# LDV: Light-weight Database Virtualization

Quan Pham [#1], Tanu Malik [#2], Boris Glavic [*3], Ian Foster [#4]

[#] *Computation Institute, University of Chicago, Chicago, Illinois, USA*
[1] quanpt@cs.uchicago.edu, [2] tanum@ci.uchicago.edu, [4] foster@ci.uchicago.edu

[*] *Department of Computer Science, Illinois Institute of Technology, Chicago, Illinois, USA*
[3] bglavic@iit.edu

***Abstract*—We present a light-weight database virtualization (LDV) system that allows users to share and re-execute applications that operate on a relational database (DB). Previous methods for sharing DB applications, such as companion websites and virtual machine images (VMIs), support neither easy and efficient re-execution nor the sharing of only a relevant DB subset. LDV addresses these issues by monitoring application execution, including DB operations, and using the resulting execution trace to create a lightweight re-executable package. A LDV package includes, in addition to the application, either the DB management system (DBMS) and relevant data or, if the DBMS and/or data cannot be shared, just the application-DBMS communications for replay during re-execution. We introduce a linked DB-operating system provenance model and show how to infer data dependencies based on temporal information about the DB operations performed by the application's process(es). We use this model to determine the DB subset that needs to be included in a package in order to enable re-execution. We compare LDV with other sharing methods in terms of package size, monitoring overhead, and re-execution overhead. We show that LDV packages are often more than an order of magnitude smaller than a VMI for the same application, and have negligible re-execution overhead.**

## I. Introduction

Sharing and repeating applications is crucial for verifying claims, reproducing experimental results (e.g., to repeat a computational experiment described in a publication), and promoting reuse of complex applications. However, no methods currently exist that enable easy sharing and efficient repeatability for applications that use a relational DB.

For example, consider Alice, who has implemented a new algorithm for finding dark matter halos in the Sloan Digital Sky Survey's SkyServer relational DB. The predominant methods of sharing and making such an application repeatable for other researchers are *building a companion web site* and/or *provisioning a virtual machine image* (VMI). The former provides access to the application programs. However, access to the relational DB used by the application or sharing part of its data may require appropriate privileges, such as being a power user on the SkyServer. Thus, building a companion website often does not guarantee repeatability. A VMI, if provisioned correctly with the application and data, ensures repeatability. However, creating such a VMI requires specialized knowledge and much effort from the application developer. Furthermore, a VMI that includes all data may be extremely large.

*Application virtualization* has recently emerged as a light-weight alternative for sharing and efficient repeatability. This approach traces system calls during application execution and copies all binaries, data, and software dependencies into a *package*. The resulting package can be run on any compatible machine (e.g., most virtualization approaches work only on particular OS) without installation, configuration, or root permissions. Application virtualization, as implemented by tools such as CDE [13], has several advantages. Because it needs to capture only the necessary data and software environment, it can create packages that are more light-weight than VM images. Furthermore, automating the creation of self-contained packages makes sharing scientific results as easy as building companion websites, and for deterministic computations, re-running the packages guarantees repeatability.

However, despite such advances, sharing and repeating DB applications remains challenging for three reasons:

1) Most applications use only a fraction of a DB's data. When sharing a DB application we may want to include only the subset that is accessed by the application. However, isolating this subset can be non-trivial when applications use update operations and/or complex queries.
2) Sharing the DB server binaries and data may not be an option, because the user may have limited access to these files (e.g., the server is run by a third party), because of licensing issues, or because of data usage policies.
3) To successfully repeat an execution, the DB has to be restored to the state valid at the start of the application. This may be challenging, in particular, if the DB is shared among multiple users and applications.

Current methods—companion websites, VM images, and application virtualization—do not address these unique requirements of DB applications. They provide no means for determining which data are relevant for an application, cannot reset a DB to a previous state, and do not solve the licensing problem of sharing commercial DBMS binaries. Virtualization can ensure reproducibility if the user has control over the DB server, but the server has to be started and stopped by the application in order to include a consistent DB state. The resulting package must inevitably include the complete DB. Application virtualization is currently limited to applications that do not communicate with server processes. In fact, when an application communicates with a DBMS, the technique can *at most* record this communication. This information is not sufficient for determining which data was used by the application (and, thus, should be included in the package) and does not solve the problem of resetting the DB to the state

valid before the application started. Temporal DBs provide a solution to the latter problem, but not for the former.

We present here an alternative method, termed *light-weight DB virtualization* (*LDV*), which allows researchers to share DB applications as self-contained packages that can be repeated by other researchers, thus achieving reproducibility for DB applications. An application shared in this way runs exactly as it did for the original user—it requires no access to the original DB, no manual installation of a DBMS, and no manual restoration of the DB. Our approach extends application virtualization in several ways to address the challenges outlined above. Importantly, we use linked OS and DB provenance to solve the first and third challenges.

## II. Light-weight DB Virtualization

We now provide an overview of our approach, highlight our contributions, and describe how we use provenance metadata to create and repeat an LDV package. Consider our example, in which Alice's halo-finder application reads some input simulation data and outputs potential halos in a region of sky. To find precise halos, the application also reads observational data from the Sloan DB. We assume that the application accesses the DB using SQL, and that it uses standard bulk copy and DB dump utilities. Though we assume here a client-server model for simplicity, our system can be expanded to more general distributed settings.

Alice would like to share this application with Bob as a package. Bob may then want to either (i) re-execute the application in its entirety, (ii) re-execute the application without reading data from the Sloan DB, or (iii) provide his own data inputs to the halo-finder. In order for Bob to re-execute and build upon Alice's application, he must be provided with access to application binaries and data, any extension modules that the code depends upon (e.g., dynamically linked libraries), a DB server, and a DB storing data relevant to the application.

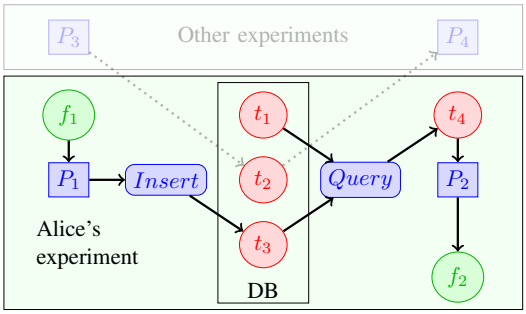


Fig. 1: Alice's application consists of two processes $P_1$ and $P_2$. Process $P_1$ reads from a file $f_1$ and inserts a new tuple $t_1$ into the DB. Process $P_2$ runs a query that produces a result tuple $t_4$ from tuples $t_1$ and $t_3$.

Determining the data relevant to an application is challenging. We want to exclude tuples that are not accessed by the application, in order to reduce package size. Thus, for example, tuple $t_2$ in the version of Alice's application shown in Figure 1 should be excluded from the repeatability package, because it was not used by any SQL statement. We must also exclude tuples that were created by the application, since re-execution will recreate the tuples, which could lead to violation of primary key constraints and/or affect the results of queries that are sensitive to duplicates. For instance, tuple $t_3$ in Alice's application should be excluded from the package because it will be regenerated during re-execution. As we will explain, we can address these challenges by monitoring both the application and DB side of the application.

**Linking Provenance Models (*Section IV*)** The two tasks of (a) identifying the DB tuples used by a computation and (b) replaying a computation at a later time can both be addressed by recording appropriate provenance (i.e., lineage) metadata for the computation's outputs. However, in order for this approach to work, we need a provenance model that can capture both DB operations ("DB provenance") and process and file system operations ("OS provenance") To this end, we introduce a generic provenance model (representable using the PROV standard [20]) capable of expressing and linking existing DB and OS provenance models. We demonstrate the effectiveness of our model by using it to link the models of the OS and DB provenance systems used in our prototype implementation. We study how to infer data dependencies across provenance models based on a combination of (a) dependencies available in the individual models and (b) temporal annotations on interactions between activities and entities of a provenance graph. This approach enables us to determine which DB tuples are required to re-execute a workflow; to determine the parts of an application that are needed for partial execution; and to answer reachability queries (does data item $d$ depend on data item $d'$).

