Data Lineage → & Deletion

PALANTIR / SOFTWARE CUTTING — EDGE
FOUNDATIONAL SOFTWARE OF TOMORROW
DELIVERED TODAY.™

EST. 2003 / PALO ALTO, CA
HQ / DENVER, CO

Privacy & Civil Liberties

Granular Lineage-Aware Deletion in Palantir Foundry

While privacy legislation such as the GDPR (General Data Protection Regulation) and CCPA (California Consumer Privacy Act) highlight the importance of deleting data, adhering to personal data deletion policies in large-scale data systems can be complex and cumbersome. In such systems, raw data tends to quickly proliferate in original or derivative formats, but nonetheless is required to respect deletion periods designated by the purposes of collection. With each data transformation, potentially involving the combination with other data, the deletion periods become more difficult to reason about and account for.

A dependable deletion solution must find instances of sensitive data across multiple transformations and combinations of the data. Palantir's Foundry platform includes a deletion capability, which we call Lineage-Aware Deletion, that aims to do exactly this for our customers. This white paper outlines a technical description of how we have implemented this capability, and is meant to accompany our Lineage-aware Deletion Blog Post.

Contact Us
→ palantir.com/pcl/data-protection/deletion

Palantir

Think Deletion Before Creation: The Importance of Data Modeling

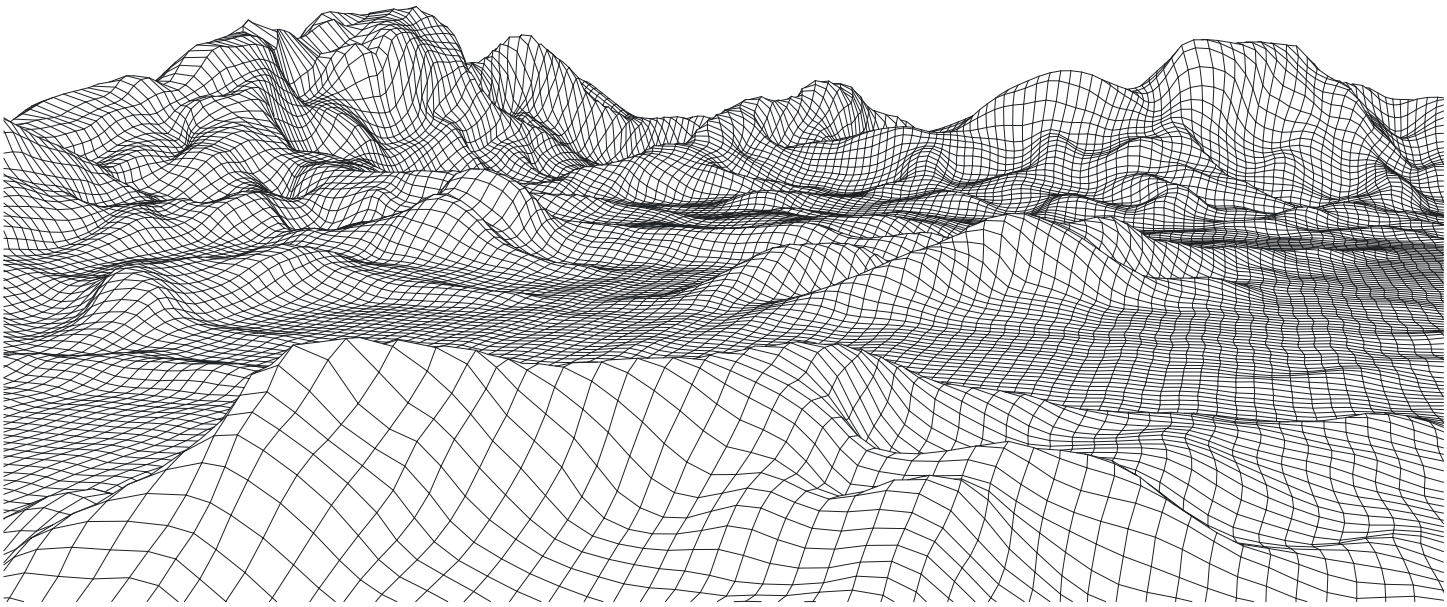
An often overlooked consideration in the management of deletion policies is the data modeling step that determines how data is ingested, transformed and processed. A thoughtful approach to deletion requires proactively modeling the data such that deletion procedures can be surgically applied at the appropriate granularity, minimizing impact on the overall data system. Doing this at the outset, before data is integrated, can save a lot of future complexity in managing deletion requirements. Some examples we have observed in practice are:

