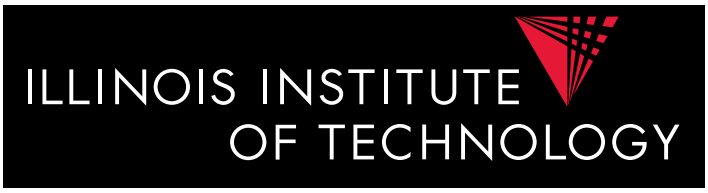

LDV: Light-weight Database Virtualization

Quan Pham, Tanu Malik, Boris Glavic, Ian Foster

IIT DB Group Technical Report
IIT/CS-DB-2014-03

2014-10

http://www.cs.iit.edu/~dbgroup/

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, which allows users to share their applications involving a relational database. Currently, users share such applications through companion websites or virtual images. However, neither of these methods provide the ability to easily re-execute the shared application. LDV monitors the execution of an application and its interactions with a database, and merges the resulting operating-system provenance with fine-grained database provenance. The system provides two options for packaging the application binaries, data files, and part of the database that is relevant to the application. The first packaging option uses fine-grained database provenance to determine the slice of the database needed to repeat the application and includes it along with the database server in the package. The second option records the queries and their results issued by the application and replays the queries and results to repeat the experiment. We present an approach for inferring data dependencies based on temporal information about the interactions of the application's processes with the database. This approach is used to determine relevant parts of a package when the user is interested in repeating part of an application. We experimentally study the trade-off between these two options in terms of package size, monitoring overhead, and overhead when repeating the application. Furthermore, we demonstrate that the system can successfully reproduce applications with database access.**

I. Introduction

Most current science projects involve large amounts of computation and data. These computations consist of complex workflows that involve multiple binaries and data files. Relational database systems are used as a persistence mechanism for intermediate and final results and to share data among researchers and across workflows. Furthermore, many publicly available scientific data repositories are stored in relational database, e.g., gene databases. In this work we refer to workflows involving a database as *database applications*. Sharing and repeating database applications is critical for verifying experimental results (e.g., repeat a computational experiment as described in a publication) and to allow scientist to reuse complex workflows that have been created with a lot of manual effort. The light-weight database virtualization approach we present in this work is a comprehensive solution for sharing database applications as self-contained packages including programs, data, and provenance.

To date the most widely adopted methods for computational reproducibility are building *companion websites* and/or *provisioning virtual machine images*. The former is quick and easy, providing access to code and data, but provides no execution mechanism and therefore doesn't guarantee repeatability [1]. The latter is a heavy-weight approach that may or may not provide access to code and data, but, if provisioned correctly, guarantees execution and ensures repeatability.

Application virtualization is a light-weight method that has recently emerged as an alternative for computational reproducibility. In this method system calls are traced during application execution and all code, data and software dependencies are copied into a *package*. The resulting package consisting of binaries, data and a software environment can be run on any compatible machine without installation, configuration, or root permissions. There are several advantages of application virtualization. By capturing only the necessary data and software environment, application virtualization tools, such as CDE [14] create packages that are more light-weight than virtual machine images. By automating package creation, self-contained packages make sharing scientific results as easy as building companion websites. Finally, for deterministic computations, re-running the packages guarantees repeatability of results.

In spite of the advances made, sharing and repeating database applications remains a challenging problem for the following reasons.

- An application may only use a small portion of the data stored in a database. For example, consider an application that analyzes local weather in Chicago in 2014 using a global weather database for the last decade. To create a repeatability package of reasonable size, one has to determine which parts of the database are actually used by the application. Standard database techniques such caching of disk resident data, data independence, write-ahead logging, and delayed writing of changed blocks to disk make it hard to track which data is modified and used by the application.
- The user executing the application may have limited access to the database server and its data files (e.g., the database server is run by a third party). Even if the user has control over the server, sharing the server binaries and data files may not be an option, because of licensing issues and data usage policies.

[1] For repeatability, we restrict ourselves to deterministic computations.

- Databases are often shared among multiple users and across many application. Thus, to re-execute an application, the database state, as of the start of the application, has to be restored.
- Provenance can be used to understand a shared application. While database and application provenance are well understood, combining these two types of provenance remains challenging.

None of the above mentioned methods - companion websites, VM images, and application virtualization - addresses these challenges. There is no automatic mechanism for capturing and linking application and DB provenance, these approaches provide no means for determining which data is relevant for an application, they do not solve the issue of resetting a database to a previous state, and do not address the licensing problem of sharing the binaries of commercial database servers.

For example, application virtualization is currently limited to local applications that do not communicate to server processes, such as a web server or a database server. In fact, when an application communicates with a database server, the technique can *atmost* record the communication between the client and database server. This is not sufficient for determining which data was used by the application (and, thus, should be included in the package) and to be able to reset the database to its state before application execution started. Temporal databases provide a solution for the later problem, but not for the earlier. Virtualization can ensure reproducibility if the user has control over the database server, the server is started as part of the application (thus the virtualization system can capture a consistent state of the database files on disk and the server binaries) and shutdown before the capture mechanism is stopped. However, this will include complete database into the resulting package.

The goal of this work is to improve computational reproducibility for database applications. The *light-weight database application virtualization* (*LDV*) approach we present in this work addresses the aforementioned challenges. In particular, LDV enables users to easily create a light-weight database application virtualization (LDV) package, consisting of code, data, software dependencies, a slice of the database with which is required for re-execution, and provenance. If shared with a 2^{nd} party, the application runs in exactly the same way as it did for the original user, without requiring installation or configuration of a database server at the target site. The provenance included in a package can used to understand data dependencies across the application and database, and to determine which parts of a workflow are needed to re-create a partial result.

II. Light-weight Database Virtualization

We describe our approach by means of an example reproducibility task. Consider a user Alice who has been using a database in the past to conduct her experiments. She has finally developed a database application which reads some input data and outputs some analysis that she believes is interesting to share with Bob (Figure 1). Alice would preferably like to share this application in the form of package P with Bob, who may want to re-execute the application in its entirety or may want to validate, just the analysis task, or provide his own data inputs to examine the analysis result.

If Alice wants Bob to re-execute and build upon her database application, then Bob must have access to an environment that consists of application binaries and data, any extension modules that the code depends upon (e.g., dynamically linked libraries), a database server and a database on which the application can be re-executed. Ideally, it would be useful if Alice's environment can be virtualized and thus automatically set up for Bob.

