.

There is a database that runs on the JVM called datomic, which builds immutable data in by default. Datomic is modelled off key-value store databases, and has four indexes, which are the permutations of accessing data by key, value, and attribute. Since they don’t allow changes, they probably have less overhead, but Postgres has no way to know that your audit trails behave as they do and apply optimizations.

If you get pinched in a situation like this, there are a few strategies you can take.

Every query will be a full table scan, followed by some collation of the results.

Once you accept that every query is going to do a full table scan, you can filter results as quickly as possible. For our audit case, you can filter to operations by particular users, times, or inserts, updates or deletes. Even filtering to just updates will save substantial time.

Most ‘not in’ style queries can be satisfied with set operations like ‘except’, which seems to be much, must faster. You can also use ‘except’ to prove that two queries are equivalent.

It’s also common for people to want to compare two sets of data, e.g. to figure out why something worked for one user and not another. I’ve noticed that a lot of people like to do this using a ‘not in’ subselect – any time you find yourself tempted to do this, you can always do it faster with set operations. Postgres allows you to find the difference or intersection of two sets, using a hashing algorithm, and the ‘except’ keyword.

In my case, the size of our data in audit tables ranges from double to thirty times the number of rows of the main data. The process where developers get production copies typically removes audit data unless requested.

As a final note with this, one of the dangers of using SQL only is that your support team won’t see whitespace problems – while this is a really minor problem from a technical perspective, I’ve found every technical and non-technical person who works on our team has been bitten by this at some point.

**Database version list**

**In conclusion….check out my open source page**

For anyone interested in activity streams, I’d investigate Heap Analytics – this is a Postgres backed tool analytics that tracks clickstreams. **[todo image]** I haven’t attempted to implement functionality like this, but if you need it, doing a trial run of their product exposes a lot of interesting use cases, like how to uniquely identify page elements.

**[todo: images of winmerge to list of diff]**