→ Only ingesting data that is necessary and dropping sensitive data columns that are not essential to the purpose of processing

→ Building data pipelines in a way that minimizes the replication of sensitive data in original and derivative formats, so that the deletion procedure impacts fewer datasets

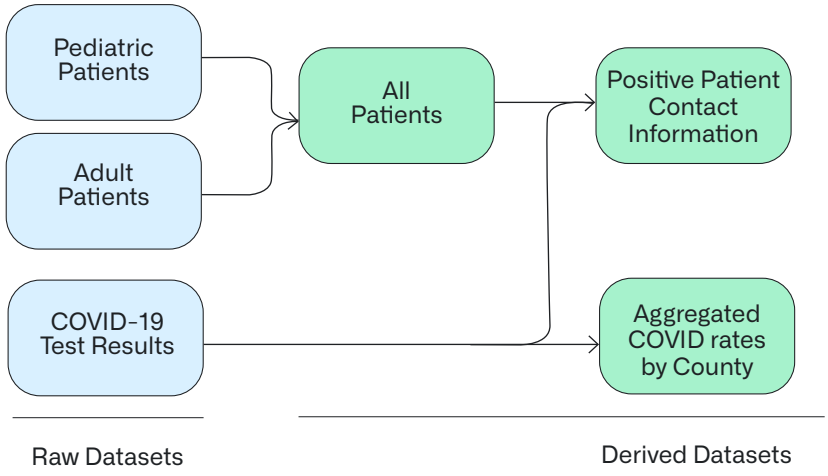
→ Adding relevant metadata about where sensitive data has been ingested and their purposes of processing, such that data engineers understand the impact of the data they are transforming

In Foundry, data ingestions can be tuned to run incrementally and at a suitable cadence such that the additions of new data to sensitive datasets are small enough to accommodate the desired deletion policy.



Mapping the Flow: Data Lineage

A typical data pipeline would involve the transformation and replication of data across multiple steps. This sequence of how data goes from its "raw" ingested form into subsequent "derivative" forms might also involve combination with other data. Mapping this flow in a legible manner is what we call the lineage of the data, and we further illustrate what this can look like with a notional example.



As data flows through the platform in this hypothetical example above, parent-child relationships are produced between datasets. For example, the "raw" Pediatric Patients and the Adult Patients are parent datasets of the "derived" All Patients, which is in turn a child dataset of both of them.

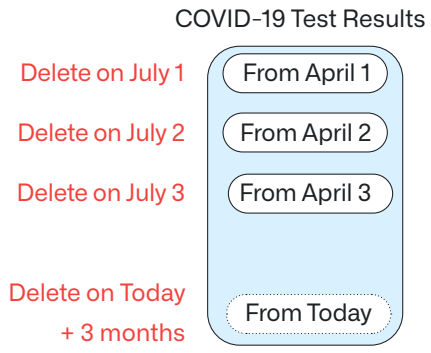
In Foundry, as data moves through the platform, the platform automatically keeps track of these parent-child relationships between datasets, which allows the whole platform to be "lineage-aware." This is a fundamental building block that allows users to ensure that when it is time for COVID 19 Test Results to be deleted, they can also delete Positive Patient Contact Information and, if we wanted, Aggregated COVID rates by County as well.

Granular Deletion

In the example above, COVID 19 Test Results isn't simply a static dataset. In reality, it would be a dynamic dataset, with new data coming regularly; as more tests are performed, more and more data is generated and held. But is it really necessary to store the individual granular data from testing events that happened months ago? Or at some point does it become sufficient to store the aggregated statistics, for example in Aggregated COVID rates by County? In the context of privacy protection, the latter is far preferable as it requires the storage of the minimal amount of data, without compromising the use case.

Doing this, however, requires more granular deletion — we can't just delete all of COVID 19 Test Results. Every chunk of data that enters the platform (in Foundry, we call this a "transaction") needs to be on a separate deletion schedule, depending on when it came in.

