Sharing and Reproducing Database Applications

Quan Pham[1], Severin Thaler[2], Tanu Malik[1,2], Ian Foster[1,2]
[1]Computation Institute, [2]Department of Computer Science
University of Chicago

Boris Glavic
Department of Computer Science
Illinois Institute of Technology

1. INTRODUCTION

Sharing and repeating scientific 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. The predominant methods of sharing and making applications repeatable are building a companion web site and/or provisioning a virtual machine image (VMI). Recently, *application virtualization* (AV), has emerged as a light-weight alternative for sharing and efficient repeatability. AV approaches such as Linux Containers create a chroot-like environment [4], while approaches such as CDE [1] trace system calls during application execution to copy all binaries, data, and software dependencies into a self-contained package.

In principle, application virtualization techniques can also be applied to *DB applications*, i.e., applications that interact with a relational database. However, these techniques treat a database system as a black-box application process and are thus oblivious to the query statements or database model supported by the database system. To overcome this shortcoming, and leverage database semantics, we recently introduced *light-weight database virtualization* (LDV)[1] [3], a tool for creating packages of DB applications. An LDV package is light-weight as it encapsulates only the application and its necessary and relevant dependencies (input files, binaries, and libraries) as well as only the necessary and relevant data from the database with which the application interacted with. LDV relies on data provenance to determine which database tuples and input files are relevant. While monitoring an application to create a package we incrementally construct an *execution trace* (provenance graph), that records dependencies across OS and DB boundaries. In addition to providing a detailed record of how files and tuples have been produced by the application, we use it to determine what should be included in the package.

The primary objective of this demonstration is to show the benefits of using LDV for repeating and understanding DB applications. For this, we consider real-world data sharing scenarios that involve a database and highlight the sharing and reproducibility challenges associated with them. We given an overview of our LDV approach to show how it can be used to build a light-weight package of a DB application that can be easily shared and reproduced. During the demonstration, the audience will experience three key features of LDV: (i) its ability to create self-contained packages of a DB application that can be shared and run on different machine configurations without the need to install a database system and setup a database, (ii) how LDV extracts a slice of the database accessed by an application, (iii) how LDV's execution traces can be used to understand how the files, processes, SQL operations, and database content of an application are related to each other.

[1]`https://github.com/legendOfZelda/LDV.git`

Proceedings of the VLDB Endowment, Vol. 8, No. 12
Copyright 2015 VLDB Endowment 2150-8097/15/08.

2. SHARING DB APPLICATIONS

We introduce several realistic DB applications to illustrate their individual repeatability requirements and challenges. In each scenario a scientist Alice would like to share her application with another scientist Bob, who would like to either re-execute it or verify the execution.

APP1: Sharing a DB application using a commercial database. Alice is an urban scientist who has published a paper about how housing affects school enrollment. Bob has read the paper and would like to repeat her analysis. The results in her paper use data that were downloaded from open-data services, such as `Opendata.gov`. Alice has imported the downloaded data into her commercial database management system (DBMS), which required additional cleaning and integration steps. Alice can share her workflow and provide a dump of the data to Bob, but not her database system as the license is not owned by Bob. Loading the data into Bob's open source DBMS will require a significant amount of work. Alice can share the queries (time periods and spatial extents) which she used to obtain the data from `Opendata.gov`. However, this service is open to the public and, thus, its database has been modified in the mean time. Hence, Alice's queries will not return the same result when run by Bob (assuming the service does not provide access to previous versions, which is typical).

R1: Packaging DB Applications with Data. *Significant communication between Alice and Bob as well as setup work on Bob's side is required to repeat the application. It would be helpful if Alice can sandbox* [2] *her application and*

[2]In the context of software development, sandbox implies creating an isolated, restricted environment that is a replica

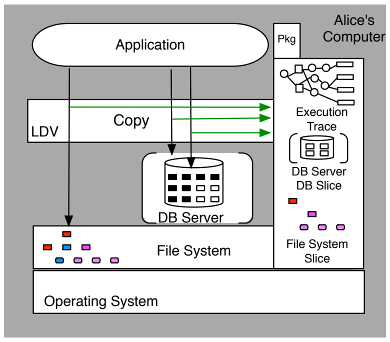


Figure 1: LDV Monitoring an Application

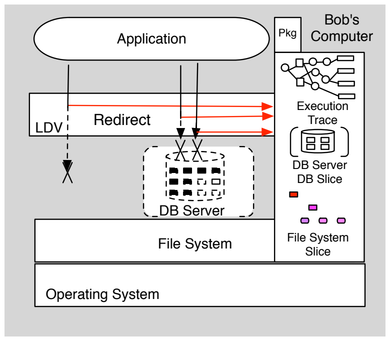


Figure 2: LDV Reexecuting an Application

her commercial database so that Bob can replay her workflow.

