Implementing Purpose Limitation for Pelton

Amrit Singh Rana
Brown University

Abstract

Privacy laws like the GDPR lay down guidelines for web applications about collection and use of personal data. GDPR says that personal data of users has to be collected for a purpose and used for the stated purpose only.

Developers write custom scripts and store metadata to comply with this law. This manual labor is expensive and error-prone. While large companies have the resources to modify their systems to comply, it is difficult and expensive for small companies to do that and non-compliance can lead to steep fines.

Pelton is a database system that complies with privacy laws by construction. Each user has their own micro-database (μDB) that contains all their data. Pelton uses in-memory materialized views that collate data to make web application requests fast even though the storage is split into many μDBs.

By extending the "compliance-by-construction" principles of Pelton, this system implements fine-grained purpose limitation for materialized views. We demonstrate that the proposed system achieves this with minimal developer effort and incurs minimal performance overheads.

1 Introduction

The European General Data Protection Regulation (GDPR) lays down guidelines for processing personal data that "data controllers" are required to follow. One of these guidelines is purpose limitation. GDPR's Article 5(1b) states:

> Personal data shall be collected for specified, explicit and legitimate purposes and not further processed in a manner that is incompatible with those purposes; further processing for archiving purposes in the public interest, scientific or historical research purposes or statistical purposes shall, in accordance with Article 89(1), not be considered to be incompatible with the initial purposes ('purpose limitation')

Compliance to the above stated article is important as non-compliance risks steep fines. Implementing purpose limitation manually is expensive and error prone. Consider a case where in an organization, multiple teams query a database for different purposes. The product team queries the database to populate views, the advertising team queries the database for ad targeting and the analytics team queries the database for business insights. In such a setting, manual scripting can be inefficient and can lead to unintentional flow of information. We need a system that offers purpose limitation out-of-the-box.

The core idea of our solution is to store the purpose of collection of each record when it is created and do not let a record leave the database for any purpose other than that. Developers can specify one or more purposes for each record. While querying, developers need to specify the purpose of the query. The framework will intercept and modify the query so that it returns only those records that are permitted to be used for the asked purpose. A query can have only one specific purpose.

Since querying on multiple μDBs can be lead to high performance costs, Pelton offers Materialized Views. Materialized Views are in-memory stored query results that have data of multiple users stored together. Materialized views offer low latency responses like memcached or Redis caching layer provides.

Pelton has well-defined heuristics that dictate how a query executes. In some cases, queries need to be sent to all μDBs. For this project, we restrict our scope to implementing purpose limitation for queries to materialized views.

When data for one user is read, it can be easily read directly from the user's μDB. Implementing purpose limitation becomes necessary when data of multiple users is located in close proximity. In such situations, errors

can creep in and lead to unintentional flow of records. In Pelton, only materialized views contain data of multiple users in a single table. This is the reason that we are attempting to implement purpose limitation for materialized views first.

This solution assumes Pelton's threat model that the developer is honest.

Since this framework intercepts and modifies queries before they execute, we expect it to have some performance cost. This framework also stores "purpose" with each data record. This will add overheads to the memory (materialized views) and disk (μDBs, where actual data resides) usage. We will show that these overheads are reasonable.

2 Design

2.1 Pelton Overview

Pelton stores data primarily by user while offering a conventional interface to the database that looks like one database schema. Instead of storing all the data in a single database, Pelton maintains a collection of independent, per-user micro-databases (μDBs), that store data related to single user. The application writes to μDBs, but Pelton satisfies reads that require data from multiple μDBs from materialized views derived from the μDBs. The overall database contents are the union of the contents of all the μDBs.

2.2 Materialized Views

Accessing multiple μDBs to fulfil one query comes with processing overheads. To achieve good performance, Pelton uses materialized views, which accelerate SELECT queries that require data from many μDBs.

For example, consider a simple e-commerce web application with the following schema:

```
CREATE TABLE users (
  ID int,
  Name text,
  PRIMARY KEY(ID)
);

CREATE TABLE products (
  ID int,
  Name text,
  price int,
  PRIMARY KEY(ID)
);

CREATE TABLE purchases (
  user_id int,
  product_id int,
  timestamp int,
  price int,
  FOREIGN KEY (user_id) REFERENCES users(ID),
  FOREIGN KEY (product_id) REFERENCES products(ID)
);

CREATE TABLE ratings (
  user_id int,
  rating int,
  product_id int,
  FOREIGN KEY (user_id) REFERENCES users(ID),
  FOREIGN KEY (product_id) REFERENCES products(ID)
);
```

All personal data of a user (name, purchases and ratings) is stored in the user's shard. Consider a product's webpage on the e-commerce web application, where we need to show average rating of that product. Since product ratings belong to the user who wrote them, they will be scattered across multiple μDBs. Executing a SELECT query that retrieves all ratings of a product and then computes an average rating will need to query all μDBs of users who have rated the product.

