Data Governance in a Database Operating System (DBOS)

Deeptaanshu Kumar[1], Qian Li[3(✉)], Jason Li[2], Peter Kraft[3], Athinagoras Skiadopoulos[3], Lalith Suresh[4], Michael Cafarella[2], and Michael Stonebraker[2]

[1] Carnegie Mellon University, Pittsburgh, USA
[2] Massachusetts Institute of Technology, Cambridge, USA
[3] Stanford University, Stanford, USA
qianli@cs.stanford.edu
[4] VMware, Palo Alto, USA

Abstract. This paper documents the data governance facilities in DBOS, a database-oriented operating system under construction at Stanford and MIT. Because all operating system state is stored in a high performance main-memory relational DBMS, DBOS has architected a novel data provenance system for all application data. This system uses a high-volume column store for historical provenance information, and provenance data can be queried in SQL. Hence, at its core, DBOS is a polystore data system. Complementing this capability are facilities motivated by GDPR including support for personal data, purposes, and the right to be forgotten.

1 Introduction

At Stanford and MIT, we are building a new operating system stack, based on sophisticated data management: the Database Operating System (DBOS). Herein we briefly motivate the need for a new stack and then turn to novel data provenance capabilities that are facilitated by DBOS. We note that this provenance system requires a collection of polystore capabilities.

Specifically, we are motivated by a collection of hardware and software trends that have occurred since the current Unix/Linux architecture was devised some 50 years ago. First, the scale of operating system (OS) resources under management has increased by several orders of magnitude. From the uniprocessor environments of the 1970's we have evolved to current data centers with thousands of processors. For example, the MIT/Lincoln Labs Supercloud [1] on which DBOS has been build encompasses some 9000 cores. Similar expansion of storage has also occurred. Hence, operating system state (files, tasks, messages, etc.) is several orders of magnitude larger than 50 years ago, and warrants a new approach to state management. Second, Unix/Linux is now elderly software, having been extended/modified/maintained for many, many years. As such, development velocity is slowing; for example, there is no multi-node support and sophisticated multi-core management has been slow in appearing. As a result,

E. K. Rezig et al. (Eds.): Poly 2021/DMAH 2021, LNCS 12921, pp. 43–59, 2021.
https://doi.org/10.1007/978-3-030-93663-1_4

multi-node capabilities must be provided by a second piece of system software (e.g. Kubernetes). This results in a duplication of services, for example two schedulers, and more difficulty in efficient resource utilization. Third, modern data centers now have heterogeneous hardware under management, for example GPUs, TPUs, and FPGAs. However, there is no ability in Linux to manage multiple kinds of processors. Lastly, a data center OS would benefit a great deal from DBMS services. For example, DBMSs provide consistency guarantees on concurrent updates, crash recovery and a high-level language (SQL) for querying OS state.

As a result, we have rearchitected the Linux stack to store *all* OS state in a multi-node, main memory, transactional DBMS. This full function RDBMS will run on top of a microkernel which provides interrupt handling, raw device drivers and very little else. Essentially all OS services (file system, messages, scheduling, etc.) are implemented in SQL on top of this DBMS. Normal user tasks run at the top level in protected fashion, as shown in Fig. 1.

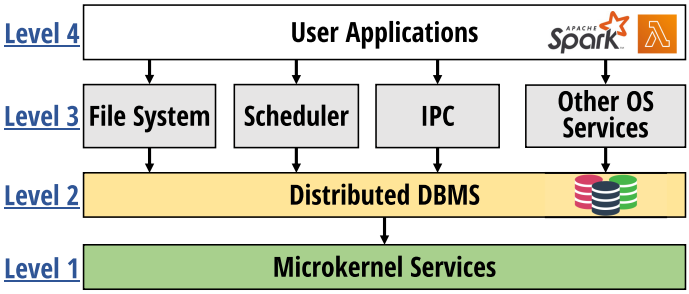


Fig. 1. Proposed DBOS stack.

This architecture has a number of compelling advantages relative to the traditional architecture. First, there is a single piece of software that is managing a multi-node hardware environment. This removes the duplication of function found in current multi-node environments. Second, many OS facilities (e.g. ls, chdir) can be implemented in SQL in a lot less code than in C++. Hence, we expect the footprint of our stack to be smaller than the current one. Third, DBMS services (concurrency control, crash recovery, high availability) are available to all OS functions, resulting in greater functionality (a transactional file system, for example). Lastly, DBMS services are implemented exactly once and then used by everybody, resulting in minimal code duplication. At the present time, we have an initial version of DBOS running as noted in [28], and performance is encouraging. File system services, messages, and task scheduling are competitive in our stack relative to the traditional one. In [28], we also documented our plan for constructing a complete end-to-end DBOS in additional implementation phases.

In this paper we discuss our approach to data governance. System administrators want a complete record of who did what to which objects. This record is useful when answering questions such as:

- Could user X have leaked information to user Y?
- Data element X has been found to be erroneous. Find all data elements that could have been corrupted by this error.
- Find the history of users who have written to file F.
- Find all applications run by user X.
- Find all files copied by user X.

