Goal: ~4,000-5,000 words

Traditional auditing requirements are a natural starting point for discussing problems around immutable data. This talk will demonstrate how historical data stored in Postgres allows our customer service team to simulate having a debugger with a long time frame, allowing us to generate timelines of data changes, validate data corrections, and build complex analytics on system usage.

Audit history typically provides a wealth of information, if you can sift through the noise. This talk discuss a variety of these issues: handling multiple overlapping time series, how to effectively filter data to points of interest, as well as performance and testing issues.

Postgres provides a number of built-in features that support sophisticated analysis, such as time range operators and window functions. Since Postgres allows easy inspection of the database schema, these features are far more powerful when used in dynamic sql generated in stored procedures.

Imagine you’re a DBA or engineer supporting a product like Netflix. A client calls up, and asks you a question: “A week ago this screen showed something different- what’s going on?” When software supports regulatory requirements, customers won’t take no for an answer, if they perceive a thread of non-compliance.

In this environment, a mismatch between a client’s expectations and a product could indicate defects requiring correction or a consequence of not using a product as intended. Figuring out how to disposition issues quickly is an important skill for front line support.

When I’ve been in this situation, I only dreamed of having a tool like I described – but we should be able to make it happen with audit history.

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.

Regardless of how you implement it, Postgres provides some useful tools. 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. However, if you prefer to put the logic in at runtime, Postgres exposes the nature of a trigger inside it’s code: whether it runs at a row or statement level, whether it’s an insert, update or delete, the table name, and trigger name – these are all really helpful if you’re doing a proof of concept and just want to get something to work.

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.

The software engineering community has recently experienced renewed interest in functional programming. At the simplest level this places value on writing code that transforms data without side-effects, and storing data in immutable structures. I was recently re-introduced to these concepts, and was struck by how many database techniques the functional programming community has used to support these goals.

Returning to the subject of support, the functionality we require is a mixture of data analysis and structured manipulation. Data analysis tools give you activity streams and big picture overviews of user behavior. 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.

For anyone interested in activity streams, I’d investigate Heap Analytics – this is a Postgres backed tool analytics that tracks clickstreams. 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.

Version control software provides a wide array of data analysis and manipulation tools. Git stores file diffs, but with our ‘store everything’ model we can easily re-create that structure to provide several useful features.

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

The simplest tool is a ‘blame’ tool, which shows us who made each modification to a column in the database. To compute this result, you need to find out whether a cell in a table was actually modified, then find out who made the most recent change. If you apply this function to each column independently, you’d have a lot of SQL – to simplify this, you can extract out most of the calculation logic for each column into a named window. **[todo: improve blame]**

If you’ve injected enough context information into your database, you can also generate queries to perform a rollback. In version control software, this applies the reverse of a diff to a file. This isn’t something you could enable anywhere – you need to test out that undoing a user’s change will create impossible scenarios from a business perspective. For me, the ideal tool will do its best, then give me queries as close as it can, that can be manually tweaked. **[todo: improve rollback]**

To make this work, you can simply generate UPDATE statements which reference only the columns that changed – this avoids issues with types. To make this work you need to be able to uniquely identify rows – you can clearly undo a change without this, but it’s so much easier if you can uniquely target a specific change.

If you want to undo a change that crosses many tables, you need to make the updates happen in the right order. This is also greatly simplified if you have the ability to target a specific request ID. To sort requests, you can use recursive SQL to sort using foreign key relationships. In my experience we periodically find a missing foreign key – if you’re in the situation where you’re forced to roll back a data change to production, you need to reduce risk. The best way to do that is to make adding the missing foreign key and fixing the data problem separate problems – you can do this by having your procedures generate SQL that you manually ‘fix’ as needed.

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?

We’ve discussed how to see these changes over time, across many tables. In our demo, we’re using a list of movies and their corresponding license information to control what is rendered to the user. As contracts change and metadata is corrected, this information changes (and quickly).

Our application provides many views that correspond roughly to what different screens require. Each view represents some entity and its associated metadata – consequently if we make a ‘movie’ view, we might include the effective date range when we’re allowed to play it to support what the application’s UI needs. The application uses these views by choosing the right view for a screen, then using an API to write out WHERE clause filters.

These views are generated – if we augmented them with effective date range information, we can expose views that can be filtered to specific times. While this isn’t too complex conceptually, our application has views that join 50+ (???) tables. Finding data that was effective at some point in time requires hundreds or thousands of lines of SQL – no support person will generate even a simplified version on their own, so by exposing some functionality we can give them a tool that enables them to do things they couldn’t otherwise.

Postgres allows us to create an interval datatype, and exposes a series of operations using it – while we could implement this without it, it makes things a lot easier.

Intervals can have open or closed ends, where a closed interval is a less than or equals, and an open interval would be a less than. To make two intervals touch without overlapping, you need to be closed on one end, and open on the other [this paragraph needs to go away]

Intervals don’t have to have ends- you can set an interval that runs from now 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, so I’ve chosen to demonstrate computing the results after the fact.

To find when a row is effective, we can find either the previous or next row with the lead or lag analytic functions to look at the history for a particular ID **[todo: test this is working],** and use that date. If that row doesn’t exist, we know it is the latest version, and we can set the effective range to infinity.

As shown, we can easily envision this for a single audit table – in fact, if you choose to modify history, this query is needless complexity.

If we combine movies and licenses, we now have two audit histories, each with their own effective ranges. It’s not enough to simple join across the foreign key – we must filter the join to only include records that were effective at the same time. If we don’t do this, we’ll see combinations of values in our view that never actually happened concurrently.

We can fix this join by forcing the time windows to overlap using the **‘[todo]**’ Postgres operator. If you’re interested in this subject, 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 defined the above operation, we know that all rows in our query were real outputs of a view at some point- the actual effective range for a row is the minimum of all ranges in the table.

There are two paths Postgres gives us to help here. We can use the ‘greatest’ and ‘least’ functions to create a new interval. **[todo picture of the timeline thing]**

Alternately, we can write a stored procedure that takes the desired time, applies it to each effective range, and returns a cursor. This allows us to write a stored procedure that has the same API as a view, with minimal additional effort.

**Performance**

If you suffer from any problems that resemble mine, you’ll need to consider performance at some point. 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.

We have a process where developers can get copies of production data scrubbed of sensitive data, like email addresses – this prevents them accidentally sending emails to clients through the application. This process typically explicitly excludes audit data, unless the developer requests it.

There is a database technology called Datomic that claims to provide a lot of the benefits I’m discussing, but running in the Java Virtual Machine. In their architecture, the data is modelled as a large key-value store, which supports a flexible arrangement of application objects, and a natural API, if you like programming in lisp/clojure. For indexing, they provide four default indexes, which allow access into data in various ways, depending whether you’re accessing objects by attribute names, values, or entity type.

I’m not sure what the storage overhead of this is in datomic, but if you did something similar for audit history on its own, the indexes would be costly [try this?]. There are a few strategies for handling this – if you’re writing data corrections, you could create an index for the time you need it, then drop it. You can copy the data to a copy of the database used for developers, and set ‘UNLOGGED’ on all the copies of audit data to prevent WAL archiving hits. Alternately, you could be patient, and write queries that are faster using full table scans.

I’ve stumbled across 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 in a query, you’ve now lost over an hour. I know a lot of people in this audience are in a DBA role, so you could probably add an index, but I’ve been in client situations where I couldn’t- and because of the space overhead of tons of indexes, we didn’t have them, nor would be able to justify adding them for later.

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.

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.

**Database version list**

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