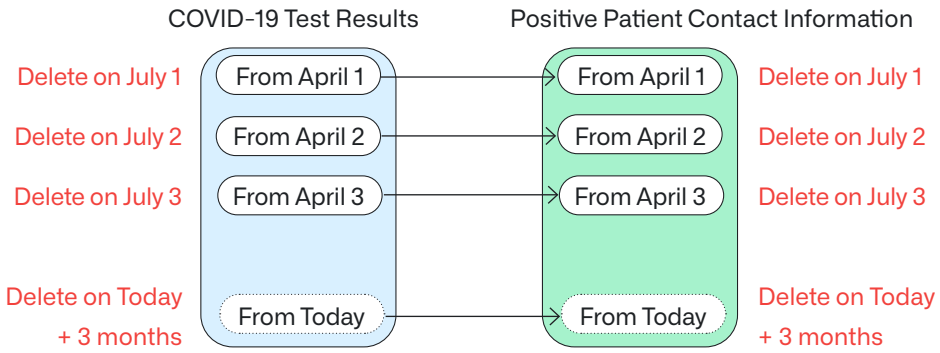
This concept can be illustrated with an example. Let's say we wanted to keep the actual COVID-19 related data for 3 months after it came in — after that point, it's not necessary to hold the granular deletion data for that long, as its only really relevant for a fixed time period after the test was administered. Essentially, we intend to give each "transaction" of data a "time to live" (TTL) of 3 months. Below, we see the deletion dates for transactions of test result data that came in each day starting April 1. With a 3 month TTL, we expect them to be deleted starting July 1, one day at a time.



Palantir

Granular Deletion

This granular level of deletion allows us to preserve the more current and relevant portions of the data, without keeping data that we no longer need. The deletion dates also extend to descendant datasets, for example, to Positive Patient Contact Information, as shown below.



This sort of granular deletion is supported in Palantir Foundry, as the platform tracks the lineage of each of the transactions within a dataset.

Data Deletion System Principles

Over time, we've found that a working deletion solution needs to adhere to certain key principles if it is to be successful in a data platform that has users:

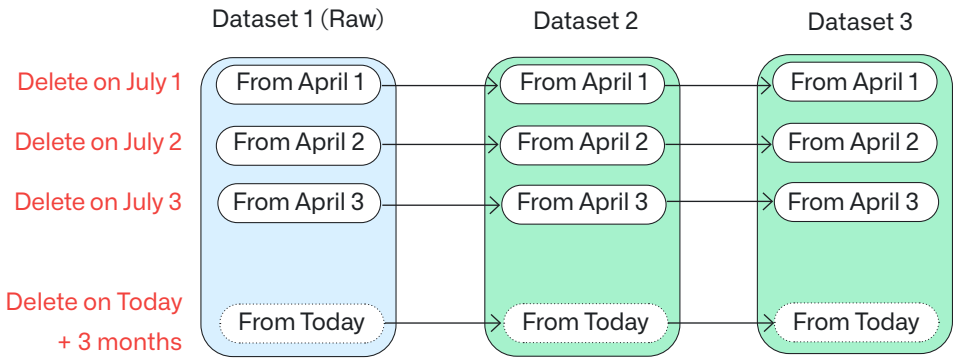
1. Correctness — All data scheduled for deletion should be deleted at the appropriate time, and data not scheduled for deletion should not be deleted.
2. Transparency — Users should be able to know when data is going to be deleted and why it is going to be deleted.
3. Verifiability — Confirmation that data was successfully deleted, including capturing the reasons/a trail of why it was deleted.
4. Efficiency — We don't want to be be re-computing large-scale datasets in the platform for a small deletion in a source dataset. Re-computing datasets, especially large ones, takes a significant amount of compute time and therefore adds to the overall cost of deletion.

For our purposes we've also explored above the ideas of lineage-awareness and granularity — we don't include them in the list above, as that list is meant to be generic for any deletion solution. However, in the deletion solution in Foundry, lineage-awareness and granularity are both important facets that we'll continue to explore below.

Palantir

Original "Recursive" Design & Subsequent Complications

The initial intuition was to first design our Lineage-Aware Deletion solution by only requiring a deletion date on raw datasets. This allows us to traverse the lineage graph to identify datasets to be deleted. Conceptually, this represents a simple and elegant “recursive” solution. In the example below, this would mean that we only placed deletion dates on the raw ancestor transactions, and NOT on the descendant child transactions. Then, on the day of deletion, Foundry would traverse the lineage arrow(s) recursively, and a full set of resources to delete would be derived.



In the example above, on July 1, we would know we had to delete the April 1 transaction from Dataset 1 (Raw), and then we would traverse the lineage arrows to know we had to delete the April 1 transactions from Datasets 2 and 3. These 3 transactions, in this case, would be our entire “deletion graph” that we would submit for deletion.