Most large enterprises implement one or more governance systems. A popular choice is Splunk [2], which requires a user to define "events" of interest, which Splunk will then capture from application systems. Often organizations, for example the MIT Supercloud, implement more than one such system, each dealing with different applications. This results in a piecemeal approach to data governance in which the complete picture is spread over several semantically distinct systems. In addition, deploying any new software requires manual intervention to capture new events from the added systems. Most large organizations struggle to meet the ever growing requirements requested by management in this area. Such requirements are unlikely to abate, given the recent legislative and regulatory interest in this area.

Since all OS state is in a DBMS, DBOS enables automatic provenance capture, which will allow easier coverage of events without manual intervention. In Sect. 2, we detail our current DBOS support for data provenance. In Sect. 3 we turn to demonstrating that there is very little overhead to running DBOS provenance, and that interesting provenance queries run with good performance. Then, Sect. 4 turns to the polystore implications of our provenance system and the future directions we are exploring in this area. Section 5 discusses one aspect of the polystore nature of storage, especially in data lakes, which is the use of data catalogs. Section 6 describes several design challenges and our plan to address them. Finally, Sect. 7 discusses GDPR capabilities, and Sect. 8 presents related work.

2 DBOS Data Provenance

2.1 Provenance Architecture

All DBOS operating system state is stored in a main memory DBMS, in our case VoltDB [3]. This includes multiple tables implementing a file system, a scheduling table and a interprocess communication (IPC) table. There are likely to be additional tables storing OS state as the project evolves. For example, the Message table contains the following fields.

```
Message (sender_id, receiver_id, message_id,
         date_time, message_contents, other_fields)
```

The sender activates a stored procedure in VoltDB, which inserts a row in this table with the various fields filled in. The Message table is partitioned across the various nodes in Supercloud. The partition key is receiver_id, so the row is added to the Message partition at the node of the receiver. As noted in [28] this is a single-row single-table operation which is very fast. An efficient implementation of messages would then use a database trigger to alert the receiver, who could use SQL to retrieve the message contents and delete the row in the table. Unfortunately, VoltDB lacks database triggers, so our implementation requires the receiver to poll the database for the message contents. Even with this limitation, DBOS messages are surprisingly performant, as noted in [28].

In [28] we also detailed implementations of a file system and a scheduler using VoltDB tables with a similar architecture.

To implement a complete provenance system, DBOS merely needs to capture all reads and writes to the message table and other tables with relevant OS state. There are several possible ways to do so. First, a conventional DBMS would log all changes to all tables for crash recovery purposes. However, VoltDB uses command logging, as it offers higher performance in their environment [17]. Hence the actual data update is not logged, just the SQL that performed the operation. In addition, a complete provenance system would also require us to capture reads as well as writes.

A second possible implementation is to use VoltDB "change capture". This facility spools all database updates to a file or other location. With no DBOS code, this will capture writes but not reads. If VoltDB supported database triggers, those could be a third possible implementation.

At the present time, we have a system running that uses VoltDB change capture to deal with all write events. To get to a complete system, we plan to migrate to a facility that performs data capture in the DBOS stored procedures that read and write database tables. That way we can capture all reads and writes to table of interest.

2.2 Provenance Specification

For every table in the DBOS VoltDB data base, the table owner must specify the level of provenance they desire. The options are:

- Capture the existence of each write operations
- Capture write operations including the actual data written
- Capture the existence of each read operations
- Capture read operations including the actual data read.

We have had substantial discussions about the granularity of provenance capture. On the one hand, we could capture coarse granularity, for example that user X wrote File Y at time T. Alternately, we could capture that user X wrote block L at time Y or even that user X wrote byte B at time Y. This will obviously dramatically change the size of the provenance database when DBOS is in "capture existence" mode. Our current thinking is to allow user specification

of granularity on a file/table basis. Obviously, there may be additional modes for the provenance system as we gain more experience with it.

2.3 Provenance Database

Obviously, provenance capture entails a massive amount of data especially if the actual data read or written is captured. A high performance OLTP DBMS like VoltDB is ill-suited to the capture of a massive amount of historical data. As a result, we are spooling provenance data transactionally to Vertica, a multi-core, multi-node DBMS based on column store technology that can readily manage petabytes of provenance data. In a DBOS environment, we expect an instance of VoltDB and an instance of Vertica will run on most DBOS nodes.

According to our industrial partners, access control is handled by standard SQL capabilities. Hence, they are worried about legal, but suspicious events, which we call the Edward Snowden effect (ESE). As such, the main use of a provenance database is for after-the-fact monitoring, as we discuss in the next section. Of course, provenance is also useful for detecting error propagation.

2.4 Provenance Queries

In this section, we present ten representative provenance queries, which guide our implementation. This list comes primarily from tasks of interest at DBOS industrial partners.