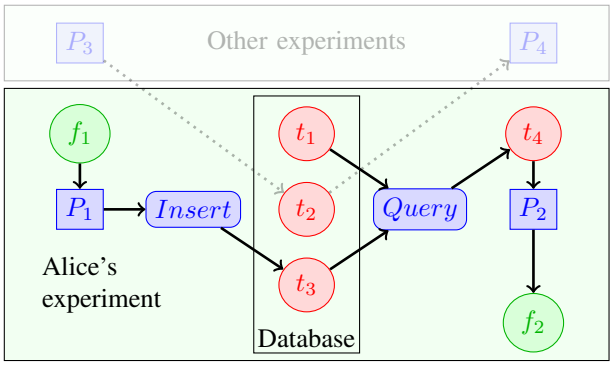

Fig. 1: Alice's experiment with processes P_1 and P_2 uses tuple t_1, inserts tuple t_3, creates final output f_2. Dumping database produces redundant tuple t_2. Capturing Alice's experiment in its fullness makes t_3 redundant. Only t_1 is needed for the experiment to execute.

If we assume that Alice's application consists of set of modules that read data from files and/or retrieve data from a database, and write data to files and/or write data to a database, the database server is accessed through standard SQL language commands, and Alice executes her application through a command line script, providing a single entry point for monitoring the application, then several questions arise with respect to virtualizing her environment. In particular:

•How do we include the necessary and sufficient data, i.e., data that corresponds to her last experiment in the virtualized environment? As Figure 1 shows there are data (tuples) in the database that are not part of the current experiment and if included in the package, may increase the size considerably, not leading to a light-weight virtualized environment.

•How can a self-contained package be created so that Bob does not have to install or configure a database server?

•How can Bob re-execute the database application, partially, or wholly, without communicating with Alice's database server?

We describe our primary contributions in addressing these questions, and also describe an overall organization map for the paper. In summary we monitor database applications and combine database and application provenance to determine necessary and sufficient data. We describe how this data can be

packaged easily with proprietary and non-proprietary database servers, and we describe how Bob can efficiently re-execute the database application.

Linking Provenance Models (*Section IV*) As a first major contribution we study how to combine database and workflow provenance graphs. We introduce a generic model (that can be represented in the standard PROV [20] model) and map standard database and workflow provenance models into this framework. Our framework extends the underlying black-box OS model and fine-grained DB provenance models with additional links between OS-side activities and DB-side operations (e.g., a process executes an SQL statement). We can infer data dependencies within and across the OS and DB part of a provenance graph using data dependencies available in the individual models and temporal annotations on interactions between elements of such graphs. This model enables us to determine which part of the database are required to re-execute a workflow and, thus, to create a light-weight package. Furthermore, we use the model to determine parts of a workflow needed for correct partial execution (if requested by the user) and to answer simple provenance queries (does data item d depend on data item d'). We apply this framework to two concrete provenance models (the models used by the provenance systems

Monitoring Database Applications (*Section VII*) To monitor database applications we need to keep track of operating system processes, their interaction with files, and their interaction with the database (SQL statements). Furthermore, we need to record provenance for all these interactions. For monitoring processes and their file interactions we can use existing solutions such as PTU [22], CDE-SP [23], OPUS [5], and others. Most of these approaches monitor system calls to determine when processes are spawned and when a process opens or closes a file. This information is then used to construct a provenance graph for the application execution. However, we also need to monitor the interaction between processes and the underlying database. We capture this interaction at the level of SQL statements and their results, i.e., we capture the code of SQL statements (queries and DML statements) executed by a monitored process, their results, and keep track of which SQL statement was executed when. While monitoring a database application we interactively construct an *execution trace* (a provenance graph of our combined provenance model) for the workflow.

Server-included and Server-excluded Virtualization Packages (*Section VII-D*) Similar to virtualization approaches such as CDE, LDV can create repeatability packages for a workflow that include binaries (and their dependencies such as dynamically linked libraries) and data files. We provide two approaches for packaging database related content. The first option, which we refer to as *Server-included*, is to include the server itself into the package and ship the relevant slice of the database needed to repeat the workflow. This option is possible when the server binaries can be shared (no legal issues and user can access the server binaries). In the second option (*Server-excluded*), we record the results of queries issued by the workflow and include these results in the package. This option does not require access to the server binaries or any additional privileges apart from the privileges that are require to execute the SQL statements issued by the workflow. As the name suggests, a Server-excluded package does not contain the database server.

Using Packages to Repeat Database Applications (*Section VIII*) A LDV package can be shared to repeat the whole or part of the workflow on a different machine. We do not have to enforce any additional requirements apart from the requirements that virtualization system impose. That is if the target machine can be used to execute a standard virtualization package, then this machine can be used to execute an LDV package. To repeat a workflow packed using the server-included option, we set-up the included database server create an empty database, start-up the server, and load all data included in the package. Whenever a process issues an SQL statement during the execution of the package, we reroute this request to the database server included in the package. To repeat a server-excluded package we ignore all DML operations. Queries are not executed using a database server, but instead their recorded results that are included in the package are returned. This guarantees that the workflow will observe the same results as in the original execution. LDV also supports partial execution of a package for which we use our inference mechanism on provenance graphs to determine which processes and data are required to reproduce a partial result.

Prototype Implementation (*Section IX*) We have developed a prototype implementation of LDV that glues the PTU OS provenance system [22] and the Perm DB provenance system [12], [11]. Our current prototype instruments the client interface of the Perm database[2] to monitor interactions between the OS and DB side. Perm is used to compute fine-grained provenance for database queries to determine the part of the database that is relevant to the workflow if the Server-included packaging option is chosen. We have expanded PTU to incorporate databases into the resulting packages using the two packing options mentioned above. In the future we plan to support additional database systems such as the GProM generic database middleware.

III. Related Work

While there are several standards prescribed for scientific reproducibility, there is no clear and concise definition. The following description is consistent with several works [17], [22], [10], [8], [15]: "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." In several cases, further validation is required for an experiment to be considered reproducible, such as by generalizing the scientific results by applying them to new data sets,

[2]Perm is an extension of the PostgreSQL open source database

verifying how they behave under different parameters, and reusing and extending the experiment [22].

Virtualization Keahey et al.[17] was the first to propose useing VMs to encapsulate large, complicated stacks of scientific software so they can be deployed across supercomputing centers without the need to install each of the software packages individually in every new environment. This technique has also been applied in the context of cloud computing by Howe et al. [15]. However, as described in [1] and [18], VMs are space inefficient and not descriptive enough to enable validation (e.g., no provenance). Application virtualization tools such as CDE [14] 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 Unix debug utilities are ReproZip [8] in the context of VisTrails workflow system [10] and Research Objects in the context of MyExperiment [13]. However, none of these approaches address the problem of packaging a database or support database provenance. Our LDV system packages databases and tracks both OS-level and DB-level provenance.