APP2: Sharing subsets of data from a large database. Bob has read Alice's research paper which simulates a sky catalog that is then verified with real observation data from the Sloan Digital Sky Survey (SDSS), a 18TB read-only database instance. Bob wants to use Alice's code and data. Getting access to Alice's code is easy, but downloading the entire SDSS database instance is not, because Bob is not a power user and Alice's application data exceeds his download quota.

R2: Share Relevant DB Slices. *Bob's repeatability experiment will be simplified, if Alice shares with him the spatial regions from SDSS she has used. Alice would have to recollect which subsets were used. It would be helpful if the relevant data tuples for Bob could be determined automatically.*

APP3: Tracking Provenance Across Files and DBMSs. We consider a text mining DB application in which Alice extracts information about scientific parameters from research publications and curates them using a DBMS. The research publications are downloaded from multiple hosted data sources such as PubMed, ChemXSeer, and BioOne. The extraction pipeline consists of cleaning and natural language processing tasks. The database is mainly used to store the extracted entries. Alice would like to share her entire application with Bob, a scientific expert, who can validate her database entries.

R3: Tracking and Visualizing Provenance. *To validate, Bob would like to link Alice's database entries to raw input text and temporary intermediate data files created by the application. Bob also would like to know which extraction rules were applied and how. Alice would have to manually maintain a graph of provenance dependencies for all input files and each database tuple. This would be a nightmare for any reasonably complex extraction pipeline.*

3. CHALLENGES AND REQUIREMENTS

In our use cases, Alice would need to provide sufficient documentation to Bob so he can understand the application well enough to repeat it and spend extensive manual effort to determine which parts of the input database are needed. In the worst case she may not be able to share her application with Bob. We now outline the main challenges that need to be addressed to fulfill the sharing requirements of applications such as the ones introduced above.

Packaging Databases. A database system may exist as a (i) personal database systems (ii) multi-user updateable database system, or as a (iii) read-only database system. In (i) the application developer has complete control over the data and the database server; the developer can start or stop the server, access its binaries, and access the database files. In (ii) the server is managed by a central authority, but users are allowed to run both updates and queries. In (iii) a central authority manages the server and all modification of the database; the users have only read access which may be limited (e.g., a form-based interface). Packaging a database system encompasses multiple challenges: we want to repeat an application without having to manually install and setup a database server; we have to recreate the database system state as seen by the operations of the application even if the application or other clients have updated the data meanwhile; and, we have to determine which data are needed for reexecution.

Licensing policies. A deployed database system may be an open-source database system or a commercial database system. For commercial database systems, Bob may not possess the required license.

User access policies. A database system may enforce access privileges limiting which relations can be accessed by a user. This is a problem if Bob does not have the necessary rights to execute the SQL statements issued by the application.

Tracking File and Database Provenance. To provide a complete record of how data was generated by the application we have to track provenance on the DBMS side, the OS side (processes and files), and across both.

We next explain how LDV manages to satisfy these requirements. In the demonstration, participants will be able to see LDV in action and will be able to get a peak behind the scenes to understand the inner workings of our approach.

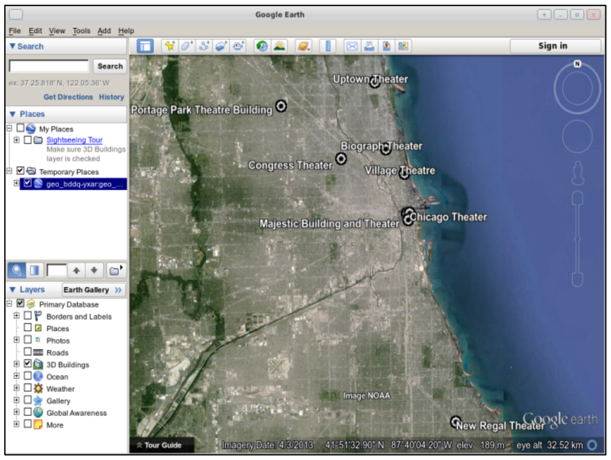
4. THE PROTOTYPE LDV SYSTEM

Monitoring and Package Creation. To use our LDV system, the user monitors an execution by prefixing the command starting the application with `ldv-audit`:

```
ldv-audit run-dbapp
```