**Monitoring DB Applications (*Section VII*)** We create a package for an application by monitoring the processes involved in an application, their file system operations, and their interactions with the DB. To monitor processes and their file system operations, we use an existing solution such as PTU [22], CDE-SP [23], or OPUS [3], which monitors system calls to determine when processes are spawned and when a process opens or closes a file. We then use this information to construct a provenance graph for the application execution. To monitor process interactions with the DB, we capture the SQL statements (queries and DML statements) executed by a monitored process (plus, in some cases, their results) and record which statement was executed when. We can use, e.g., Perm [10] to compute the provenance of SQL statements. While monitoring a DB application we interactively construct an *execution trace* (a provenance graph of our generic provenance model) for the workflow.

**Server-included and Server-excluded Virtualization Packages (*Section VII-D*)** Much as in application virtualization approaches, LDV repeatability packages include application binaries, dependencies such as dynamically linked libraries, and data files. We consider two approaches for packaging DB-related content. In the *server-included* approach, the DB server binaries and relevant DB data are included within the

package. This approach is applicable if the DB server binaries can be legally shared and are accessible by the user. In the *server-excluded* approach, we do not include the DB server binary or any DB content; instead, we record the results of queries issued by the application and include these results in the package. When re-executing the package, we replay the recorded results instead of actually executing SQL statements. This option does not require access to the DB server binaries.

**Prototype Implementation (*Section IX*)** We present a prototype implementation that integrates the PTU OS provenance system [22] and the Perm DB provenance system [11], [10]. This prototype instruments the client interface of Perm, an provenance-enabled version of the PostgreSQL open source DBMS, to monitor DB interactions. When using the server-included packaging option, we use Perm to determine, for each DB query, the part of the DB that is relevant to the workflow. We have expanded PTU to incorporate DBs into the resulting packages using the two packaging options described above. We plan in future work to support additional DB systems using the GProM [2] provenance middleware.

## III. Related Work

The following description of scientific reproducibility is consistent with several works [16], [22], [9], [7], [14]: "Given a science experiment conducted entirely using computational artifacts, at the least, scientific reproducibility is the verification of scientific results by repeating (or replicating) them on nominally equal configurations." Often, further validation is required for an experiment to be considered reproducible, such generalizing scientific results by applying them to new data sets, verifying how they behave under different parameters, and re-using and extending the experiment [22].

**Virtualization** Keahey et al. [16] were the first to propose using VMs to encapsulate large, complicated stacks of scientific software so they can be deployed across supercomputing centers without the need to install each software package individually in every new environment. This technique has also been applied in the cloud computing context by Howe et al. [14]. However, as noted by Brown [4] and Lampoudi [18], VMs are space inefficient and not descriptive enough to enable validation, especially through provenance-tracing mechanisms.

Application virtualization tools such as CDE [13] are more space efficient. CDE uses the UNIX *ptrace* utility to monitor system calls and create a software package consisting of application binaries, data, and all static and dynamic software dependencies that can be traced during program execution. While a CDE package provides the ability to rerun the application in a different Linux environment, it provides no provenance and, thus, no means of validation. PTU packages [22] were proposed to enable validation by constructing OS-level provenance graphs using *ptrace* auditing. Other packaging systems that use provenance for validation but use different capture mechanisms are ReproZip [7] (VisTrails workflow system [9]) and Research Objects (MyExperiment [12]). None of these approaches address the problem of packaging a DB or support fine-grained DB provenance which is necessary for repeating DB applications.

**Unified Provenance Models** The different computational and data models used on the OS and DB side have led to different provenance models. Most OS provenance models track provenance at the granularity of processes and files [3], [22], [23], [19]. Each process is considered as a black box, such that all outputs are dependent on all inputs. Notable exceptions are approaches that use dynamic instrumentation to compute fine-grained provenance for binary programs [26], [24]. In contrast, DB provenance [5], [15] is fine-grained (usually at the granularity of tuples), connecting result tuples to input tuples of query operators. Various proposals for unified provenance models have been made in the past. Cheney et al. define fine-grained provenance for a data flow language that is capable of expressing both DB-style queries and workflows. Amsterdamer et al. [1] take the opposite approach of modeling workflows as collections of programs written in a subset of the Pig language, which corresponds to nested relational algebra, and capture fine-grained provenance using a DB provenance model. Unified provenance models enable precise and detailed description of provenance. However, they require the adoption of a particular programming model and/or system. Reimplementing a complex DB application in a different programming language is often not feasible.

**Combining Database and OS Provenance** An alternative approach is to *link* DB and OS provenance, for example by linking nodes from OS and DB provenance graphs. Reddy et al. [21] first described a cross-layer provenance system that links provenance from different system layers and establishes dependencies between underlying data and processes. In their system, cross-layer provenance issues were explored in the context of workflow systems and NFS servers. VisTrails [6] considers workflows that interact with a DB, using the auditing and temporal DB features of a commercial DBMS to support provenance. The features required by this approach are currently not supported by most DBMS. Furthermore, the approach does not support fine-grained DB provenance.

A full understanding of the trade-offs between the unified and linking approaches will require further research. In this work, we use the linking provenance model since it requires no modification of the user's DB application and leverages established OS and DB provenance models. We link a fine-grained DB provenance model to an OS provenance model and infer dependencies across the two models. This helps us to determine which parts of the DB are relevant to a computation, and thus results in reduced package sizes.

## IV. DB and OS Provenance Models

### A. Provenance Model

To be able to record provenance for DB applications and repeat them, we need to connect OS and DB provenance. We also need to infer dependencies between an application's OS entities (e.g., files) and entities managed by the DBMS (tuples). Such cross-model dependencies enable us to reason

about which data is required to reproduce an execution (or part of an execution) and, thus, to determine which parts of the DB to include in a repeatability package. We assume that the following holds for both the DB and OS provenance models:

- The model defines a set of activity and entity types valid in its domain, e.g., the DB model may represent SQL statements as activities.
- The model defines inference rules for determining data dependencies that connect entities, e.g., a file written by a process $p$ may depend on a file read by $p$.
- The produced provenance can be represented in PROV. (We do not require that these systems export provenance in PROV format, but only that we can encode their provenance in PROV. Given PROV's generality, this requirement should not represent a limitation.)

We define a *provenance model* generically as follows:

**Definition 1** (Provenance Model)**.** *Let $\mathbb{L}$ be a domain of labels. A provenance model is a triple $\mathbb{P} = (\mathcal{A}, \mathcal{E}, \mathcal{L})$ where $\mathcal{A} \subseteq \mathbb{L}$ is a set of activity types and $\mathcal{E} \subseteq \mathbb{L}$ is a set of entity types. $\mathcal{L}$ is a subset of $\mathbb{L} \times (\mathcal{A} \cup \mathcal{E}) \times (\mathcal{A} \cup \mathcal{E})$. Each triple in $\mathcal{L}$ represents an edge type with an allowed start and end activity or entity type. We require that activity, entity, and edge labels be pairwise distinct.*

A provenance model defines the admissible types of activities and entities in a domain and how these types can be connected via edges of specific types. A provenance model also defines how to represent an execution of activities in the domain of the model. We term a record of such an execution an *execution trace*.

### B. Execution Traces

**Definition 2** (Execution Trace)**.** *Let $\mathbb{P} = (\mathcal{A}, \mathcal{E}, \mathcal{L})$ be a provenance model. An execution trace for $\mathbb{P}$ is a labeled directed graph $G = (V, E, T)$ with nodes $V$ and edges $E \subseteq V \times V$. Each node must be of one of the activity and entity types specified in the provenance model and each edge must fulfill the type constraints specified by $\mathcal{L}$. Finally, $T : E \to \mathbb{T} \times \mathbb{T}$ is a function mapping edges to intervals from a discrete time domain $\mathbb{T}$. We use $T(v_1, v_2)$ to denote the time interval associated by $T$ to the edge $(v_1, v_2)$ and $I_b$ and $I_e$ to denote the lower respective upper bound of an interval $I$.*

An execution trace is a graph in which the nodes are instances of the provenance model's activity and entity types and edges represent provenance dependencies. Each edge is annotated with a time interval indicating when the two connected nodes interacted: for example, the time interval during which a process (activity) was reading from a file (entity), or a time at which a query produced a result tuple.