Unified Provenance Models Database and OS provenance have been modeled differently due to the inherent mismatch in their computational and data models. Most OS provenance models track provenance at the granularity of processes and file [5], [22], [23], [19]. Processes are considered as black-boxes where all outputs of the process dependent on all inputs. Notable exceptions are approaches which use dynamic instrumentation to compute fine-grained provenance for binary programs [26], [24]. Database provenance [6], [16] is fine-grained (usually at the granularity of tuples) and provides strong guarantees about which outputs of a query depend on which inputs. Some proposals of unifying these different types of provenance have been introduced in related work. Cheney et al. define a *common* graph model for addressing cross-layer provenance for database applications [2]. They propose a fine-grained model based on program dynamic instrumentation. Another recently proposed unified model approach in the context of e-commerce database applications is to model data-dependent processes as finite state machines and use linear temporal logic to model provenance [9]. Amsterdamer et al. [3] model workflows as collections of programs written in a subset of Pig corresponding to nested relational algebra and capture fine-grained provenance using database provenance model. Unified models enable precise description of provenance for database applications as considered in this work. However, they often require the adoption of a particular programming model and/or system. Reimplementing their database application in a different programming language or porting it to a different system is unfeasible for scientific users that have often spend huge amounts of time for creating and fine-tuning their database application.

Combining Database and OS Provenance A less-intrusive alternative to unified provenance models are approaches that *combine* the two types of provenance, e.g., by linking nodes from separate OS and DB provenance graphs. This approach has the advantage that existing provenance systems can be used to track provenance as long as it is possible to connect the types of provenance. Cross-layer provenance was first described in [21] sighting that single-layer systems fail to account for the different levels of abstraction at which users need to reason about their data and processes. These systems cannot integrate data provenance across layers and cannot answer questions that require an integrated view of the provenance. These issues were explored in the context of workflow systems and for NFS servers. Combining provenance where one of the layers is a database is an active area of research. The combined model has been explored in the context of VisTrails [7] workflows using a centralized temporal database. However, this approach does not support fine-grained database provenance.

A full understanding of the trade-offs between these two modelling approaches is an outstanding research topic. In this work we use the linking model since it leverages currently more established and standardized models of OS and DB provenance, lends itself for integrating existing provenance systems, and requires no modification of the user's database applications. In contrast to alternative approaches, our model enables inference of data dependencies across the database and OS provenance models. Furthermore, we demonstrate how to apply this model to reduce package sizes by including only relevant parts of a database and for partial reproducibility.

IV. DB and OS Provenance Models

A. Provenance Model

To be able to record the provenance of database applications and repeat them, we need to connect provenance recorded on the OS side with provenance for the database. Furthermore, we want to infer dependencies between entities from the application system (e.g., files) and data items managed by the database system (tuples). Such merged dependencies enable us to reason about which data is required to reproduce a application or part of a application and to determine which parts of the database to include in a repeatability package. We assume that for both the DB and OS provenance models the following can be provided:

- A definition of a set of activity and entity types valid in their domain. For instance, the activities on the database side may be SQL statements.
- A set of inference rules for determining data dependencies that connect entities. For example, a file written by a process may depend on a file read by the process.
- The produced provenance can be represented in PROV.[3]

[3]We do not require that these systems can actually export provenance in PROV format, but only that it is possible to encode their provenance in PROV. Given the generality of PROV this should not be a limitation.

Furthermore, we require that the DB and OS provenance system's are capable of monitoring an execution to record direct provenance dependencies between entities and activities.

We first introduce *provenance models* and define *execution traces* that record direct provenance dependencies with temporal annotations. For example, a process p has read from a file f between time T_b and T_e or a query q has produced a result tuple t at time T. Afterwards, we introduce*combined execution traces* that link OS and DB provenance though activities that represent database queries and updates. We then study how to infer temporally-constrained data dependencies based on the direct dependencies of execution traces.

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, $\mathcal{E} \subseteq \mathbb{L}$ is a set of entity types, and $\mathcal{L}$ is a set of triples from $\mathbb{L} \times \mathcal{P}(\mathcal{A} \cup \mathcal{E}) \times \mathcal{P}(\mathcal{A} \cup \mathcal{E})$. These triples represent edge types with constraints on the start and end node types (sets of allowed types). We require that activity, entity, and edge labels are pairwise distinct.*

A provenance model defines the admissible types of activities and entities in a specific domain and determines how these types can be connected through edges of specific types.

B. Execution Traces

A provenance model can be used to model an execution of activities in the domain of the model. We call a record of such an execution an *execution trace*. An execution trace is a graph consisting of instances of the provenance model's activity and entity types (the nodes of the graph). Edges in an execution trace are labelled with a time interval indicating when the two nodes in the trace interacted with each other. For instance, such a label may represent the time interval during which a process (activity) was reading from a file (entity).

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 $G = (V, E, T)$, where V is a set of nodes (activities and entities). Each node has to be of one of the types specified in the provenance model. $E \subseteq V \times V$ is a set of edges that fulfill the type constraints specified by $\mathcal{L}$. Finally, $T : E \rightarrow \mathbb{T} \times \mathbb{T}$ 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.*

We now we introduce the provenance models that are used by the provenance systems we integrate in our LDV prototype.

C. The Blackbox Process OS Provenance Model

The blackbox process provenance model is used to model the provenance of OS processes and their interaction with files. As the name suggests, we do not assume any knowledge of about the inner workings of such processes. Processes are the activities of the provenance 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 $\mathbb{P}_{BB}$ activities, entities, and edge types:*

$$\begin{aligned}\mathcal{A} =& \{process\}\\ \mathcal{E} =& \{file\}\\ \mathcal{L} =& \{readFrom(file, process), hasWritten(process, file),\\ & executed(process, process)\}\end{aligned}$$

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

D. The Lineage DB Provenance Model

At the database side, activities are SQL statements and entities are tuples. We consider two types of relationships: an SQL statement reads a tuple, and an SQL statement produces a tuple - either a new tuple version produced by an update operation or a tuple returned by a query. We consider four types of SQL statements in this work: `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 and modifications (insert, updates, and deletes) are connected to original versions of tuples they have modified. In Section VI we will discuss dependencies between tuples based on a standard database provenance model.