Figure 1 shows how LDV monitors the execution of the user's application and its interactions with the OS and DB system. By intercepting system calls such as file operations and process creation as well as SQL statements send to the

of the original environment, and which allows testing of programs



```
Alice:~$ ldv-audit app.sh
Application package created as app-pkg
Alice:~$ ls
app-pkg    app.sh    src    data
Alice:~$echo "Hi Bob, Please find the pkg --Alice" \ |
mutt -s "Sharing DB Application -a "./app-pkg"  \
-- bob-vldb2015@gmail.com
```

```
Bob:~$ ls .
app-pkg
Bob:~$ cd app-pkg
Bob:~$ ls
app.sh     src     data
Bob:~$ldv-exec app.sh
Running app-pkg....
```

Figure 3: Creating a package for Alice's application (left) and re-executing it on Bob's Machine (right) recreates the same image

DB we incrementally build what we call an execution trace - a provenance graph that records both OS and DB operations and data dependencies. In addition to creating the execution trace and including it in the reproducibility package for the application, LDV also copies accessed files and database tuples into the package. We support two options for shipping the database. The *server-include* packaging option includes a DB server and the relevant DB slice in the package. The *server-excluded* packaging option captures the results of queries issued by the application and stores these query results in the package. By providing these two packaging options we support the types of DB deployments, licensing options, and access policies introduced in Section 3.

The operating system part of an execution trace is created by using the *ptrace* system utility on Linux systems, and by using *dyld* on Mac OS X. We use both utilities to monitor system calls including process system calls and filesystem system calls to create execution trace nodes that connect process *activities* and file *entities*. LDV records a time interval for each process-process and process-file interaction and attaches it to the respective edge in the execution trace.

The database part of an execution trace is created by using the Perm provenance system to compute the provenance of SQL operations, i.e., to track the input tuples on which an output tuple of the operation depends. For each executed SQL statement, the system creates a node in the execution trace which is then linked to nodes for all of its result tuples and the node for the process issuing the statement. Our prototype implementation uses an instrumented version of `libpq`, the C language client interface of PostgreSQL (Perm is a modified PostgreSQL server). We intercept Select, Insert, Update, and Delete statements send to the DB and modify each statement to compute its result tuples and return all tuples on which the result tuples depend upon (this is how we preserve previous tuple versions for updates).

Based on the execution trace of an application we determine which tuples need to be included in the package to guarantee successful reexecution. These are all input tuples that are in the provenance of an output of an SQL operation except for tuples created by the application itself (re-created during reexecution). Thus, we fulfill the requirement of packaging only the relevant database slice.

LDV stores execution traces in LevelDB, a highly optimized key-value store. We offer the user the option to deposit execution traces into PROVaaS (`http://provaas.org`) by exporting the LevelDB data into PROV JSON format (a serialization of the PROV [2] provenance exchange format) and depositing it to the service. This service implements the PROV standard using a Neo4J database and supports a viewer for querying and exploring provenance data. The execution traces we generate fulfill the provenance tracking and querying requirements outlined in Section 3.

Reexecuting a Package. To replay the execution of an application stored in a package, without any installation or configuration, the user issues for a shared package:

```
ldv-exec run-dbapp
```

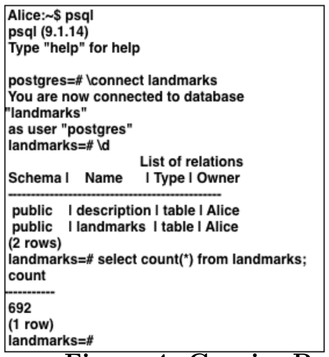
Before starting the actual application, LDV will startup the database server included in the package, create the schema of the application, and load the DB slice. During application execution we reroute SQL queries to the package database and file operations into the package. If the server-excluded packaging option was chosen then we replay query results included in the package from files instead of actually executing any SQL operations.

5. DEMONSTRATION OVERVIEW

In the demonstrations we will use applications similar to the ones we have described above to showcase the following features of the the LDV system: (i) reproducing a database applications on different machine configurations; (ii) rerun a DB application without using a database; (iii) creating packages of reduced size by extracting the relevant part of an accessed database needed for re-execution; (iv) navigate execution traces (provenance) to understand the dependencies between data items created and used by an applications.