1. File/DB touch—For a file F or a table T, who was the last person to write each block/record. Who was the first person to do so? Which block/record has the most updates in the last week? In the last year?

2. Connectivity—If X made a network connection with Y, then there is a bi-directional arc between X and Y. Construct the connection graph in the last week. Do the same for the last year. Construct the connection graph of people who talked in the last year but not in the last week. Do the same for systems, described below.

3. Compromised systems/users—A user interacted with a system if the scheduler ran a task on the system on his/her behalf. Who interacted with potentially compromised system S in the last month? What systems did a potentially compromised user interact with in the last month? Trace all connections (transitively) from a compromised system S in the last month.

4. Downstream provenance—Find all blocks/records that could have resulted from information in block/record X. In other words, find a block Y that was written by some user who previously read block X within 5 s. This is "one hop" provenance. Complete provenance requires the transitive closure of this operation.

5. Upstream provenance—Find any block/record X that could have been influenced by block/record Y. In other words, somebody read X and wrote Y - transitively.

6. Debugging—What is the state of a file/table at time T. Now "single step" forward for 3 h.

7. Could X have leaked info to Y?—We define possible leakage as X wrote a file block and Y read the same block within 1 min. In addition, X sent a message to Y, or X wrote a DBMS record and Y read it within 1 min. This is "one-hop" possible leakage. Complete possible leakage is the transitive closure of this operation.

8. Ranking suspicious objects—Administrators are often called upon to rank suspicious objects or behaviors: network packets from potential intrusions, files that are potentially infected, and user data reads that are potentially inappropriate. To rank an object, we can compute an object score using provenance-derived statistics. For example, scoring a particular file open might need to know how many individuals open the file on a typical day. The provenance system should allow administrators to specify and efficiently compute different "object ranking views" that use provenance data.

9. Input auditing—An organization wants to know that it has the legal rights to all of the data resources used to compute a particular output (possibly a sensitive ML model). For file X, the system should: (1) compute every ancestor file of X, and (2) consult a database of file rights to make sure it has legal right to all X's ancestors. That database might be partially derived from GDPR activity, but probably also reflects commercial transactions and other information.

10. Pipeline Modeling—A data pipeline is a long sequence of programs that yields a set of data products. If pipelines are first-class objects, then the provenance system can answer valuable questions valuable to administrators, such as, "Did pipeline X complete successfully on July 23?" and "What pipeline produced file Y?"

The query processing implications of these queries are discussed in Sect. 2.6.

2.5 Provenance Schema

There are two possible approaches to a Vertica provenance schema. First, for any update, we can capture (old_value, new_value) pairs from VoltDB using change capture or our own stored procedures. For inserts, there is no old value and for deletes, there is no new value. For any operation, we insert the appropriate record into Vertica. As such, Vertica manages an insert-only provenance database. The second option for updates is to capture only the new value, and perform a Vertica update (rather than an insert). Since Vertica does not overwrite data, the historical record is preserved with appropriate timestamps and we store two records, without duplicating data between them. We plan to explore the performance and ease of querying for both options.

2.6 Provenance Query Processing

Some of the above queries (1, 6, 8, and 10) can be expressed in normal SQL. On the other hand, several (2, 3, 4, 5, 7, and 9) require transitive closure, which is available in some SQL engines but not others. Specifically, Vertica does not have built-in support for transitive closure. There has been a lot of work in this area [4,9,29]. However, in Vertica it will likely be fastest to code a breadth first algorithm, removing duplicates between iterations. A depth first exploration would require many more user queries and would make parallelism difficult to exploit. On the other hand, breadth-first means running a transitive closure iteration as a parallel SQL query, adding the result to the answer being assembled, removing duplicates at each iteration.

In query 7, in our opinion, indirect leakage is quite rare. Hence, one could stop after one or two iterations, with very low probability of missing a leakage path. Since the iteration is in user code, we can watch the size of the answer being assembled and stop if it does not grow.

Furthermore, one can maintain the transitive closure for each of these queries dynamically, incrementally updating the result when events occur. Alternatively, one can compute the transitive closure only when there is a provenance query. The tradeoff, of course, is the ratio of VoltDB updates to provenance queries. When provenance queries are rare (the usual case), computing the transitive closure in advance is probably the wrong thing to do.

Lastly, in query 7 if (X, Y) is a possible leakage path, then there is no reason to find additional instances of this possibility. As such, this is a "first match" query in which additional instances are not useful.

Although we could run Vertica at level 4 in the diagram of Fig. 1 (i.e. in user space) performance would suffer. Our planned implementation uses the VoltDB store procedures for read and write. If Vertica is run in user space, then an extra two messages will be required. Hence, we are planning to run both DBMSs in the kernel at level 2.

The trend in data warehouse systems is to separate compute from storage, pioneered by systems such as BigQuery [27] and Snowflake [30]. In this way, a storage layer with perhaps limited compute is separate from a compute layer. Of course, the reason for this architecture is to allow compute resources to be scaled up and down elastically as query needs change. Vertica is moving toward this architecture, and in time, most warehouse vendors will offer elasticity on a query-by-query basis.