Complications

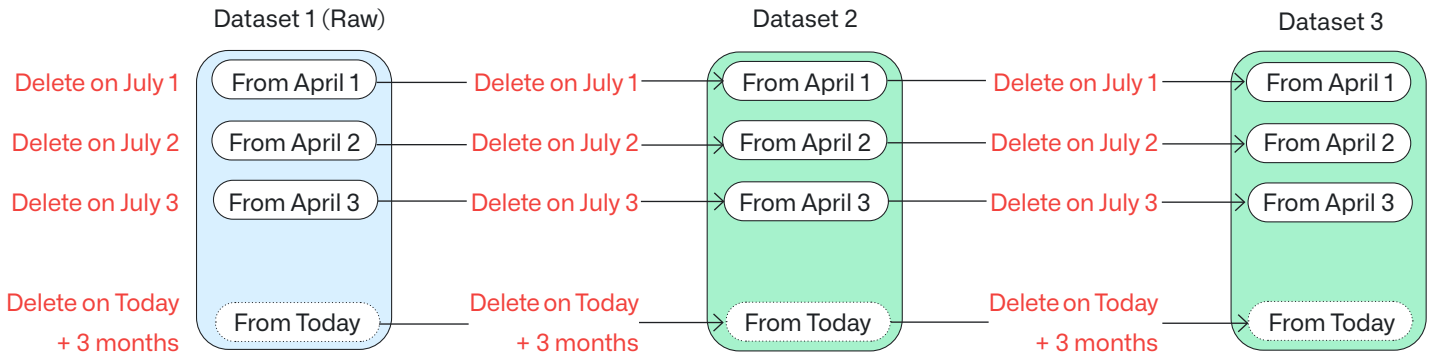
We soon realized that this wasn’t strictly always *correct*. Between the time we derived the “deletion graph” and the time we executed the deletion on July 1, a new piece of data, say in a new Dataset 4, could be derived from existing data in Dataset 3. This freshly derived data in Dataset 4 would not show up in our “deletion graph”, because it didn’t exist when we were computing our deletion graph, and therefore would not be flagged for deletion. Resolving this correctness issue would require the system to transactionally lock down all the data in the deletion graph, such that no new data could be created from it, a feat that is near-impossible to achieve with a micro-service architecture.

The second thing we realized was that this design didn’t lend itself to transparency of deletion. In order to find out what was to be deleted in the next N days, we would be be forced to traverse the entire data graph starting from every raw dataset that had a deletion date — an expensive affair for every such user request.

Palantir

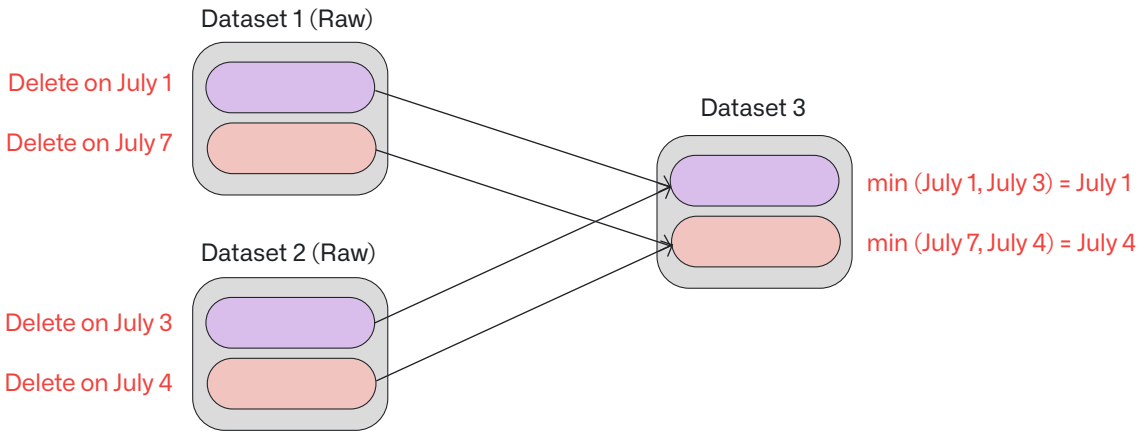
Updated "Declarative" Design with Deletion Policies

We managed to improve this by switching to a “declarative” design, where every transaction had its own deletion date through a deletion policy.



This means that when a deletion date is applied to a transaction on an ancestor dataset, we have to trigger an asynchronous background job that eventually updates the deletion dates of all of its descendant transactions. When an ancestor transaction deletion date is updated, we again have to trigger an asynchronous background job that does something similar.

Additionally, when a new descendant transaction is committed, that new transaction needs to adopt the minimum deletion date of its parents. We show an example of this below.



Tradeoffs

All the bookkeeping we have to do requires rigor to sequence and perform, but it is not without reward. Both our problems with the recursive design — the lack of correctness and transparency — are remediated with this design.