We now use these definitions to instantiate the OS and DB provenance models and execution traces used in LDV.

### C. The Blackbox Process OS Provenance Model

We use a blackbox process provenance model to model the provenance of OS processes and their interaction with files. As the name suggests, we do not assume any knowledge of the inner workings of such processes. Processes are the only type of activity in this model and files are entities created and consumed by processes. We track three types of direct relationships: a process was *executed* by a process, a process has read from a file (*readFrom*), and a file was written by a process (*hasWritten*). An output of a application can be traced back to its input through these provenance links. However, connectivity in the graph does not necessarily imply dependency, as we will discuss in Section VI.

**Definition 3** (Blackbox Process Model)**.** *The blackbox process model $\mathbb{P}_{BB}$'s activities, entities, and edge types are:*

$$\mathcal{A} = \{process\} \qquad \mathcal{L} = \{readFrom(file, process),$$
$$\mathcal{E} = \{file\} \qquad\qquad hasWritten(process, file),$$
$$executed(process, process)\}$$

**Example 1.** *The top of Figure 2 shows part of an execution trace involving two processes $P_1$ and $P_2$. Process $P_1$ reads two files $A$ (during time interval $[1, 6]$) and $B$ (during $[7, 8]$).*

### D. The Lineage DB Provenance Model

On the DB side, activities are SQL statements and entities are tuples. Each SQL statement both *reads* tuples (its inputs) and *produces* tuples (the results of an update operation or a tuple returned by a query) We consider four types of SQL statements: `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements. Queries (`SELECT`) are connected to their result tuples in the execution trace. Similarly, modifications (`INSERT`, `UPDATE`, and `DELETE`) are connected to the tuple versions they produce. Queries are connected to all tuples from their input relations; modifications (insert, updates, and deletes) are connected to the original versions of the tuples that they modified. In Section VI we will define dependencies between tuples based on a standard DB provenance model.

**Definition 4** (Lineage Model)**.** *The lineage model $\mathbb{P}_{Lin}$ defines the following activities, entities, and edge types:*

$$\mathcal{A} = \{query, insert, update, delete\} \qquad \mathcal{E} = \{tuple\}$$
$$\mathcal{L} = \{hasReturned(\mathcal{A}, tuple), hasRead(tuple, \mathcal{A})\}$$

**Example 2.** *Consider the execution trace shown on the bottom of Figure 2.* Insert$_1$ *inserts two tuples $t_1$ and $t_2$ and* Insert$_2$ *inserts tuple $t_3$. Tuples $t_1$ and $t_3$ were read by query* Query *which returns two result tuples $t_4$ and $t_5$.*

## V. Combined Provenance Model

In order to support both OS and DB provenance, we combine OS and DB provenance models and introduce additional edge types that connect nodes across models.

**Definition 5** (Combined Provenance Model)**.** *Let $\mathbb{P}_O = (\mathcal{A}_O, \mathcal{E}_O, \mathcal{L}_O)$ and $\mathbb{P}_D = (\mathcal{A}_D, \mathcal{E}_D, \mathcal{L}_D)$ be OS respective DB provenance models. The combined model $\mathbb{P}_{D+O}$ is:*

$$\mathcal{A} = \mathcal{A}_O \cup \mathcal{A}_D \qquad \mathcal{E} = \mathcal{E}_O \cup \mathcal{E}_D$$
$$\mathcal{L} = \mathcal{L}_O \cup \mathcal{L}_D \cup \{run(\mathcal{A}_O, \mathcal{A}_D), readFrom(\mathcal{E}_D, \mathcal{A}_O)\}$$

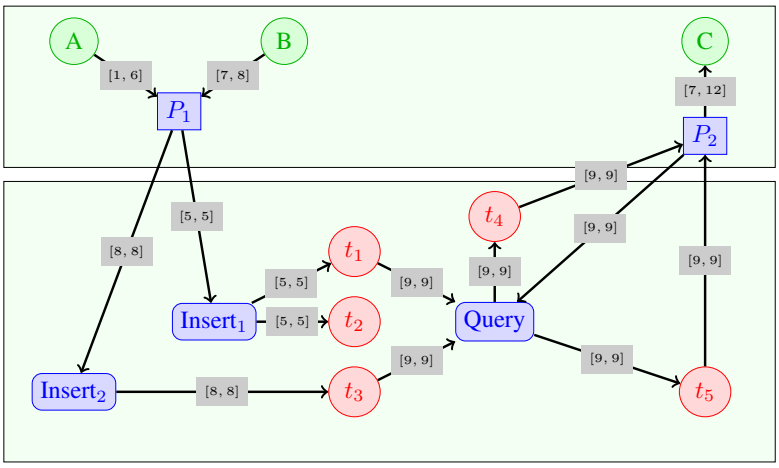


Fig. 2: An execution trace with processes and database operations

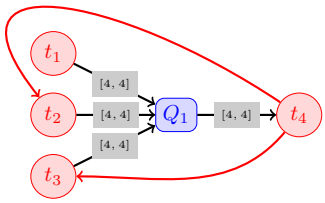


Fig. 3: $\mathbb{P}_{Lin}$ trace and data dependencies.

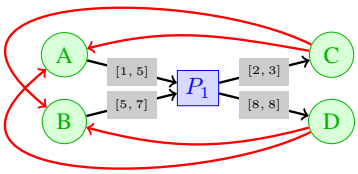


Fig. 4: $\mathbb{P}_{BB}$ trace and data dependencies.

*sales*

| | id | price |
|---|---|---|
| $\{t_1\}$ | 1 | 5 |
| $\{t_2\}$ | 2 | 11 |
| $\{t_3\}$ | 3 | 14 |

*result*

| | ttl |
|---|---|
| $\{t_2, t_3\}$ | 25 |

Fig. 5: Annotated relation *sales* and query result

A combined execution trace models the execution of a DB application including its processes, file operations, and DB accesses based on a OS and a DB provenance model.

**Definition 6** (Combined Execution Trace)**.** *Let $\mathbb{P}_{DB}$ and $\mathbb{P}_{OS}$ be DB and OS provenance models. Every execution trace for $\mathbb{P}_{DB+OS}$ is a combined execution trace for $\mathbb{P}_{DB}$ and $\mathbb{P}_{OS}$.*

**Example 3.** *A combined execution trace for the $\mathbb{P}_{Lin}$ and $\mathbb{P}_{BB}$ models is shown in Figure 2. This trace models the execution of two processes $P_1$ and $P_2$. Process $P_1$ reads two files $A$ and $B$, and executes two insert statements (at time 5 and 8 respectively). These insert statements create three tuple versions $t_1, t_2$, and $t_3$. Process $P_2$ executes a query which returns tuples $t_4$ and $t_5$. These tuples depend on tuples $t_1$ and $t_3$. Finally, process $P_2$ writes file $C$.*

## VI. Data Dependencies

The above definitions describe interactions of activities and entities in an execution trace of a provenance model, but do not model data dependencies, i.e., dependencies between entities. In our model, a dependency is an edge between two entities $e$ and $e'$ where a change to the input node ($e'$) may result in a change to the output node ($e$). Given a provenance model, dependency information may or may not be explicitly available; it depends upon the granularity at which information about entities and activities is tracked and stored. For instance, the blackbox provenance model $\mathbb{P}_{BB}$ operates at the granularity of processes and files and may not compute exact dependency information. Consider a process $P$ that reads from files $A$ and $B$ and writes a file $C$. File $B$ may be a configuration file that determines whether the process $P$ logs debug output—an option that has no effect on the content of file $C$. A $\mathbb{P}_{BB}$ execution trace cannot not inform about the absence of this dependency and thus we have to assume that each output depends on all inputs. On the contrary, a fine-grained DB provenance model such as $\mathbb{P}_{Lin}$ exactly determines the input tuples on which a query result tuple depends. We now formally state and explain dependency tracking in these models. By defining model-specific dependencies and temporal constraints we subsequently show how to exclude spurious dependencies and to infer additional dependencies.