With this separation, there is the option of pushing portions of a provenance query into the storage layer. In a recent paper [32], some of us analyzed the desirability of pushing down filters and joins into the storage layer. When data blocks are re-referenced frequently, it will be desirable to perform most-to-all of query processing in the compute layer. Alternately, when re-reference is low, then it is best to push down query pieces into the storage layer, when possible. Since provenance queries are expected to be infrequent, it will generally be advantageous to push down as much computation as possible.

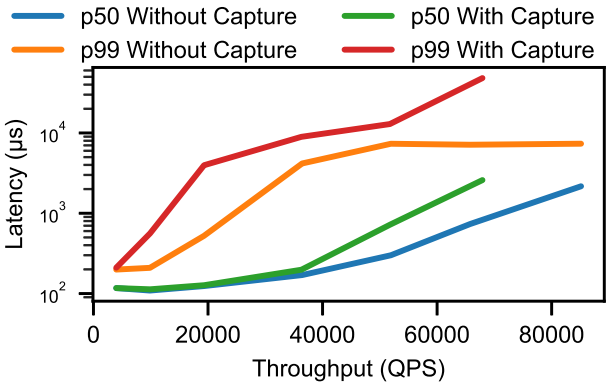


Fig. 2. Throughput versus median and tail latency for a social network workload with and without provenance capture for writes.

3 Performance

To demonstrate the practicality of provenance capture, we instrumented a simple DBOS workload to capture all write operations including the actual data written, then measured workload performance with and without capture. We implemented this instrumentation using VoltDB's change data capture feature, exporting all information to a remote Vertica server. Our benchmark uses the simple Twitter clone Retwis [26], adapted to store all data in VoltDB instead of in Redis. This workload stores all data in VoltDB tables (e.g., a "posts" table) so provenance capture requires logging updates to these table. We execute a workload of 100% writes to a single VoltDB partition, repeatedly making posts for randomly selected users.

We show all results for this benchmark in Fig. 2. We measure throughput versus median and tail latency with an increasing amount of offered load. We find that overhead associated with provenance capture slightly reduces maximum achievable throughput. It has little effect on latency at lower loads, but increases latency somewhat at higher loads. A DBA would have to decide whether detailed provenance was worth the overhead, given the particulars of his load.

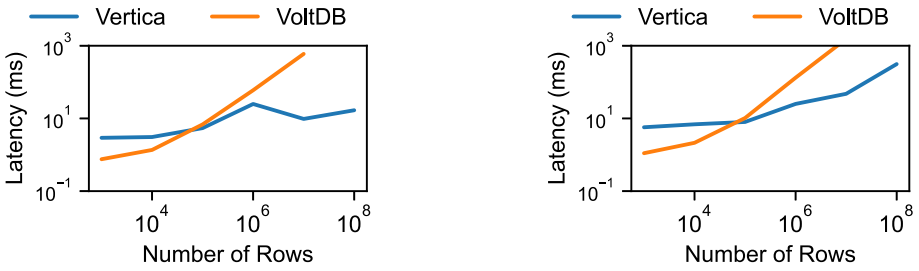
We next evaluate query performance on this captured data. We adapt two of the queries from Sect. 2.4 to Retwis and measure their latency in both Vertica and VoltDB, showing results in Fig. 3.

The first query is "Who was the last person to write a post?":

```
select USERID from RETWISPOSTS order by TIMESTAMP desc limit 1;
```

We show the performance of this query in Fig. 3a. With 100M rows, Vertica can execute this query in 17 ms, while VoltDB slows down and eventually times out when given too much data.

The second query is "Who posted the most since time X?":



(a) Query 1 (b) Query 2

Fig. 3. Performance of provenance queries on social network data using Vertica and VoltDB.

```
select agg.USERID, agg.cnt
  from (select USERID, count(*) as cnt from RETWISPOSTS
    where TIMESTAMP >= 100 group by USERID) as agg
  order by agg.cnt desc limit 1;
```

Figure 3b demonstrates the performance of this query. As before, Vertica can execute this more complex query in 313 ms given 100M rows, while VoltDB slows down and eventually times out with too much data.

These experiments demonstrate that a dedicated OLAP system like Vertica can easily handle provenance queries on large amounts of data. They also demonstrate the need for a polystore in provenance management, as a dedicated OLTP system like VoltDB is not capable of executing large-scale provenance queries.

4 Polystore Implications

As noted previously, DBOS is a fairly simple polystore that spools provenance data from VoltDB to Vertica. However, it is obviously a good idea to support a file system on top of Vertica. For gigantic files, this will offer a compressed column store implementation which will outperform the VoltDB row store. Also, there is no reason to disallow users from storing DBMS data in Vertica, if they so choose. As such, we will have two different DBMSs generating provenance information.