Pelton avoids this cost by maintaining materialized views of derived query results over the μDBs. These in-memory materialized views serve queries directly, offering low-latency responses similar to those a memcached or Redis caching layer provides. When Pelton receives a SELECT query, it satisfies the query from the materialized view if there exists a materialized view for this query.

For the given schema, we can create a materialized view for average ratings per product in the following manner:

```
CREATE VIEW v1_avg_rating_per_product AS
  '"SELECT products.ID,products.Name,
    AVG(ratings.rating) AS AVG_RATING
    FROM products
    JOIN ratings ON products.ID = ratings.product_id
    GROUP BY products.ID, products.Name"';
```

The view "v1_avg_rating_per_product" will store the results of the query inside the triple quotes as a single table that can be queried like just another SQL table.

2.3 Dataflow Graph

Pelton must keep the contents of the materialized views up to date. It achieves this via an incremental, streaming dataflow computation triggered by writes to μDBs.

Pelton turns each materialized view definition into a graph of dataflow operators that process incremental updates to the materialized view. Each table in the schema is associated with an Input operator in the dataflow graph, and when Pelton writes to a μDB, it also injects a copy of the new record into the corresponding input operator. The dataflow then processes this record through a

sequence of operators to reach an update to the materialized view, and applies this update.

Figure 1 describes the dataflow graph for a materialized view that contains average ratings for each product.

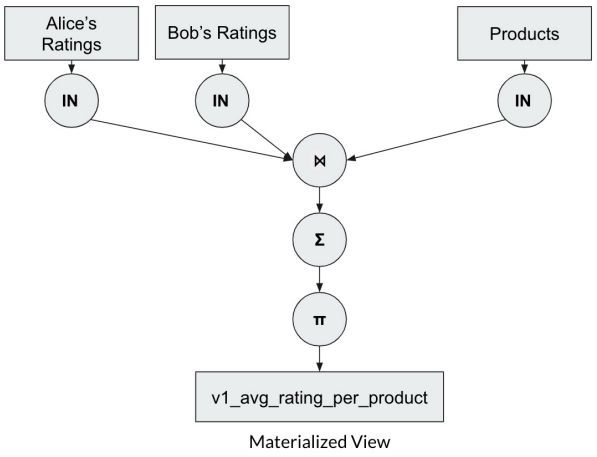

Figure 1: Dataflow graph for v1_avg_rating_per_product materialized view

2.4 Purpose Limitation

To be able to restrict flow of records based on purpose, firstly we need to define a purpose of collection for each record when it is created.

This framework intercepts CREATE TABLE queries and rewrites them to add the column *gdpr_purpose* to the schema. Each table contains this column that stores the purpose of collection for a record as comma separated strings.

A SELECT query comes with a purpose and fetches only those records that match that purpose.

2.5 Expressing Purposes

Materialized views are created for a purpose. Using our example of a simple e-commerce web application, there can be a materialized view for all user ratings that can be used to populate the ratings page of a product. There can be a materialized view for all purchases, average purchases per product and average purchases per user that can be used by the analytics team for business insights. Since a materialized view corresponds to a purpose of collection of data, we use view names as purposes.

2.6 Purpose Operator

We propose a Purpose operator (ψ) as a relational operator just like projection, filter and join.

Figure 2: Purpose Operator

It is a special purpose unary relational operator that selects only those records in a relation that satisfy the purpose of the query and removes the purpose column after selection.

The key idea of this system is that each query should have a single purpose and it should retrieve only those records with complying purposes. Since purposes are names of materialized views, the purpose operator filters records based on the name of the materialized view for which it is filtering.

The purpose operator is the first operator that is called for each shard when it is read by the dataflow. Placing the purpose operator there ensures that the data is filtered out in the beginning of the flow. This is particularly useful when we are joining across multiple tables to create a materialized view.