### A. Lineage DB Dependencies

We use the Lineage provenance model for DB queries to determine dependencies between input and output tuples of DB operations in the $\mathbb{P}_{Lin}$ model. This model [5], [15] represents the provenance of a query result tuple $t$ as the set of tuples from the DB instance that were used to derive $t$. This set can be derived from provenance polynomial of tuple $t$ according to the semirings annotation framework [15]. In the semiring framework, tuples are annotated with elements from a commutative semiring which represent their provenance. Lineage is less informative than provenance polynomials, but is simpler and sufficient for our use case: determining dependency edges between tuples. Systems such as Perm [10] compute provenance polynomials (and thus also Lineage) on-demand for an input query. In the following we will use $Lin(Q, t)$ to denote the Lineage of a tuple $t$ in the result of a query $Q$.

**Example 4.** *Consider the sales table shown in Figure 5. The Lineage of each tuple in the sales table is a singleton set containing the tuple's identifier. The result of a query* `SELECT sum(value) AS ttl FROM sales WHERE price > 10` *is a single row with ttl $= 11 + 14 = 25$. The Lineage contains all tuples ($t_2$ and $t_3$) that were used to compute this results.*

We define data dependencies in the $\mathbb{P}_{Lin}$ model based on Lineage. We connect each tuple $t$ in the result of a query $Q$ to all input tuples of the query that are in $t$'s Lineage. Similarly, we connect a modified tuple $t$ in the result of an update to the corresponding tuple $t'$ in the input of the update.

**Definition 7** ($\mathbb{P}_{Lin}$ Data Dependencies)**.** *Let $G$ be a $\mathbb{P}_{Lin}$ trace. Let $Lin(s,t)$ denote the Lineage of tuple $t$ in the result of DB operation $s$, and let $t$ and $t'$ denote entities (tuples). The dependencies $D(G) \subset D \times D$ of $G$ are defined as:*

$$D(G) = \{(t,t') \mid \exists s : (t',s) \in E \wedge (s,t) \in E \wedge t' \in Lin(s,t)\}$$

**Example 5.** *Consider the trace shown in Figure 3 where $Q_1$ is the query from Example 4. Tuple $t_4$ depends on $t_2$ and $t_3$, because these tuples are in the Lineage of $t_4$ according to $Q_1$.*

### *B. Blackbox Process OS Dependencies*

As mentioned before, the applications that we track can maintain arbitrary internal state. Without static program analysis or dynamic instrumentation [26], [24] it is impossible to know which outputs depend on which inputs. Thus, we must assume (conservatively) that a file $f$ depends on another file $f'$ if there exists a process that reads from file $f'$ and writes file $f$. Recall that in the $\mathbb{P}_{BB}$ model a process may execute a child process. Thus, file $f$ also depends on $f'$ if it is connected to $f'$ through a path of process nodes.

**Definition 8** ($\mathbb{P}_{BB}$ Data Dependencies)**.** *Let $G$ be an $\mathbb{P}_{BB}$ trace, $f$ and $f'$ be entity (file) nodes in $G$, and $P_i$ be a process node. The data dependencies $D(G)$ of $G$ are defined as:*

$$D(G) = \{(f,f') \mid \exists P_1,\ldots,P_n : (f',P_1) \in E \wedge (P_n,f) \in E \wedge \forall i \in \{2,\ldots,n\} : (P_{i-1},P_i) \in E\}$$

The above definition states that there exists a data dependency between files $f$ and $f'$ if these two files are connected in the execution trace through a path of processes $P_i$ in which (a) the first process reads file $f'$ and the last process writes file $f'$, and (b) each process $P_i$ was executed by process $P_{i-1}$.

**Example 6.** *Consider the trace shown in Figure 4. Process $P_1$ reads files $A$ and $B$ and writes files $C$ and $D$. Thus, both $C$ and $D$ are data dependent on $A$ and $B$.*

### *C. Inferring Temporally Restricted Data Dependencies*

We next introduce a generic approach for inferring dependencies between entities in a combined execution trace based on the direct data dependencies between entities from the same model (e.g., a tuple is in the Lineage of another tuple) and the temporal annotations on edges in the trace.

Since the direct data dependencies of the individual provenance models may contain false positives (e.g., see the definition of data dependencies for $\mathcal{P}_{BB}$) developing an exact inference algorithm is challenging. However, we can leverage temporal constraints on interactions between nodes in an execution trace and intuitive assumptions on possible dependencies (e.g., an entity $A$ cannot depend on an entity $B$ if $A$ was produced before $B$ existed) to prune some dependencies. The result is an inference algorithm that is more precise while remaining *conservative*, meaning that while it may return a superset of the real dependencies, it will never a miss a dependency. For creating repeatability packages, conservatism is more important than preciseness, because it guarantees that sufficient data is contained in repeatability packages to reproduce results. Nonetheless, a high number of false positives would cause unnecessarily large repeatability packages. Thus, our goal is to formalize intuitive assumptions that are *conservative* and then derive inference rules that are *sound* and *complete* with respect to the set of all dependencies that fulfill the assumptions.

Our inference approach relies on a set of minimal and intuitive assumptions that we will formally state in the following. These assumptions are similar to those used in a formalization of the OPM provenance model [17]. That work demonstrated how to infer temporal constraints based on direct or indirect dependencies inferred over an OPM provenance graph. In contrast, we assume the temporal constraints as given (recorded when creating an execution trace) and use these annotations to restrict what edges have to be inferred. Similarly, Dey et al. [8] determine all possible orders of events that are admissible for an OPM provenance graph.

**Definition 9** (Dependency Axioms)**.** *We assume that any valid inferred data dependency $(e,e')$ for an execution trace $G$ has to fulfill the three conditions shown below. We use $D_{all}(G)$ to denote the set of all such dependencies for trace $G$.*

1) *Inferred dependencies must be informed by existing dependencies, i.e., if entity $e$ is dependent on entity $e'$, then either i) there exists a direct dependency $(e,e')$ in the trace, or ii) there exists a path between $e'$ and $e$ in the trace that does not pass through other entities and $e$ and $e'$ are from a different provenance model, or iii) there exists $e''$ so that dependencies $(e,e'')$ and $(e'',e)$ exist in the trace or are other inferred dependencies.*
2) *Execution traces model all interactions between activities and entities, i.e., there can be no data dependency from $e$ to $e'$ if there is no path from $e'$ to $e$ in the execution trace.*
3) *Dependencies do not violate temporal causality, i.e., the "state" of a node $n$ in the trace only depends on past interactions and transitively on the "state" of nodes $n'$ at the time of the interaction between $n'$ and $n$.*

To infer such dependencies we need to understand which direct interactions (edges) in the execution trace influence the state of a node $v$ at a time $T$. Based on assumption 3 introduced above, the state $S(v,T)$ of an activity or entity $v$ at time $T$ depends on all incoming interactions (incoming edges) it had up to time $T$. For example, for a process $p$ these are all the entities read by the process up to that time and any process that triggered $p$ before $T$ (if any). For a file $f$, this includes all processes that have written $f$ before $T$.

**Definition 10** (State)**.** *Let $v$ be a node in a combined trace*

*G. The state $S(v,T)$ of node $v$ at a time $T$ is defined as:*

$$S(v,T) = \{v' \mid (v',v) \in E \wedge T(v',v)_b \leq T\}$$

The state of a node can be used to infer dependencies between entities based on the temporal annotations on interactions in the execution trace which full the conditions of Definition 9. The state of an entity $e$ depends on an entity $e'$ at a time $T$ if 1) there is a path between $e'$ and $e$ in the execution trace, 2) adjacent entities from the same provenance model on this path are connected through data dependencies, and 3) the temporal annotations on the edges of the path do not violate temporal causality.