More generally, there will potentially be other DBMSs in which user data is stored and/or files supported. On a case-by-case basis, we will explore supporting such other DBMSs. Also, over time we expect to have to support provenance information in multiple data warehouse-oriented column stores. This situation could arise if applications insist on spooling their provenance data to a preferred DBMS. This leads to the general polystore architecture of Fig. 4.

With multiple provenance stores, standard SQL queries will access only one of the repositories. However, figuring out which one will require a data catalog, discussed in the next section. Also, the scope of our transitive closure queries will be all systems. We distinguish two cases of interest. In the first case provenance information is separable and there is no cross-talk between the systems. In this

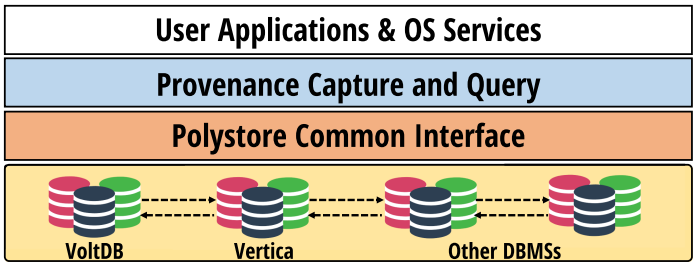


Fig. 4. Proposed polystore architecture.

case there is no possibility of a user reading a file or a database that spools to one repository and then writing a file or a database that spools elsewhere. Hence, one can run the transitive closure queries on each repository individually and then merge the answers. On the other hand, there will be situations where provenance information is not separable. This will lead to a more complex query processing strategy, whereby intermediate results must be traded between provenance stores.

However, it is a reasonable assumption that a provenance system need only support warehouse-oriented DBMSs and only for a subset of possible queries. As such polystore complexity is limited.

5 Support for Data Catalogs

The previous section noted the problem of finding metadata across multiple storage systems. Obviously, the metadata within a single DBMS is correct; however, enterprises are typically running several-to-many DBMSs. Also, metadata for files is often not captured anywhere. A common architecture is to move all such data to a data lake (or lakehouse, if you wish) and then build a catalog for lake objects.

There are a number of recent data catalog systems that do exactly this. These are standalone systems that serve as authoritative sources of metadata for all the datasets in an organization. Commercial systems include Alation, Collibra, and data.world. Open source systems include ckan, Amundsen, and Magda.

Data catalogs allow metadata queries by humans and external systems. For example, a compliance system might compare the access permissions of an observed database with the data privacy requirements stated in the organization's data catalog. Catalogs can also be instrumental in enabling better organizational data search.

In addition, data catalogs also play an important role in data access control and information security initiatives. For example, companies frequently implement course-grained or fine-grained access control based on data classifications stored within these catalogs.

In our conversations with commercial users of such systems, we have observed two common problems. First, the catalogs are incomplete. Although most of

these systems include crawlers that will traverse existing data assets and help populate the catalog, there is still some human curation effort needed to ensure datasets have correct schemas, personal data settings, and so on. Moreover, many users build semi-private datasets that are inaccessible to crawlers and are never added to the catalog. Second, the catalogs fail to attract wide audiences inside the organizations that build them. This might be due to the poor quality of the catalogs, or because the catalogs simply do not deliver enough compelling value to the typical data users.

We wish to make two points in this section. First, if an enterprise decided to run DBOS everywhere, then a rudimentary data catalog is automatically constructed. This DBOS catalog is by definition complete and accurate, avoiding the criticisms discussed above. Also, if DBOS provenance is used, then the lineage of every object is automatically provided. This is a powerful form of metadata, which will help users uncover the semantic definition of a data set, even when it is missing or incomplete.

6 Design Challenges

We now describe a few ongoing design challenges for any useful provenance system, and how we address them in DBOS.

6.1 Provenance Data Capture

Data capture is a serious challenge for provenance systems. Past efforts have addressed this challenge in two main ways, both unsatisfying:

1. Users must rewrite their code with a new toolchain, which generally yields high-quality data at the cost of high human effort. With this approach, the coverage of data provenance often suffers.
2. Automatic instrumentation of unmodified code, which generally yields low-quality data at a low human cost. With this approach, the usefulness of data provenance often suffers.

The design of DBOS alters this playing field dramatically. Since provenance is integrated with the OS itself, all important operations are captured and logged. We expect this design to make a big difference.

However, DBOS provenance still has shortcomings. For example, suppose OS-visible operations do not reflect operations that downstream users are interested in. Consider a privacy policy-compliance process that scans every file in DBOS and generates a per-user report file describing possible violations. The DBOS provenance system will show that each output report file is dependent on every file in the system. In most cases, such provenance information is either misleading or useless. Capturing such provenance data is not helpful to the application being run, and another (presumably higher level) system is required.

Another problem arises when data crosses DBOS-visible boundaries. In particular, the provenance of any dataset that escapes via traditional I/O channels

(e.g., a screen, a log file, or over a network socket) can no longer be tracked with confidence. Disabling traditional display and network access would make DBOS unusable for many applications.