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

$$\begin{aligned}\mathcal{A} &= \{query, insert, update, delete\}\\ \mathcal{E} &= \{tuple\}\\ \mathcal{L} &= \{hasReturned(\mathcal{A}, tuple), hasRead(tuple, \mathcal{A})\}\end{aligned}$$

We consider SQL statements to be executed instantaneously. That is, both incoming and outgoing edges of a statement executed at time T are labeled with the interval $[T, T]$.

Example 2. *Consider the execution trace shown on the bottom of Figure 2. $Insert_1$ has inserted two tuples t_1 and t_2 and $Insert_2$ has inserted tuple t_3. Tuples t_1 and t_3 where read by query $Query$ which has returned two result tuples t_4 and t_5.*

V. Combined Provenance Model

In order to support both application and database provenance, we combine an OS provenance model with a DB provenance model and augment these models with additional edges types that connect activities across model boundaries.

Definition 5 (Combined Provenance Model)**.** *Let $\mathbb{P}_{OS} = (\mathcal{A}_{OS}, \mathcal{E}_{OS}, \mathcal{L}_{OS})$ be an OS provenance model and $\mathbb{P}_{DB} =$*

$(\mathcal{A}_{DB}, \mathcal{E}_{DB}, \mathcal{L}_{DB})$ *be a database provenance model. The combined model* $\mathbb{P}_{DB+OS}$ *for* $\mathbb{P}_{OS}$ *and* $\mathbb{P}_{DB}$ *is defined as:*

$$\mathcal{A} = \mathcal{A}_{OS} \cup \mathcal{A}_{DB}$$
$$\mathcal{E} = \mathcal{E}_{OS} \cup \mathcal{E}_{DB}$$
$$\mathcal{L} = \mathcal{L}_{OS} \cup \mathcal{L}_{DB} \cup \{run(\mathcal{A}_{OS}, \mathcal{A}_{DB}), readFrom(\mathcal{E}_{DB}, \mathcal{A}_{OS})\}$$

A. Combined Execution Traces

A combined execution trace models the execution of a database application including its processes, file operations, and database 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. A combined execution trace for* $\mathbb{P}_{DB}$ *and* $\mathbb{P}_{OS}$ *is an execution trace for* $\mathbb{P}_{DB+OS}$*.*

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 *has read two files* A *and* B*, and has executed two insert statements (at time* 5 *and* 8 *respectively). These insert statements have created three tuple versions* t_1, t_2*, and* t_3*. Process* P_2 *has executed a query which has returned tuples* t_4 *and* t_5*. These tuples depend on tuples* t_1 *and* t_3*. Finally, process* P_2 *has written file* C*.*

Combined execution traces are created by monitoring system calls (e.g., ptrace) and intercepting calls to the database. This will be discussed in detail in Section VII.

VI. Data Dependencies

When producing a repeatability package, we need to know which database content and files need to shipped to guarantee successful re-execution of the trace or part of the trace. A successful repetition of a trace requires that the dependencies of all entities produced by the trace are available. Thus, we need to know on which entities an entity depends on. For example, we use this information to determine which database tuples need to be included in a package. We assume that an entity can be recreated by reexecuting part of the trace it directly or transitively depends on. Data dependencies, i.e., dependencies between entities, are specific to a provenance model. For example, in a fine-grained database provenance model we can exactly determine on which input tuples a query result tuple depends on whereas for black box processes we have to assume that an output depends on all inputs. Based on these model specific dependencies we will demonstrate in Section VI-C how to exclude spurious dependencies and how to infer indirect dependencies based on temporal constraints.

A. Lineage DB Dependencies

We use the Lineage provenance model for database queries to determine dependencies between input and output tuples of database operations in the $\mathbb{P}_{Lin}$ model. Lineage [6], [16] models the provenance of a query result tuple t as a set of tuples from the database instance that were used to derive t. Lineage can be expressed using the semirings annotation framework [16]. In the semiring framework, tuples are annotated with elements from a commutative semiring which represent their provenance. The semiring framework provides strong theoretical guarantees, e.g., it was proven to generalize several extensions of the relational model. Provenance polynomials are the most general form of provenance expressible in this framework. Lineage is less informative, but simpler and sufficient for our use case - determining dependency edges between tuples. Systems such as Perm [11] 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.

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

Example 4 (Lineage)**.** *Consider the sales table shown in Figure 5. The Lineage of each tuple* t *is shown as sets of tuple identifiers on the left of* t*. All tuples in the sales table are annotated with a singleton set containing the identifier of the tuples itself. If we run a query* `SELECT sum(value) AS ttl FROM sales WHERE price > 10`*, the result would be a single row with* $ttl = 11 + 14 = 25$*. The Lineage contains all tuples that were used to compute this results. In the example these are the tuples with ids* t_2 *and* t_3*.*

We will now define how to infer data dependency edges in the $\mathbb{P}_{Lin}$ provenance model based on Linage. In particular, 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 operation to the corresponding tuple t' in the input of the update.

Definition 7 ($\mathbb{P}_{Lin}$ Data Dependencies)**.** *Let* G *be an* $\mathbb{P}_{Lin}$ *trace. Let* $Lin(s, t)$ *denote the Lineage of tuple* t *in the result of database operation* s*, and* 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 execution trace shown in Figure 3 where* Q_1 *is the query from Figure 5. Recall that in the* $\mathbb{P}_{Lin}$ *model, operations are assumed to be instantaneous. Tuple* t_4 *is data dependent on* t_2 *and* t_3*, because these tuples are in the Lineage of* t_4 *according to query* Q_1*.*

B. Blackbox Process OS Dependencies

The binary applications we are tracking can keep arbitrary state and without static program analysis or dynamic instrumentation [26], [24] it is impossible to know which outputs are dependent on which inputs. Thus, a file f depends on another file f' if there exists a process that reads from file f' and writes file f. That is all outputs of a process depend on all inputs of that process. Furthermore, in the $\mathbb{P}_{BB}$ model a

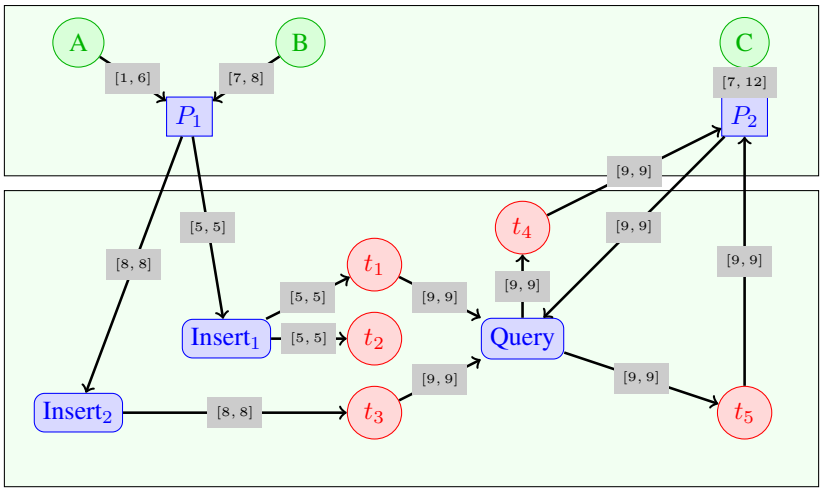

Fig. 2: An Execution Trace with Processes And Database Operations

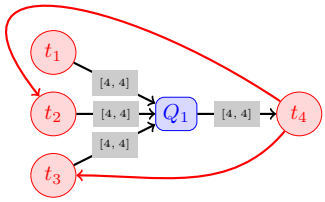

Fig. 3: $\mathbb{P}_{Lin}$ Trace and Data Dependencies.