**Example 7.** *In the execution trace shown in Figure 4, there exists a path between file $B$ and file $C$ ($B \rightarrow P_1 \rightarrow C$). However, we cannot infer that $C$ depends on $B$, because file $C$ was written ($[2,3]$) by $P_1$ before it has read file $B$.*

**Definition 11** (Dependency Inference)**.** *Let $G$ be an combined trace for provenance models $\mathbb{P}_{OS}$ and $\mathbb{P}_{DB}$. The data dependencies of an entity $e \in G$ at time $T$ include all entities $e'$ such there exists a path $v_1, \ldots, v_n$ in the execution trace with $v_1 = e'$ and $v_n = e$ that fulfills the conditions stated below. Let $e_1, \ldots, e_m$ denote all entities on this path (where $e_1 = v_1 = e'$ and $e_m = v_n = e$). We use $D^*(G)$ to denote the set of all such dependencies.*

1) *For all $i \in \{2, m\}$, if $e_i$ and $e_{i-1}$ are from the same provenance model, then $(e_i, e_{i-1})$ in $D(G)$.*
2) *There exists a sequence of times $T_1, \ldots, T_n$ so that for each $i \in \{1, m-1\}$ we have $T_i \leq T_{i+1}$ and $T_i \leq T(v_i, v_{i+1})_e$.*
3) *For all $i \in \{2, n\}$, the node $v_{i-1}$ is in the state of $v_i$ at time $T_i$: $v_{i-1} \in S(v_i, T_i)$.*

Given assumption 2) an entity $e$ can only depend on entity $e'$ if they are connected in the execution trace. Also all adjacent entities on such a path should be directly data dependent on each other if they belong to the same provenance model (the 1st assumption enforced by condition 1 of the definition above). This guarantees that we do not introduce dependencies that do not hold based on the individual provenance models. Conditions 2) and 3) make sure that a dependency does not violate temporal causality, i.e., the information flow from $e'$ to $e$ complies with the temporal annotations.

**Theorem 1** (Inference is Sound and Complete)**.** *The inference rules of Definition 11 are sound and complete with respect to dependencies that fulfill the assumptions of Definition 9.*

*Proof:* Let $D_{all}(G)$ denote the set of all dependencies between nodes in $G$ and that are conformant with the three assumptions we have stated. Furthermore, recall that $D^*(G)$ denotes the set of dependencies inferred using Definition 11. We have to prove $D_{all}(G) \subseteq D^*(G)$, i.e., the rules are complete and $D^*(G) \subseteq D_{all}(G)$, i.e., the rules are sound.

$D_{all}(G) \subseteq D^*(G)$: Let $(e,e') \in D_{all}(G)$, i.e., $(e,e')$ is a dependency that fulfills assumptions 1 to 3. We have to show that $(e,e') \in D^*(G)$. There have to exist one or more paths between $e'$ and $e$, because if there is no path between $e'$ and $e$ in the trace then this would directly violate assumption 2. If conditions 1-3 of Definition 11 hold for one of these paths, then $e \in D^*(G)$. We will now incrementally construct such a path. Given that $(e,e')$ is a dependency that fulfills the three assumptions we know that there exists an entity node $e''$ so that the state of $e$ at a time $t$ contains $e''$ and the state of $e''$ at time $t$ contains $e'$. Otherwise, the dependency would violate temporal causality and/or assumption 2. WLOG let there be no other entity on the path between $e''$ and $e$ that caused $e''$ to be in the state (according to Definition 10), i.e., $e''$ is the "closest" entity to $e$ on this path. Let $v_1 = e'', \ldots, v_n = e$ be this path. Based on assumption 3 we can infer that condition 3 of Definition 11 holds for this path. Based on the definition of state (Definition 10) it follows that condition 2 holds too. Finally, if $e$ and $e''$ are from the same model, then $(e,e'')$ has to be a dependency in this model based on assumption 1 which means condition 1 of Definition 11 holds. Thus, $(e'',e)$ is a dependency in $D^*(G)$. Now the same argument can be applied to find a $e'''$ between $e'$ and $e''$. By induction we can construct the needed path between $(e',e)$ and it follows that $(e',e)$ is in $D^*(G)$.

$D^*(G) \subseteq D_{all}(G)$: Let $(e,e') \in D^*(G)$. Then we have to prove that $(e,e') \in D_{all}(G)$. In other words, $(e,e')$ does not violate any of the three assumptions (and, thus, would be in $D_{all}(G)$). This is obviously the case, because $e'$ and $e$ are connected through a chain of data dependencies, are connected in the execution trace, and temporal causality is not violated. ■

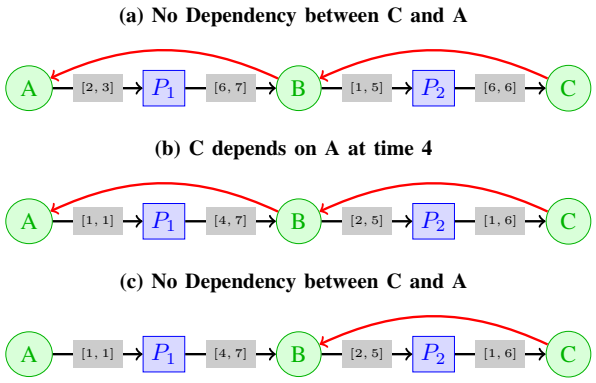


Fig. 6: Example traces with different temporal annotations

**Example 8** (Indirect Data Dependencies)**.** *Figure 6 shows several versions of the same execution trace with different data dependencies and temporal annotations. In trace 6a there exists a path between $A$ and $C$ and the entities on that path are connected through data dependencies. However, given the temporal constraints, $C$ cannot depend on $A$, because $P_2$ stopped reading $B$ before it was written by $P_1$. No matter what time sequence $T_1, \ldots, T_5$ is chosen, the third condition of the definition will fail for $v_i = B$. Trace 6b has different time annotations and in this trace $C$ depends on $A$ at time 4. For trace 6c there is no data dependency between $B$ and $A$.*

*Thus, we cannot infer that $C$ depends on $A$.*

## VII. Creating Execution Traces

Creating a re-executable package for a DB application's execution involves first *monitoring* the application in order to construct an execution trace and then *packaging* all objects accessed by that trace. We first discuss how to monitor an application and its DB interactions. Then, we discuss how to create a re-executable package.

### A. Creating Execution Traces for the BB Process OS Model

Recall that processes are the activities and files are the entities of the $\mathbb{P}_{BB}$ provenance model. We use the Unix *ptrace* system call to monitor the processes of a DB application and their interactions with files. This system call provides a mechanism to observe and trace another process (the "tracee") by examining all system calls executed by the tracee. For example, it can detect when a process opens a file by intercepting the *fopen()* system call, and detect when a process forks or execs another process by intercepting the *fork()* and *execve()* system calls, respectively. The advantage of using *ptrace* is that it allows monitoring an application without requiring any knowledge about the inner workings of the tracee.

The PTU system [22] uses *ptrace* to construct a provenance graph that connects process activities and file entities. We create the OS portion of an execution trace from a PTU provenance graph by recording a time interval for each interaction and attaching it to the edge in the provenance graph. For process-process edges, the time interval is a point in time, assuming instantaneous fork or exec of the child process. For process-file edges we assign the time interval between the times when the file was first opened and last closed by the process. Since in the $\mathbb{P}_{BB}$ model all files written by a process depend on all files read by the process (and its ancestors), it is not necessary to store these dependencies explicitly. Instead, we compute them on-the-fly when needed for inference.