We can mitigate this problem by sandboxing stored procedures and either noting in the provenance record when bytes left the DBOS system, or disabling such operations altogether for sensitive data. There are several solutions of interest in this space. For example, cloud providers already use techniques like runtime sandboxing [24] and restricting process privileges via mechanisms like SECCOMP [15]. In addition, sandboxing techniques in dynamic information flow control (IFC) systems like Trapeze [5] are also applicable in our setting.

These issues are serious. While DBOS' current provenance design partially addresses them, they are still questions for ongoing research. One approach is discussed in the next subsection.

6.2 Application Integration

Organizations operate with multiple data abstractions. Obviously, organizational activity can be captured as files, records, processes, and function invocations. But organizations also have pipelines, approval processes, business patterns, and other "objects" that intersect with, but are not identical to, computational objects.

For example, "did marketing approve the latest commercial?" is a business question, but it can also be framed as a provenance question when combined with a file identified as "the latest commercial", a user group identified as "marketing", and a particular process execution identified as "approve".

DBOS can enable integration of provenance with these external non-provenance concepts in two ways.

Concept-As-View—The user can define views that model external concepts. For example, the table of "commercials" might be written as a view over the set of DBOS files that are in the commercials directory and which have a member of the marketing team as an owner. An important property of such a system is the use of user-defined functions as part of the view definition. This allows arbitrary domain-specific questions to be asked of the DBOS and its provenance objects.

Federated Query Optimization—A user query that involves non-provenance objects might involve query processing over multiple schemas, for example a relational database of DBOS files and a graph database of provenance information. We plan to study optimization across multiple systems so that user queries can be executed in reasonable amounts of time.

7 Support for Capabilities Motivated by GDPR

7.1 Personal Data

GDPR legislates special support for personal data. One of us is a lawyer specializing in privacy issues such as this one. Although it would be very helpful to

have an algorithm decide what fields are personal data and what ones are not, such a feature seems out of reach, since personal data is somewhat subjective. Instead, a human must specify what is personal data. In a DBOS environment, this requires tagging every column of every table in a DBOS instance with a notation whether it is personal data or not. Furthermore, it is equally difficult to automatically mark derived data (materialized views, query results). Hence, it is assumed that all derived data will be appropriately marked, and we will not try to build a system to automatically mark derived data.

Such a marking system can be added trivially to the system catalogs (metadata).

7.2 Purposes

GDPR legislates that every person with personal data stored in a service have the right to decide for what purposes his/her data can be used. Example purposes might be medical research or advertising. Hence, the service provider decides on a collection, K, of (otherwise uninterpreted) strings, called purposes. Every item of personal data is tagged with the purposes the owner of that data item allows for that data element.

Although in theory, there can be tens or hundreds of purposes, we expect the normal case will be a half a dozen or less. Otherwise, it will be too confusing for users to say yes/no to each of tens of purposes. In a previous paper [13], we advocated using extra bits in each record to store this yes/no information. However, when the number of purposes is small, we think an alternate implementation will be more efficient.

For every column of personal data and for each purpose, we plan to store in DBOS an exception list of record identifiers of persons who have opted out of allowing their data to be used for that purpose. We expect the normal case is that people will not opt out, so these lists will not be onerous to store. Every query which is sent to a service must include one of the authorized purposes.

The query executor just needs to add the following processing step whenever it picks up a piece of personal data:

- Look up the person ID in the appropriate exception list
- If found, do not return the requested data element.

We anticipate the exception lists for a table will be small and will be cached in main memory when the table is active. A bit-oriented implementation will require one bit per record. This scheme requires one record identifier per person that opts out. As long as the opt out rate is low (less than 1%), this scheme will be more efficient. We can also use delta encoding for record identifiers to cut down on the amount of space they consume.

7.3 The Right to be Forgotten

Any person, P, with personal data in a service can request to be forgotten. In this case all personal data (defined above) should be deleted by the service.

It is assumed that P presents their identifier, I, or the service can look it up. The identifier is assumed to be the key of one or more tables.

If there is a "path" from the key to an item of personal data, then this item must be deleted (nulled). A path is defined as a collection of column names, $N_1, \ldots, N_k$, such that $(I, N_1), (N_1, N_2), \ldots, (N_{k-1}, N_k)$ are the composite keys of intermediate tables and N_k is the key of a table, T, with personal data. Any personal data in the appropriate row of T should be nulled.

We expect to look for efficient ways to perform this operation. In addition, we GDPR legislated that a service has 30 days to perform this operation. Hence, it is possible to batch such requests and perform them in bulk. We expect to see if this technique is more efficient than forgetting people one at a time.

8 Related Work

There has been a substantial amount of provenance research, including work on data models, query processing, and practical systems.

Provenance Models—There has been a vast amount of theoretical and model-related provenance research. Cheney, Chiticariu, and Tan provide a useful overview [7]. Provenance queries are generally divided among three models:

- *Why provenance* queries that identify all the source values contributed to the computation of a particular output,
- *How provenance* queries that describe the computation that combined the source values, and
- *Where provenance* queries that describe where a particular piece of output information was copied from.