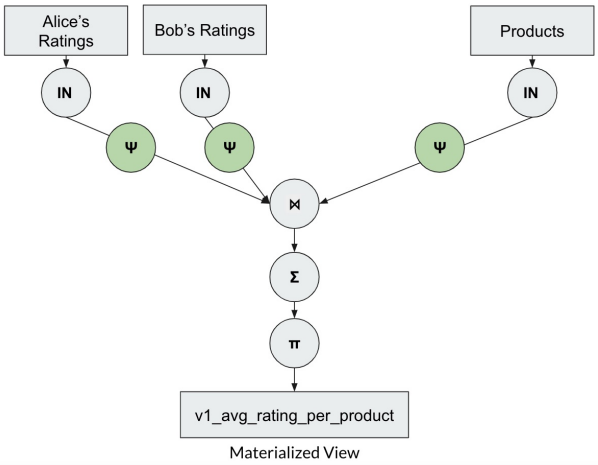

Figure 3: Dataflow graph for v1_avg_rating_per_product materialized view with purpose limitation

3 Implementation

To implement this system, we have made two major changes to Pelton.

Firstly, we intercept the CREATE TABLE query and add the *gdpr_purpose* column to it before it executes.

Secondly, we have created the purpose operator and call it with all Input operators, i.e. whenever data is read from a μDB.

4 Evaluation

4.1 Annotation Effort

Developers who will use this database system to build applications need to add extra value for *gdpr_purpose* in INSERT queries. In all other places, the *gdpr_purpose* column is hidden from the user so usage of all other queries remains the same.

It is reasonable to say that the system requires minimal annotations and requires minimal extra effort by developers to understand and use.

4.2 Performance

Setup: We run all experiments on Apple Macbook A1502 with 8GB of memory and 2.6 GHz Dual-Core Intel Core i5 processor with all databases stored on NVMe SSD storage. We compare our system, Pelton with purpose limitation, with unmodified Pelton as baseline.

Experiment: We generate load by feeding Pelton with a SQL trace file in a closed-loop setting. This trace file simulates the SQL workload of a simple e-commerce web application with 5,000 users using 100,000 queries with 20% writes and 80% reads.

Metrics: The primary metrics of interest are user request latency and the memory required to generate and store in-memory materialized views and µDBs on the hard disk

4.2.1 Latency

There is no extra latency in querying materialized views when compared with the baseline. This is because in both cases, we are querying materialized views. In our system, the materialized view contains only those records that fulfil the purpose of the view. Hence, for the applications, querying on our materialized view is the same as querying on the baseline materialized view.

There might be some latency in making the materialized views consistent after a write query but measuring this is beyond the scope of this project.

4.2.2 Memory

Since the size of the materialized views is the same for our system and the baseline, it consumes no extra memory in our system.

On the other hand, since we have modified the dataflow graph and the underlying µDBs to generate materialized views, there will be some extra memory usage to compute the materialized views after a write query.

We measured that our system consumes 37% extra memory to refresh materialized views after a write query.

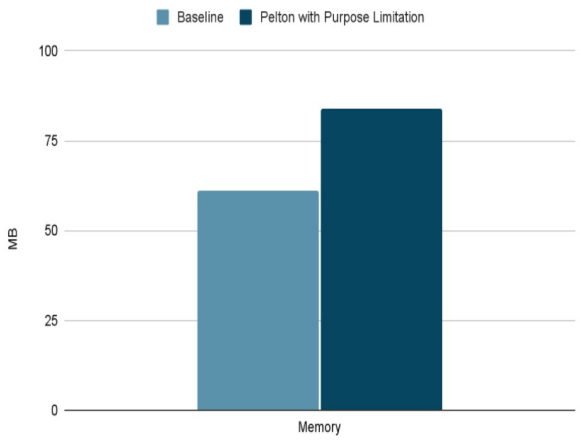

Figure 4: Average Memory consumption to refresh materialized views

The added memory usage can be primarily attributed to storing purpose as long comma separated strings. Storage of purposes can be optimized to store purpose IDs instead of explicit purpose strings.

4.2.3 Disk Space Usage

Our experiment consists of 4 tables (¡Anrut¿) in 5000 µDBs. Each table has an extra column (*gdpr_purpose*).

In this setup, the baseline system uses 0.75 MB of disk space while our system takes 1.46 MB of disk space. This overhead can be attributed to the *gdpr_purpose* column which stores comma separated purpose strings. As discussed in the previous section, optimizing this field can significantly reduce this overhead.

5 Discussion

This system achieves fine-grained purpose limitation for Pelton with minimal overheads and developer annotation. This system achieves this through the purpose operator and no changes to most types of queries.

Since the purpose operator is the first operator for any dataflow graph, it can be extended to log refreshes to the materialized views. This can prove helpful for observability and proving compliance.

We can explore ways to make purpose limitation even more fine-grained. As of now, our system offers purpose-limitation at a record level but we can extend this to record and field, i.e cell level.

References

Pelton Preprint