### B. Creating Execution Traces for the DB Lineage Model

To create the DB portion of an execution trace, we need to trace dependencies between the result and input tuples of each DB operation. We use the Perm [10] system, which given an SQL query, rewrites the query to track the input tuples on which an output tuple of the query depends. For each executed SQL statement, we create a node in the execution trace which is then linked to nodes for all of its result tuples. We compute the provenance of the query with Perm and link each result tuple to the input tuples in its Lineage, i.e., its data dependencies according to the $\mathbb{P}_{Lin}$ model. We can also track the provenance of updates by translating them into queries [2].

While the use of a temporal DBMS would make tracking the provenance of SQL operations straightforward, we seek a general solution. Thus, we must overcome two problems to be able to create $\mathbb{P}_{Lin}$ execution traces with Perm. First, each modified tuple generated by an update operation depends on a previous version of the tuple that is no longer available. To address this problem, we retrieve the provenance for the update before executing it [2]. Second, in an application that performs DB updates, we must be able to distinguish in an execution trace between the versions of a tuple used by different queries. We address this problem by extending the schema of each relation accessed by a DB application to store version information. Specifically, we add the following attributes to each such relation, and modify SQL statements to modify these attributes: attribute `prov_rowid` stores a unique identifier for each row in the database; `prov_v` stores a timestamp for the latest update to the tuple; and `prov_usedby` and `prov_p` store unique query and process identifiers which we use to link tuples to activities in an execution trace.

### C. Creating Combined Execution Traces

We create combined execution traces by connecting execution traces for the $\mathbb{P}_{BB}$ and $\mathbb{P}_{Lin}$ models. In particular, we introduce edges between processes and SQL statements and between the query result tuples and processes, i.e., the $run(\mathcal{A}_{OS}, \mathcal{A}_{DB})$ and $readFrom(\mathcal{E}_{DB}, \mathcal{A}_{OS})$ edges. To this end, we instrument the client library of the DBMS to transparently audit each such interaction. Whenever a process connects to the DBMS, we assign a unique *process_id*. We also assign each executed SQL statement a unique *query_id*. For each executed SQL statement we apply the technique discussed above to retrieve its provenance. The modification to relations required by this technique is performed whenever the relation is first accessed by the DB application.

Our prototype implementation uses an instrumented version of *libpq*, the C language client interface of PostgreSQL. We intercept Select, Insert, Update, Delete statements sent to the DBMS and modify each statement to compute its result tuples and return all tuples on which the result tuples depend (their provenance). For queries, this modification involves adding the `PROVENANCE` keyword as supported by Perm (recall that Perm is an extension of PostgreSQL). For modifications (insert, update, delete) we use the reenactment techniques pioneered in GProM [2], which enable us to use a query to track the statement's provenance. We aspire in future work to instrument generic DB client interfaces such as ODBC or JDBC.

### D. Creating Virtualization Packages

To enable successful (partial) re-execution, a package should include all required binaries, data files, and library dependencies; the execution trace; the relevant DB subset; and the DBMS (or should be capable of sandboxing it). Similar to application virtualization systems such as CDE, we create a package by extracting file path parameters from system calls that were intercepted via *ptrace*. The files at the respective paths are copied into a package root directory, while recreating the sub-directories and symbolic links of the original file's location within the root directory. This creates a *chroot*-like environment for application re-execution on compatible architectures. In addition, we also include a serialization of the execution trace for the application into the package. As mentioned in Section II, we support two options for packaging the relevant part of a DB for an application.

**Server-included** For this packaging option, we use *ptrace* to determine the DB server binaries and associated dynamically-linked libraries (e.g., user-defined functions) and include them in the package. We do not, however, include any of the raw data files of the DBMS. Instead we use the execution trace to determine which tuples versions are relevant for the application and store these tuple versions in CSV files in the package. A tuple version is relevant to the application if it is not created by application itself (no incoming edge in the execution trace) and the state of an activity in the execution trace depends on it (this implies that a data dependency exists from a result tuple of an SQL statement to this tuple). Recall that we have discussed in Section II why tuple versions created by the application itself should be excluded from the package.

As explained above, we track tuple versions by adding attributes to each table accessed by the DB application; immediately compute the provenance for every operation to gather tuples that need to be included in the package; and write these tuples to files on disk. We create one CSV file for every table accessed by the application. Our current prototype implementation uses an in-memory hash table to avoid adding duplicate entries to such a file. (If this hash table exhausts available memory, we could apply a disk-based duplicate removal operation as a post-processing step, e.g., by using standard disk based sorting methods.) Table I summarizes our approach for creating a server-included package. *The server-included packaging option is only applicable if sharing the server does not violate any license agreement (open source DBMS or the user is only sharing with researchers that have a license) and the user has access to the server files.*

**Server-excluded** For this packaging option, we also instrument the DB client interface. For each query executed by the application, we create a file in the package and store all results of this query in the file. These materialized query results enable us to replay the DB server's responses when re-executing the package. For this packaging option we do not need to include the DB server in the package. *The server-excluded packaging option is applicable in all scenarios and does not require any access to the server other then through its client interface.*

**Trade-offs** As mentioned above, there are some use cases where only the server-excluded option is applicable. For applications where both options are available, users can choose the one that best fits their needs. As we will demonstrate in our experimental evaluation, package size and performance varies significantly for these options based on the query and update workload run by the application. A server-included package has the advantage that the DB content may be used for similar experiments (as long as they access a subset of the data that was included in the package). This is not supported by server-excluded packages which allow faithfull re-execution of an application (or parts thereof), but neither support changes to queries nor the application.

## VIII. Repeating Executions with Packages

The methods used to re-execute an LDV package vary according to its type.

A **server-included** package is re-executed much as in an application virtualization system: during application re-execution, file system calls are redirected to files within the package and DB operations to the server within the package. The latter redirection is achieved by intercepting connection calls to the DB from the client library, and connecting to the default DB in the package. Since the DB server has been configured during package creation with all tuples required to answer the application queries, and we restore these tuples before any query occurs, the DB application can be repeated.

A **server-excluded** package must be replayed in the same order as in the original execution trace. We do not preserve the execution trace, but only the elements required for re-execution. Thus, when replaying the server-excluded package, the LDV system continues to audit calls to the DB at the client library. It redirects connection requests as well as read (query) and write (update) calls to the simulated DB. We match each incoming request against the requests recorded during package generation (stored in a lightweight DB). We ignore SQL Update statements and calls related to connection handling. In the case of a read request (query), the specific memory buffers used by the DB client library are substituted with the recorded response from the package. As long as read and write request follows the order and specification stored in the package (which is guaranteed if we fully or partially re-execute a package), then this guarantees repeatability.

## IX. Experiments

We evaluate the LDV system from the perspectives of performance, usability, and generality. We focus on the LDV prototype described in Sections VII and VIII. We evaluate this system's audit and replay performance, and also compare the size of LDV packages with those obtained when using the PTU virtualization approach, a provenance-enabled virtualization approach, and the virtual machine approach. Our prototype is highly usable in that to begin auditing an execution, users need to install LDV and simply prepend their application executable with an *ldv-audit* command. Similarly, users who which to replay must install LDV and an replay a shared package by prepend their application executable with an *ldv-exec* command. Among the compared packaging options, our results show that LDV performance and package size are inversely related.

### A. Datasets and Workloads

We use the TPC-H benchmark [25] with a scale factor 1 (1GB) for our experiments. This benchmark involves a suite of 22 decision support queries. In this evaluation we use a DB application that executes the following steps:

- **Insert**: Insert 1000 tuples into tables *orders* (according to the update workload specified by TPC-H)
- **Select**: Run 10 instances of one of the workload queries (see below)

TABLE I: OS and DB interposition for server-included use case

| | Operating system | DB |
|---|---|---|
| **Method** | Use *ptrace* to intercept system calls | Rewrite DB client library to intercept communication between server and application |
| **Monitoring** | On system call interception, *record* path parameter | On DB query interception, *record* statements and tuples that affect result |
| **Replaying** | On system call interception, *replace* path parameter with replayed data | On DB connection, *restore* DB from recorded tuples |

- **Update**: Update 100 tuples in table *orders*.