For most of our DBOS target queries, *why provenance* and *where provenance* are likely the most relevant model.

Query Processing—Query processing is a major thrust of provenance work. Green, *et al.* [8] showed that query processing for why-provenance queries can be viewed as an example of a broader class of query processing methods that can also be used in probabilistic and incomplete databases. Recent work [22] takes a user's example *why provenance* query and rewrites it to match user guidance about which entities should be included or not; this method might be a good fit to typical DBOS scenarios. Chiticariu, *et al.* [12] introduced a system that permits manual annotations of relational data, along with a mechanism for users to describe how annotations should be propagated.

Data Collection—For many non-relational systems, there is an additional challenge associated with non-relational software: how to actually capture the provenance. The noWorkflow [20] system automatically collects information about Python programs at code-definition time as well as runtime. Vamsa [21] uses static analysis of Python programs to derive provenance for machine learning models. Chapman, *et al.* [6] aim to capture provenance for data preprocessing

code; they introduce a set of operators that closely resemble common preprepocessing patterns, then annotate Python code with their standard operators. Scientific workflow systems [10,14,18,31] ask users to manually annotate code for provenance collection. Dagger asks users to annotate data at certain interfaces between code blocks [25]. Other systems [11,19] collect provenance via automatic automatic instrumentation of a process' interaction with the computational environment; this is perhaps the most similar approach in previous work to what DBOS does today.

All of these systems struggle to obtain provenance data that is relevant and complete without huge human effort. Unlike relational databases with their fixed set of operators, general-purpose programs have neither a fixed set of operations, nor an obvious best place for instrumenting those operations. In work to date, either the programmer must manually annotate existing code to capture provenance information, at great human effort; or the system must try to automatically instrument unmodified code, and thereby potentially capture confusing "operations" at an inappropriate level of granularity.

By moving many operations into a relational model, DBOS has some data capture advantages over previous work. Many OS operations—such as file create, or network transmissions, or process launches—can be observed as a standard relational INSERT. In many cases, the semantics of these operations are broadly understood and can support a range of likely downstream queries.

However, there is nothing that requires a DBOS-visible operation to make sense to a future provenance query-writer; consider that a user launching a single program from a shell will appear to be a new entry in a process table, as will just one of the many independent processes that together allow a modern web browser to operate. As a result, even though DBOS operates via the relational model, some aspects of DBOS data capture closely resemble the challenges usually associated with general-purpose program provenance.

Practical Systems—There are many issues that arise when building practical provenance systems, especially when the volume of provenance data grows very large. Zheng and Ives examine how to build a provenance system that is efficient and tamper-proof enough for long term archival use [33]. The Smoke system [23] is an in-memory database explicitly designed for efficient provenance capture and querying, employing specialized optimizations when provenance queries are known in advance, which is likely in many use cases. As provenance is especially useful in data science use cases, the NBSafety [16] system is tailored for preserving provenance in notebook-style settings where cell dependencies are easy to lose track of.

9 Conclusions

In this paper we have presented a provenance system built into the DBOS operating system. This automatically captures a lot of provenance events without manual intervention by a user. We have show that the run-time overhead of the

system is modest and query performance on the provenance database is reasonable. The polystore implications of our approach were also discussed. Our plan going forward is to build a complete end-to-end DBOS implementation.

References

1. Mit supercloud (2021). https://supercloud.mit.edu/
2. Splunk (2021). https://www.splunk.com/
3. VoltDB (2021). https://www.voltdb.com/
4. Agrawal, R., Jagadish, H.: Direct algorithms for computing the transitive closure of database relations. In: VLDB, vol. 87, pp. 1–4 (1987)
5. Alpernas, K., et al.: Secure serverless computing using dynamic information flow control. In: Proceedings of the ACM Programming Languages (OOPSLA), October 2018. https://doi.org/10.1145/3276488,https://doi.org/10.1145/3276488
6. Chapman, A., Missier, P., Simonelli, G., Torlone, R.: Capturing and querying fine-grained provenance of preprocessing pipelines in data science. Proc. VLDB Endow. **14**(4), 507–520 (2020). https://doi.org/10.14778/3436905.3436911
7. Cheney, J., Chiticariu, L., Tan, W.C.: Provenance in databases: why, how, and where. Found. Trends Databases **1**(4), 379–474 (2009). https://doi.org/10.1561/1900000006
8. Chiticariu, L., Tan, W.C., Vijayvargiya, G.: Dbnotes: a post-it system for relational databases based on provenance. In: Conference: Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005, pp. 942–944, January 2005. https://doi.org/10.1145/1066157.1066296
9. Dar, S., Ramakrishnan, R.: A performance study of transitive closure algorithms. ACM SIGMOD Record. **23**(2), 454–465 (1994)
10. Frew, J., Bose, R.: Earth system science workbench: a data management infrastructure for earth science products, pp. 180–189, January 2001. https://doi.org/10.1109/SSDM.2001.938550
11. Frew, J., Metzger, D., Slaughter, P.: Automatic capture and reconstruction of computational provenance. Concurr. Comput. Pract. Exp. **20**, 485–496 (2008). https://doi.org/10.1002/cpe.1247
12. Green, T.J., Karvounarakis, G., Tannen, V.: Provenance semirings. In: Proceedings of the Twenty-Sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2007, pp. 31–40. Association for Computing Machinery, New York (2007). https://doi.org/10.1145/1265530.1265535,https://doi.org/10.1145/1265530.1265535
13. Gadepally, V., Mattson, T., Stonebraker, M., Wang, F., Luo, G., Laing, Y., Dubovitskaya, A. (eds.): DMAH/Poly -2019. LNCS, vol. 11721. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-33752-0
14. Lin, C., et al.: A reference architecture for scientific workflow management systems and the view SOA solution. IEEE Trans. Serv. Comput. **2**, 79–92 (2009). https://doi.org/10.1109/TSC.2009.4
15. Linux: Linux seccomp. https://man7.org/linux/man-pages/man2/seccomp.2.html
16. Macke, S., Gong, H., Lee, D.J.L., Head, A., Xin, D., Parameswaran, A.: Fine-grained lineage for safer notebook interactions (2021)
17. Malviya, N., Weisberg, A., Madden, S., Stonebraker, M.: Rethinking main memory OLTP recovery. In: 2014 IEEE 30th International Conference on Data Engineering, pp. 604–615. IEEE (2014)