Now, instead of building out a "deletion graph" as in the recursive case, every transaction has its own deletion date. Because we're storing these dates proactively, this allows us to answer the question, "What is going to be deleted in the next N days" without traversing through all the resources in the platform every time a user wants to make such a request. It also gives us stronger correctness guarantees, as we only delete data that we've proactively applied a deletion date onto, rather than building the "deletion graph" at deletion time.

Deletion Policies

Now that we know how deletion dates flow through Foundry via lineage, we have to know how those deletion dates come to be in the first place. This is done by having a primitive in the platform, known as a deletion policy.

Deletion policies are applied at the dataset level (not the transaction level), and describe a paradigm for assigning deletion dates. There are 2 types of policies:

1. Time to Live (TTL) Policy — If applied on a dataset, every transaction of data in that dataset must be deleted some fixed amount of time after that transaction was created.
2. Fixed Deletion Date Policy — If applied on a dataset, all transactions of data will be deleted on a fixed date specified in the policies.

Policies can be applied on any dataset, but generally are most useful on raw datasets. If a descendant dataset has ancestors with different policies, the minimum deletion date is evaluated for each transaction in that descendant dataset.

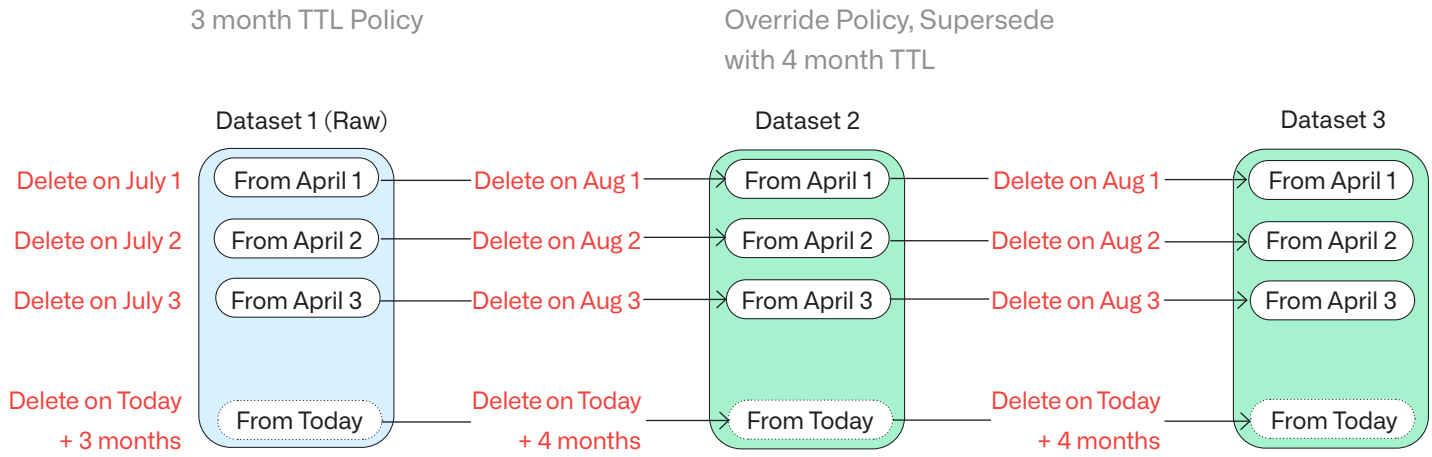
Override Policies

Users that have certain elevated permissions can set override policies on descendant dataset. An override policy is a policy that essentially "severs" the influence of all ancestor deletion policies on the transactions of that dataset. Like with any policy, once an override policy is applied on a dataset, all descendants of that dataset are also impacted by it.

These types of policies are quite useful when certain derived data no longer has data that requires deletion. For example, if a downstream dataset performs an aggregation over sensitive data, the resulting aggregated data may no longer be sensitive, and therefore need not be subject to a stringent deletion policy.

An override dataset can also be specified with an optional superseding policy — a new policy to subject the downstream dataset to.

Palantir



Safeguarding the Use of Policies

The configuration of deletion policies is typically regarded as a privileged action that needs to be applied carefully and accountably. This is because any changes to deletion policies can have significant consequences to the integrity of data, such as keeping data longer than is legally required on one hand, or accidental data loss on the other hand.

Foundry's lineage-aware deletion solution is integrated into Foundry's robust access control system, which allows only specific trusted users to have the privilege of applying policies. Additionally, a user must have an even higher level of access within this access control system to apply override policies.

Furthermore, it is also integrated with our Checkpoints service to require individuals to justify the configuration of deletion policies. This means that even for individuals with the permissions to apply or remove a deletion policy, Checkpoints can require them to justify their rationale and the justifications can be reviewed in-platform with attribution.