**Workload** We do not use the original TPC-H queries in our experiments: as most TPC-H queries touch large fractions of the tables and return a small number of results, considering only those queries would not allow us to study trade-offs among package options for different output package size to provenance size ratios. Indeed, those queries would result in an unnatural preference for the server-excluded packaging option, since replaying a small number of result tuples is very fast. To enable a broader comparison, we define queries that span a wide range of output package size to provenance size ratios.

Table II shows the queries (Q1 to Q5) used in the experiments. We generate different versions for each query $Qi$ by varying the selectivity (*Sel.*) of the query and use Qi-j to denote the variant that uses the $j^{th}$ parameter as shown in column *PARAM* of Table II. Query Q1 is a simple selection. Its variants (Q1-1 to Q1-5) have selectivities ranging from 1% to 25%. Queries Q2 and Q3 are slightly more complex, using join operators (and aggregation in case of Q3). We vary selectivity by changing the number of leading 0s for the parameter. Query Q4 aggregates the result of a selection to find the average quantity and total cost per order for orders with a certain supplier. We vary selectivity by changing the range of $l\_suppkey$.

**Measures** We measure package size, audit performance, and replay performance of those queries as part of the aforementioned application. To create a baseline for comparison, we measured the execution time of the application using an standard PostgreSQL server. We used PTU to audit the application and create a portable software package without DB provenance. In this configuration we start the DB server as the first step of the experiment and shut it down before the experiment is finished. This ensures that the server and its data files will be included in the package.

### B. Audit Performance

In our first experiment we measure the execution time of an applications when it is audited to create a package. The results of this experiment for the application with query Q1-1 are shown in Figure 7a.

*1) Server-included:* The monitoring and package generation causes overhead for all 3 steps of the application. The overhead in the *Select* step results from the need to query provenance, persist tuples in the provenance that should be included in the package, and updating of accessed tuples to implement the versioning described in Section VII-B. In the first query (cold cache), LDV needs to write almost all accessed tuples to external storage. Subsequent queries do not need to record tuples which have already been written to disk again, but they still need to run queries to retrieve the provenance and updates to keep the version information up-to-date. Similar provenance queries are required for DB update operations, which results in the overhead observed for the *Update* step. For insert statements, there is no need to run additional provenance queries; hence the low overhead for the *Insert* step.

*2) Server-excluded:* This scenario shows a lower overhead than the server-included scenario as no extra query is needed to retrieve provenance. The small overhead is due to writing query result tuples to disk.

### C. Re-execution Performance

Figure 7b compares the replay performance of LDV server-included and server-excluded packages with a non-audited execution. The server-included package has large overhead for DB initialization, since LDV needs to create the DB using the tuples included in the package. Query performance on a server-included package is the same as, or better than, non-audited execution, because the relevant DB subset included in the package may be significantly smaller than the original DB, leading to better query performance.

In almost all experiments, server-excluded packages result in lower execution times than do server-included packages and normal execution. This result is explained by the fact that during re-execution of a server-excluded package, query results are directly read from disk, which takes time linear in the size of the query result—in contrast to query execution which is superlinear for most queries. As is evident from Figure 8b, the extreme case is query Q3, which returns only one result tuple. We also measured the performance using a virtual machine as a packaging option. This approach has the highest overhead.

### D. Query Performance

We next measure the effect of auditing and replay on the execution times of queries. Figure 8 shows the execution time for our workload queries with different selectivities. Execution time increases linearly as the queries scan a larger number of tuples from the DB. While the relative overhead is quite large, it is relatively stable across selectivities.

### E. Package Size

To explore the improvement of LDV packages over repeatable software packages that contain a full DB, we compare the sizes of LDV and PTU packages. A PTU package contains all the necessary binaries, libraries and files required to re-execute the application including all files accessed by the DB

TABLE II: The 18 TPC-H benchmark queries used in our experiments

| Queries | SQL | PARAM | Sel. |
|---|---|---|---|
| Q1-1 to Q1-5 | SELECT l_quantity, l_partkey , l_extendedprice , l_shipdate , l_receiptdate FROM lineitem WHERE l_suppkey BETWEEN 1 AND PARAM | 10, 20, 50, 100, 250 | 1%, 2%, 5%, 10%, 25% |
| Q2-1 to Q2-4 | SELECT o_comment, l_comment FROM lineitem l, orders o, customer c WHERE l.l_orderkey = o.o_orderkey AND o.o_custkey = c.c_custkey AND c.c_name LIKE '%PARAM%'; | 0000000, 000000, 00000, 0000 | 66%, 6.6%, 0.66%, 0.06% |
| Q3-1 to Q3-4 | SELECT count(∗) FROM lineitem l, orders o, customer c WHERE l.l_orderkey = o.o_orderkey AND o.o_custkey = c.c_custkey AND c.c_name LIKE '%PARAM%'; | 0000000, 000000, 00000, 0000 | 66%, 6.6%, 0.66%, 0.06% |
| Q4-1 to Q4-5 | SELECT o_orderkey, AVG(l_quantity) AS avgQ FROM lineitem l, orders o WHERE l.l_orderkey = o.o_orderkey AND l_suppkey BETWEEN 1 AND PARAM GROUP BY o_orderkey; | 10, 20, 50, 100, 250 | 1%, 2%, 5%, 10%, 25% |

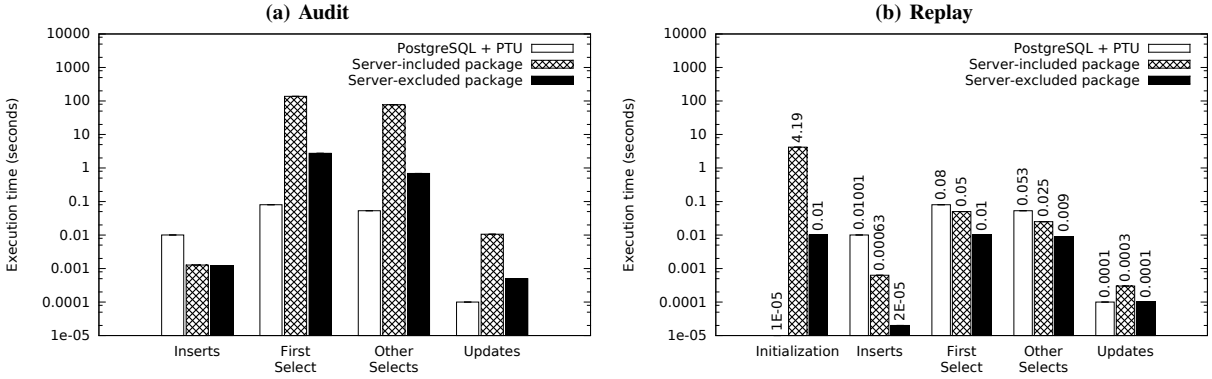


Fig. 7: Execution time of each step in an application with query Q1-1

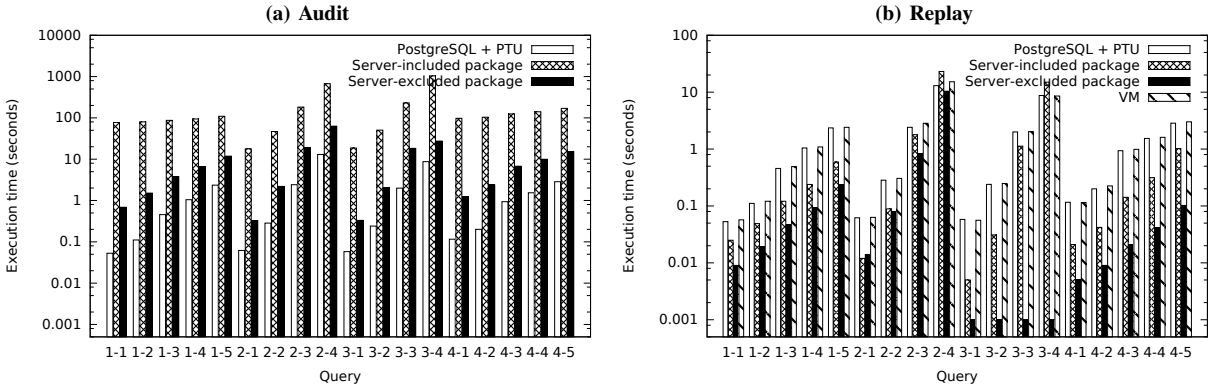


Fig. 8: Execution time for each query, during audit (left) and replay (right)