18. McPhillips, T., Song, T., Kolisnik, T., Aulenbach, S., Freire, J.: al et: Yesworkflow: a user-oriented, language-independent tool for recovering workflow information from scripts. Int. J. Digit. Cur. **10**(1), 298–313 (2015)
19. Muniswamy-Reddy, K.K., Holland, D.A., Braun, U., Seltzer, M.: Provenance-aware storage systems. In: Proceedings of the Annual Conference on USENIX 2006 Annual Technical Conference, ATEC 2006, p. 4. USENIX Association (2006)
20. Murta, L., Braganholo, V., Chirigati, F., Koop, D., Freire, J.: noworkflow: capturing and analyzing provenance of scripts. In: Ludäscher, B., Plale, B. (eds.) Provenance and Annotation of Data and Processes, pp. 71–83. Springer, Cham (2015)
21. Namaki, M.H., et al.: Vamsa: Automated Provenance Tracking in Data Science Scripts, pp. 1542–1551. Association for Computing Machinery, New York (2020). https://doi.org/10.1145/3394486.3403205
22. Namaki, M.H., Song, Q., Wu, Y., Yang, S.: Answering why-questions by exemplars in attributed graphs. In: Proceedings of the 2019 International Conference on Management of Data, SIGMOD 2019, pp. 1481–1498. Association for Computing Machinery, New York (2019). https://doi.org/10.1145/3299869.3319890,https://doi.org/10.1145/3299869.3319890
23. Psallidas, F., Wu, E.: Smoke: fine-grained lineage at interactive speed. Proc. VLDB Endow. **11**(6), 719–732 (2018). https://doi.org/10.14778/3199517.3199522
24. PyPy: Pypy's sandboxing features. https://doc.pypy.org/en/release-2.0-beta2/sandbox.html
25. Rezig, E.K., et al.: Dagger: a data (not code) debugger. In: 10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, 12–15 January 2020, Online Proceedings. www.cidrdb.org (2020). http://cidrdb.org/cidr2020/papers/p35-rezig-cidr20.pdf
26. Salvatore Sanfilippo: Retwis: a twitter toy-clone (2014). https://github.com/antirez/retwis
27. Sato, K.: An inside look at google bigquery. White paper (2012). https://cloud.google.com/files/BigQueryTechnicalWP.pdf
28. Skiadopoulos, A., et al.: DBOS: a DBMS-oriented Operating System. Submitted for publication (2021)
29. Valduriez, P., Khoshfian, S.: Parallel evaluation of the transitive closure of a database relation. Int. J. Parallel Program. **17**(1), 19–42 (1988)
30. Vuppalapati, M., Miron, J., Agarwal, R., Truong, D., Motivala, A., Cruanes, T.: Building an elastic query engine on disaggregated storage. In: 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2020), pp. 449–462. USENIX Association, Santa Clara, February 2020. https://www.usenix.org/conference/nsdi20/presentation/vuppalapati
31. Wolstencroft, K., et al.: The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud. Nucl. Acids Res. **41**(W1), W557–W561 (2013). https://doi.org/10.1093/nar/gkt328,https://doi.org/10.1093/nar/gkt328
32. Yang, Y., et al.: Flexpushdowndb: Hybrid pushdown and caching in a cloud DBMS. In: VLDB, vol. 14 (2021)
33. Zheng, N., Ives, Z.G.: Compact, tamper-resistant archival of fine-grained provenance. Proc. VLDB Endow. **14**(4), 485–497 (2020). https://doi.org/10.14778/3436905.3436909