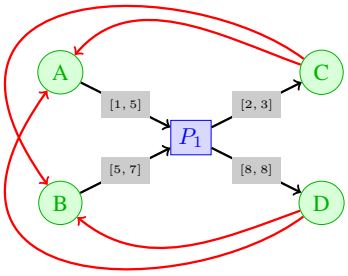

Fig. 4: $\mathbb{P}_{BB}$ Trace and Data Dependencies.

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 $\mathbb{P}_{BB}$ execution trace. Let f and f' denote entity (file) nodes in G and P_i be process nodes. The data dependencies $D(G) \subset D \times D$ of G with 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 where the first process reads file f' and the last process writes file f'. Furthermore, each process P_i was executed by process P_{i-1}.

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

C. Inferring Temporally Restricted Data Dependencies

In this section we 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. Inference should be *conservative*, i.e., it may return a superset of the real dependencies, but never a miss a dependency. Conservatism guarantees that sufficient data contained in repeatability packages to reproduce results. However, a high number of false positives would cause unnecessarily large repeatability packages. Our goal is to define inference rules that are *conservative* and reduce the number of false positives.

Our inference approach relies on three intuitive assumptions: 1) the execution trace records all direct interactions between activities and entities and 2) the state of an activity or entity at a time T only depends on past interactions and 3) the data dependencies defined by the individual provenance models are conservative (e.g., the dependencies of the $\mathbb{P}_{BB}$ model may contain spurious dependencies).

To infer such dependencies we need to understand which direct interactions (edges) in the execution trace influence the state of a node v at at time T. Based on the assumption 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 9 (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. Intuitively, 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 10 (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 $e_m = v_n = e$).

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 $T_1, \ldots, T_n$ with $T_i \leq T_{i+1}$ 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 1) 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 3rd 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 temoral 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 according to Definition 10 are sound and complete with respect to dependencies that fulfill the assumptions stated in the beginning of this section.*

Proof: Recall we were assuming that the following three assumptions hold for all dependencies in an execution trace:

- Execution traces record all direct interactions
- States of nodes at time t only depend on past interactions this node had before t
- Data dependencies of the individual provenance models are conservative

Let $D_{all}(G)$ denote the set of all dependencies between nodes in G that are conformant with the three assumptions we have stated. Furthermore, let $D^*(G)$ denote the set of dependencies inferred using Definition 10. 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.

$D_{all}(G) \subseteq D^*(G)$ (complete): 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 $(e, e') \in D^*(G)$. We prove this fact by contradiction. Assume that $e \notin D^*(G)$. We have to distinguish two cases:

Case 1: There is no path between e' and e in the trace. However, this would violate the assumptions, because we assume that individual model dependencies are conservative and the state of entities only depends on past interactions. Thus, if there is no interaction (edge) between two nodes v and v' then there cannot be any dependency.

Case 2: There exists one or more paths between e' and e. Then for every path between e' and e, one of the three conditions of Definition 10 has to fail, because else we would have $e \in D^*(G)$. The contradiction follows if we can construct at least one path for which conditions 1-3 of Definition 10 hold. From case 1 we know that there exist paths between e and e' in G. 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'. 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 9. Let $v_1 = e'', \ldots, v_n = e$ be this path. Based on assumption 2) we can infer that condition 3 of Definition 10 holds for this path. Based on the definition of state (Definition 9) it follows that condition 2 holds too. Finally, (e, e'') has to be a dependency in one of the individual provenance models based on assumption 3) which means condition 1) of Definition 10 holds. Thus, (e'', e) is a dependency in $D^*(G)$. Now the same argument can be applied to find an entity e''' on a path between e' and e'' and so on. By induction it follows that (e', e) in $D^*(G)$.

$D^*(G) \subseteq D_{all}(G)$ (sound): Let $(e, e') \in D^*(G)$. 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 1) e' and e are connected through a chain of conservative (assumption 3) data dependencies and temporal causality is not violated. ■

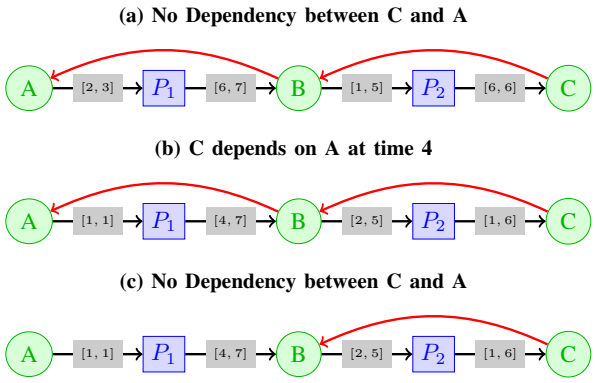

Fig. 6: Example Traces with Different Temporal Annotations

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

To create a re-executable package of a database application we need to transparently create execution traces. We describe how such traces can be created using the Unix ptrace utility on the application side. We then describe a ptrace-like utility that for creating execution traces on the database side.

A. Creating Execution Traces for the BB Process OS Model

The Unix *ptrace* system call provides a means by which one process (the "tracer") can observe and control the execution of another process (the "tracee") by examining all system calls executed by the tracee. For example, when a process accesses a file or a library using the system call *fopen()*, *ptrace* intercepts the tracee's *fopen()* system call. Similarly, if a process forks or execs another process then *ptrace* can intercept the *fork()* or *execve()* system call to examine the new called process and obtain its process id and other details from */proc*. PTU [22] is a system that uses *ptrace* to construct a provenance graph that connects process activities and file entities.