TABLE III: Package Contents: PTU packages contain all data files of the full DB, whereas server-included LDV packages contain the data files of an empty DB.

| Package type | Software binaries | DB server | Data files | DB provenance |
|---|---|---|---|---|
| PTU | ✓ | ✓ | ✓(full) | ✗ |
| LDV server-included | ✓ | ✓ | ✓(empty) | ✓ |
| LDV server-excluded | ✓ | ✗ | ✗ | ✓ |

server in the application execution, i.e., the server binaries and data files. An LDV package contains DB provenance for re-execution, the DB server binaries, and an empty data directory in the server-included scenario (Table III).

Figure 9 shows the sizes of the PTU, server-included, and server-excluded packages constructed for the queries listed in Table II. Server-included LDV packages are significantly smaller than PTU packages, because they contain only those tuples needed to re-execute the application—which, for these queries, is at most ∼25% of all tuples. Server-excluded LDV packages are often yet smaller, because they contain only the query results—which, for many of our experiment queries, are smaller than the tuples required for re-execution. However, recall that server-excluded packages have less flexibility than do server-included packages.

### F. Comparison with the Virtual Machine Approach

We compare a virtual machine image (VMI) with the server-included and server-excluded LDV approaches. The VMI is created based on a bare-bone Debian Wheezy 64bit VMI on which we install the DB server and the experiment binaries as in Section IX-A. We use "apt-get" to install a DB server, and "scp" to copy all DB files and source code for the experiment from our machine. Using the created VMI, we run the same application to compare the size and performance of this VMI

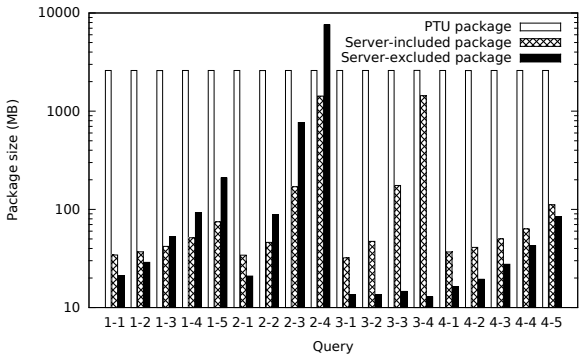


Fig. 9: LDV packages are significantly smaller than PTU packages when queries have low selectivity.

and the LDV packages. The VMI is 8.2 GB: 80 times larger than the average LDV package (100MB). To evaluate runtime performance, we instantiate this VMI using the same number of cores and memory as in our machine to execute our queries. Recall that Figure 8b shows that re-executing these queries in a VM is slightly slower than a non-audited PostgreSQL execution, and significantly slower than LDV packages.

## X. Conclusions

We introduced a light-weight DB virtualization (LDV) system that can permit sharing and re-execution of applications that perform DB operations. This system uses data collected via application monitoring to create re-executable packages that include an application, its dependencies (data files, relevant DB tuples), and a combined execution trace. Such packages can be used to repeat an application or part of an application in a different environment.

Our LDV framework features an innovative integration of distinct OS and DB provenance models, and new methods for inferring data dependencies that cross model boundaries. The resulting system creates execution traces according to this framework and uses these traces to determine which data needs to be included in a repeatability package. It leaves to the user the choice of whether the package should include the DBMS. Our prototype implementation integrates the PTU (OS) and Perm (DB) provenance systems. In future work, we plan to integrate with the DB-independent GProM [2] middleware.

## Acknowledgements

This material is based upon work supported by the National Science Foundation under Grants ICER-1343816 and SES-0951576, and by the US Department of Energy under contract DE-AC02-06CH11357. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

## References

[1] Y. Amsterdamer et al. Putting Lipstick on Pig: Enabling Database-style Workflow Provenance. *PVLDB*, 5(4), 2011.

[2] B. Arab, B. Glavic, et al. A generic provenance middleware for database queries, updates, and transactions. In *Proceedings of TaPP*, 2014.

[3] N. Balakrishnan, T. Bytheway, et al. Opus: A lightweight system for observational provenance in user space. In *TaPP*, 2013.

[4] C. T. Brown. Some myths of reproducible computational research. http://ivory.idyll.org/blog/2014-myths-of-computational-reproducibility.html.

[5] J. Cheney et al. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases*, 1(4), 2009.

[6] F. Chirigati and J. Freire. Towards integrating workflow and database provenance. In *Provenance and Annotation of Data and Processes*. 2012.

[7] F. S. Chirigati, D. Shasha, and J. Freire. Reprozip: Using provenance to support computational reproducibility. In *TaPP*, 2013.

[8] S. C. Dey, S. Riddle, and B. Ludäscher. Provenance analyzer: Exploring provenance semantics with logic rules. In *TaPP*, 2013.

[9] J. Freire and C. T. Silva. Making computations and publications reproducible with vistrails. *Computing in Science and Engineering*, 14(4), 2012.

[10] B. Glavic et al. Perm: Processing Provenance and Data on the same Data Model through Query Rewriting. In *ICDE*, 2009.

[11] B. Glavic et al. Using sql for efficient generation and querying of provenance information. In *In search of elegance in the theory and practice of computation: a Festschrift in honour of Peter Buneman*. 2013.

[12] C. A. Goble and D. C. De Roure. myExperiment: social networking for workflow-using e-scientists. In *Proceedings of the 2Nd Workshop on Workflows in Support of Large-scale Science*, 2007.

[13] P. J. Guo et al. CDE: using system call interposition to automatically create portable software packages. In *USENIX Annual Technical Conference*, Portland, OR, 2011.

[14] B. Howe. Virtual appliances, cloud computing, and reproducible research. *Computing in Science & Engineering*, 14(4):36–41, 2012.

[15] G. Karvounarakis and T. Green. Semiring-annotated data: Queries and provenance. *SIGMOD Record*, 41(3):5–14, 2012.

[16] K. Keahey et al. Virtual workspaces for scientific applications. *Journal of Physics: Conference Series*, 78(1), 2007.

[17] N. Kwasnikowska, L. Moreau, and J. Van den Bussche. A formal account of the open provenance model. *journal*, 2010.

[18] S. Lampoudi. The path to virtual machine images as first class provenance. *Age*, 2011.

[19] T. Malik, L. Nistor, and A. Gehani. Tracking and sketching distributed data provenance. In *International Conference on eScience*, 2010.

[20] L. Moreau and P. Missier. Prov-dm: The prov data model. *http://www.w3.org/TR/2013/REC-prov-dm-20130430/*, 2013.

[21] K.-K. Muniswamy-Reddy, U. Braun, D. A. Holland, P. Macko, D. Maclean, D. W. Margo, M. I. Seltzer, and R. Smogor. Layering in provenance systems., 2009.

[22] Q. Pham, T. Malik, and I. Foster. Using provenance for repeatability. In *TaPP*, 2013.

[23] Q. Pham, T. Malik, and I. Foster. Auditing and maintaining provenance in software packages. In *IPAW*, 2014.

[24] M. Stamatogiannakis et al. Looking inside the black-box: Capturing data provenance using dynamic instrumentation. In *TAPP*, 2014.

[25] Transaction Processing Performance Council. TPC-H benchmark specification. *Published at http://www.tcp.org/hspec.html*, 2008.

[26] M. Zhang et al. Tracing Lineage beyond Relational Operators. In *VLDB*, 2007.