We can create an execution trace from a PTU provenance graph by attaching time stamp intervals to each edge. To create execution traces, we maintain a stack data structure for process-process and process-file edges. This data structure keeps track of file open and close system calls. A process-file edge is assigned the time interval between the first open system call and the last close system call. For process-process edges, the time interval is a point in time, assuming instantaneous fork or exec of the child process by the parent process.

B. Creating Execution Traces for DB Lineage Model

Dependencies between result and input tuples of a database operation can be computed using Perm [11].[4] However, a system like Perm computes provenance on-demand. Thus, we need a mechanism for tracking which SQL statements were executed by an application. This information can then be used to retrieve provenance for each such statement. As a proof of concept, we have chosen to create a trace by instrumenting *libpq*, the C language client interface of PostgreSQL (and Perm). By intercepting Select, Insert, Update, Delete statements send to the database via function *PQexec* in the *libpq* interface, we modify each statement (adding the `PROVENANCE` keyword supported by Perm) to compute its result tuples and also return all tuples on which the resulting tuples depend upon (its Lineage). In the future we will instrument ODBC as a more generic database client interface. Our interception layer records when each database statement was executed.

C. Combined Execution Traces

To create combined execution traces we need to connect traces for the $\mathbb{P}_{BB}$ and $\mathbb{P}_{Lin}$ models. In particular, we need to maintain edges between the processes and the SQL statements and between the tuples of a query result and a process, i.e., the edges $run(\mathcal{A}_{OS}, \mathcal{A}_{DB}), readFrom(\mathcal{E}_{DB}, \mathcal{A}_{OS})$. In general, this information can be maintained on the application or database side. We chose to store it at the database side alongside with tuples. To maintain these edges, we modify the schema of each relation in the database by adding additional attributes as described in Table I. This modification to a relation is done whenever the relation is first accessed by the database application. The values for the attributes are set by the *libpq* layer, which creates a unique identifier for the operation (`query_id`) and the id of the process that issued the query (`process_id`). The database provides the timestamp for the tuple when it is inserted. The unique row id acts a artificial primary key for tuple versions. By adding these attributes to every relation accessed by the application we keep track of which tuples where created by the application. When these inserted tuples are queried, then using Perm's provenance tracking mechanism, we can link the Lineage of these tuples to the application provenance. For Update and Delete operations, the system modifies the session id, updating them to the current process that executed the statements.

Name	Default	Description
prov_p	md5(random()::text)	unique query id
prov_usedby	0	session id
prov_v	now()	timestamp (tuple version)
prov_rowid	md5(random()::text)	unique row id for all tables

TABLE I: Attributes Added to Existing Relations

D. Creating a Virtualization Package

We first review how application virtualization tool create a package for non-database applications. We then describe the packing options for database applications supported by LDV.

In virtualization systems such as CDE or PTU a user can monitor any set of Linux commands by passing them as arguments to the virtualization system. The virtualization system executes the commands and uses ptrace to identify the code used by a running application (e.g., program binaries, libraries, scripts, data files, and environment variables), which it then records and combines to create a self-contained package. For example, when a process accesses a file or a dynamically-linked library using the system call *fopen()*, the system intercepts that system call, extracts the file path parameter from the call, and makes a "deep-copy" of the accessed file into a package directory and consisting of all sub-directories and symbolic links of the original file's location. The resulting package can be redistributed and run on another machine with the same architecture (e.g. x86). When reexecuting the package the virtualization system again uses ptrace to intercept system calls and redirects file-related calls into the package.

To build a virtualization package for a database application, we must include (or simulate) a database server, with which the application can interact with. Whether a server can be included (*Server-included* packaging option) or has to be simulated (*Server-excluded* packing option) depends upon on the level of control the database application has over the server. Control could be restricted due to permissions as well as due to licensing requirements. We consider three scenarios and describe packaging requirements under each scenario:

Scenario 1: Open-Source DB with Complete Access The application developer uses an open-source database server that is under complete control. The developer can start or stop the server, access its binaries, and access the database files. In this scenario, it is easy to package the server, since all the server's

[4]Technically, Perm only supports provenance for queries, but updates can be traced by rewriting them into queries to track their provenance [4].

binaries and any dynamically-linked library (e.g., user-defined functions) can be included in the package.

Scenario 2: Central DB with SQL Access The application uses a centralized database server. Such a server is usually being used by multiple users and is managed by central authority (IT department). The user can only access the server through its SQL language interface. Packaging the binary or source code of a central database server is not possible. An alternative is to package the non-database application, and provide the user of the shared package an account on the central server. However, either a new database would need to be created with pre-existing data or the original database must be brought to a state prior to database application execution. The latter functionality is available if the server itself supports versioning or time-travel feature.

Scenario 3: Proprietary DB with Complete Access The database server is proprietary. While the user may have control over starting and stopping the server, the server's binaries cannot be included in the package due to licensing constraints.

Using execution traces, we can create a "minimal" database in the original state for the first scenario. We need to include all tuples database that were not created by the application itself (no incoming edges in the trace) and were used by at least on SQL statement issued by the application (are the end-point of at least one data dependency). The application is packaged using *ptrace*, which can create the package and the execution trace for the black-box process OS model. The minimal database is stored in as plain files in the package.

Since it no possible to package the server and database in the 2^{nd} and 3^{rd} scenarios, we can intercept calls to the database and record query results returned from the server to be able to later simulate the database. For PostgreSQL (Perm) we can record all communication from the Perm server to *libpq* by tracing function `pqsecure_read(memory_buffer)` and dumping data from this buffer.

VIII. Repeating Executions with Packages

The reexecution of a LDV package depends on the type of package (*Server-included* or *Server-excluded*).

Server-included When the application first connects to the database during reexecution, then LDV intercepts this call and restores the database following these steps: 1) Extract the database name *dbname* from the given connection string; 2) Connect to the default database (*postgres*) and create a new database *dbname*; 3) Connect to *dbname* and restore the database content from the LDV package; 4) Return the connection to *dbname* in *PQconnectdb*. Once the database is restored to its original state all other database commutation procedes without interference from LDV. Since all tuples required to answer the applications queries were recorded by LDV's monitoring step and restored before any query occurs, the database application can be repeated successfully.

Server-excluded: In this scenario the package contains the results of all queries issued during the original execution. results server and LDV simulates the existence of the server and replays the recorded query results based on their time order. In this case, LDV simulates all database communication made via *libpq* to re-play the database connection and queries. A database connection is simulated by changing the behavior of functions in *libpq* that connect, read, and write to database servers. The following steps are taken to simulate the server: (i) When the application connects to the server, LDV interrupts functions *PQconnectPoll* and *pqWaitTimed* in *libpq* to bypass the process of creating a database connection. The simulated database connection returns a success status without connecting to any server; (ii) On a write request to a database server, LDV intercepts function *pqsecure_write* to ignore the request and always returns success; (iii) On a read request to a database server, LDV intercepts function *pqsecure_read(memory_buffer)*, writes recorded data to *memory_buffer* and returns the corresponding result. We assume that the client issues the same queries to the database server in the same time order as in the capturing step. By returning the recorded data in the same order as before, LDV replays the communication between a client and a server and allows the client to execute successfully without an actual server.

IX. Experiments and Evaluation

In this section, we describe our experiments to analyze LDV audit and replay performance, and compare package size for databases applications. We also evaluate LDV on provenance querying by building dependency graphs for the experiments. Our experiments show a trade-off in LDV performance and package sizes among our scenarios. While LDV spends more time on capturing LDV packages and initializing the re-execution, LDV packages are 28% - 84% smaller in size than a full capture of the application. Actual runtime of the application re-execution is about the same for the Open-Source DB Server scenario, and significantly faster, with 1000%-10,000% improvement for the Proprietary DB Server scenario for queries with small result sizes.

A. TPC-H Benchmark

We used the TPC-H benchmark [25] with a scale of 0.01 to generate data for our evaluation experiments. TPC-H is a decision support benchmark with a suite of business oriented ad-hoc queries and data modifications. The experiments contain an application and a PostgreSQL database of approximately 86,000 tuples. The application runs the following steps sequentially.

- Test database connection (and prepare database if needed in audit and re-execution mode)
- Insert 1000 tuples into tables *orders* and *lineitem* as specified by TPC-H SF1
- Send 101 queries to the database using one of the 22 TPC-H queries.
- Update 100 tuples in table *orders*

. To create a based line for comparison, we measured the execution time of the application using an uninstrumented PostgreSQL server. We used PTU to audit the application dependencies and create a PTU portable software package without database provenance. This PTU package contains

	Name	Scanned / Result	Category
Q1	Pricing Summary	57982 / 4	large input / small output
Q2	Minimum Cost Supplier	10100 / 8000	small input / large output
Q3	Shipping Priority	58 / 10	small input / small output

TABLE II: Numbers of scanned and returned tuples in TPC-H queries used for performance and size evaluation

all binaries, libraries and data required to re-execute the application.

We measure the LDV package size, audit performance, and replay performance for the first three queries of TPC-H. The first query in TPC-H is the Pricing Summary Report Query (Q1). This query reports the amount of business that was billed, shipped, and returned as of a given date. The query scans approximately 96% of tuples in table *lineitem*, or 66% of the database, and return 4 tuples (large input / small output). The second query is the Minimum Cost Supplier Query (Q2). This query determines which suppliers should be selected to place an order for a given part in a given region. We relaxed the query by removing some restricted conditions in the `WHERE` clause to include a higher number of tuples in its result (small input / large output). The third query is the Shipping Priority Query (Q3). This query retrieves the 10 unshipped orders with the highest potential revenue (small input / small output). Actual numbers of tuples scanned and returned in these queries are shown in Table II.

1) Audit Performance: We use LDV to audit the application to create an LDV software package. Figure 7 shows the LDV audit performance for the Open-Source DBS and the Proprietary DBS scenarios in comparison with the normal uninstrumented database server.

a) Scenario 1: Open-Source DB Server (DBS): In all audit phases, the preparation step and the select query step of the application show a significant overhead comparing to the normal uninstrumented execution. The overhead of the preparation step is accounted by the modification to database tables. In this scenario, LDV needs to add columns to any accessed tables and fill in their default unique values. For large tables such as *lineitem*, this process adds a significant overhead. The overhead of the select query step is resulted from querying for provenance and capturing unrecorded tuples. In the first select query of Q1, LDV needs to record almost 58000 tuples (67% of the database). Additional select queries do not need to record those tuples again, but they still need to make provenance queries for those tuples. Similar provenance queries are required for database update operations, which results in the overhead from the update query step. These provenance queries are not needed in insert queries; hence the overhead is light and lower than other steps.

b) Scenario 3: Proprietary DB Server: This scenario shows a lower overhead than the Open-Source DBS scenario. An exception is in auditing Q2 experiment where the returning result of the select query Q2 is large, and recording such result repeatedly introduce higher overhead in comparison with Open-Source DBS and uninstrumented scenarios.

Package	Software binaries	Server binaries	Data directory	Database provenance
PTU	✓	✓	✓(full)	✗
Open-Source DBS	✓	✓	✓(empty)	✓
Proprietary DBS	✓	✗	✗	✓

TABLE III: Content of PTU and LDV packages: PTU packages contain data directory of the full database, whereas Open-Source DBS LDV packages contain a data directory of an empty database (created by the *initdb* command)

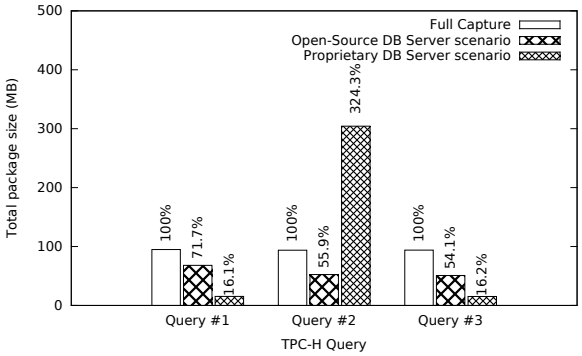

Fig. 9: Sizes of packages compared with a full capture

2) Performance in Re-Execution: Figure 8 compares the replay performance of LDV software packages in Open-Source DBS and Proprietary DBS scenarios and a normal execution. Open-Source DBS scenario exhibits a long database preparation since LDV needs to restore database state using audited provenance. Once the database is restored, performance of Open-Source DBS scenario is the same as in the normal uninstrumented execution. It is noticeable that in almost all experiment steps, Proprietary DBS scenario shows a lower execution time than Open-Source DBS scenario and normal execution. This can be explained as Proprietary DBS application does not wait for database server to process queries, but read their results directly from local disks. An exception appears in the Proprietary DBS scenario with select queries for the query Q2 where large-size results were returned. In normal and Open-Source DBS scenarios, the cache from database server returned results faster than repeatedly reading results from disks in Proprietary DBS scenario.

3) Package Size: To explore the improvement of LDV packages over repeatable software packages that contain full database, we compare the sizes of LDV and PTU packages to show a remarkably smaller size packages of LDV approach. A PTU package contains all the necessary binaries, libraries and files required to re-execute the application. This package contains all files accessed by the database server in the application execution. An LDV package contains database provenance for re-execution, and the database server binaries and a data directory of an empty database in the Open-Source DBS scenario (Table III).

Using the application with TPC-H query Q1, Q2, and Q3, we constructed their corresponding PTU and Open-Source DBS, and Proprietary DBS LDV packages. Figure 9 shows

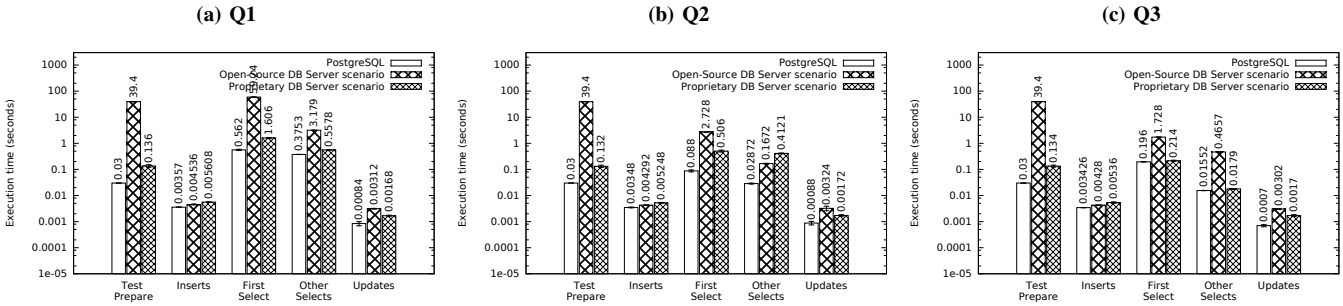

Fig. 7: Execution time of each step in an execution of TPC-H benchmark application in **Audit** mode

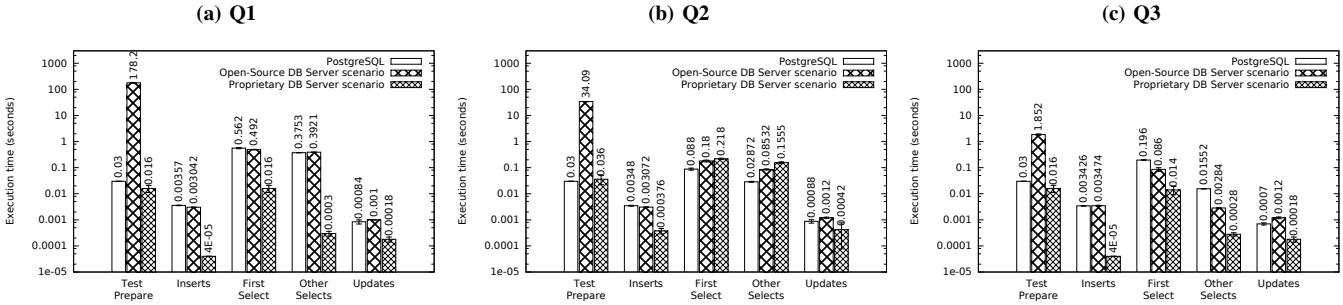

Fig. 8: Re-Execution time of each step in an execution of TPC-H benchmark application in **Replay** mode

Open-Source DBS LDV packages are smaller than PTU packages since Open-Source DBS LDV packages does not contain redundant tuples that PTU packages have. For TPC-H queries Q1 and Q3, since the number of output tuples are significantly smaller than the number of input tuples (Table II), the sizes of Proprietary DBS LDV packages are usually smaller than in PTU and Open-Source DBS LDV packages. However, in Q2, the number of output tuples are high and recorded repeatedly in the capture of our application. Hence, Proprietary DBS LDV package size in Q2 is considerably larger than PTU and Open-Source DBS LDV packages. Figure 10 shows a similar observation: Q1 requires a high number of tuples from the orginal database to be recorded, whereas Q10, Q11 and Q14 show a high number of output tuples being repeatedly recorded for re-execution.

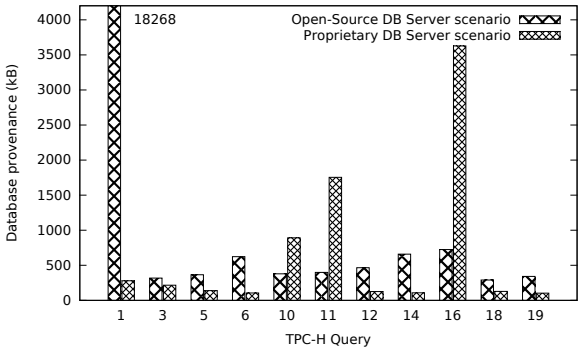

Fig. 10: Sizes of audited data captured for provenance query and re-execution

B. Provenance Query

LDV provides a query interface for returning a graph representation of the a whole or part of an execution trace. This interface creates a GraphViz-format provenance graph for the experiment in 0.4 ± 0.05 seconds. Using this provenance graph, users can examine dependencies along a timeline. Including inferred dependencies is optional.

X. Conclusions

We introduced a light-weight database virtualization (LDV) system that monitors applications involving a database. This system creates reexecutable packages including the application, its dependencies, data files, the relevant part of the database, and a combined execution trace (DB and OS provenance). Such packages can be used to repeat an application or part of an application in a different environment. We have presented a framework for combining different provenance models and for inferring data dependencies that cross model boundaries. Our system creates execution traces (provenance graphs) according to this framework and uses this type of provenance to determine which data needs to be included in a repeatability package. LDV leaves the choice whether the repeatability package should include the database server to the user. Our first prototype implementation integrates the PTU (OS) and Perm (database) provenance systems and instruments the Perm client interface to monitor SQL statements.

In future work, we plan to instrument client interfaces of other database systems. Furthermore, we will develop a third packaging option that uses a temporal database to retrieve past state of the database needed when reproducing a result and integrate our approach with the database-independent GProM [4] provenance middleware.

REFERENCES

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

[2] U. Acar, P. Buneman, et al. A graph model of data and workflow provenance. In *TaPP '10*, 2010.

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

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

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

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

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

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

[9] D. Deutch et al. A provenance framework for data-dependent process analysis. *PVLDB*, 7(6), 2014.

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

[11] B. Glavic et al. Perm: Processing Provenance and Data on the same Data Model through Query Rewriting. In *ICDE '09*.

[12] 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.

[13] 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.

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

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

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

[17] K. Keahey et al. Virtual workspaces for scientific applications. In *Journal of Physics: Conference Series*, volume 78, 2007.

[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.