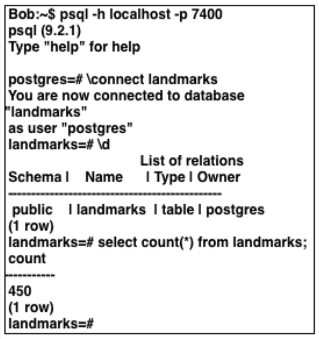
Sharing and Reexecution. We will demonstrate how to repeat DB applications with LDV on a wide variety of linux distributions. Alice's machine (Ubuntu 14.10, Linux 3.13) will be used to run her DB application that reads theater landmarks from a landmarks database of the City of Chicago. The landmarks (output as KML files) will be displayed visually using Google Earth. We will use Alice's machine to monitor a DB application, then copy the generated package to Bob's machine, and demonstrate that it can be successfully re-executed on Bob's machine. The GUI (Figure 3) will show the configuration of both machines, a visualization of the application's results for both machines, and a terminal for starting the LDV audit and replay executions. We will use different configurations for Bob's machine to demonstrate successful re-execution. This will include (i) CentOS 6.2 (Linux 2.6.32), (ii) Fedora 20 (Linux 3.11.28), (iii) openSUSE 12.0 (Linux 3.12.32), (iv) Ubuntu 14.04 (Linux 3.13), (v) Red Hat 7.1 (Linux 3.10.0). We will pre-install these configurations with different database servers *viz.* Postgres *v*9.0, and *v*9.1, and MySQL. It is to be noted that Google Earth does not install cleanly on all Linux configurations, but within our package it re-executes smoothly.

Reexecution Without a Database. LDV's execution traces can also be used to re-execute an application when



```
Alice:~$ psql
psql (9.1.14)
Type "help" for help

postgres=# \connect landmarks
You are now connected to database
"landmarks"
as user "postgres"
landmarks=# \d
                    List of relations
Schema |    Name     | Type | Owner
-----------------------------------------------
 public    | description | table | Alice
 public    | landmarks  | table | Alice
(2 rows)
landmarks=# select count(*) from landmarks;
count
----------
692
(1 row)
landmarks=#
```



```
Bob:~$ psql -h localhost -p 7400
psql (9.2.1)
Type "help" for help

postgres=# \connect landmarks
You are now connected to database
"landmarks"
as user "postgres"
landmarks=# \d
                    List of relations
Schema |    Name     | Type | Owner
-----------------------------------------------
 public    | landmarks  | table | postgres
(1 row)
landmarks=# select count(*) from landmarks;
count
----------
450
(1 row)
landmarks=#
```

Figure 4: Concise Repeatability Packages

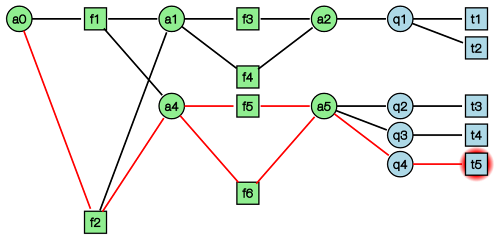


Figure 5: Execution trace showing the dependencies between database tuples and files containing source data

sharing the database is not an option (e.g., as explained before Alice may use a commercial DBMS and Bob does not have a license). In this case LDV will store the results of queries in the package and replay these results during re-execution. For this demonstration we will simulate a session in which Alice executes three queries on the Google Earth application which shows a) landmarks in Chicago, b) landmarks since 1950, and c) un-preserved landmarks. We will show how the three queries can be replayed on Bob's machine, which does not have a copy of the database, showing the exact same results in the same chronological order. The simulation will add natural human delay between queries, which will be maintained at Bob's machine as well.

Concise Repeatability Packages. We then showcase LDVs capability to determine which parts of a database are needed to reproduce an DB application and that this can significantly reduce package size. Alice is executing a variation of the Google Earth application described above that only visualizes a subset of the data. We will demonstrate that the generated LDV package shipped to Bob is significantly smaller than the original database. Furthermore, using a standard database client we will show the differences between the schemas and data of the original and packaged database. In particular, the packaged database only contains relations that are accessed by the application and only contains tuples needed to produce the same results for queries as in the original execution. We illustrate this feature by re-executing some of the queries used by Alice's application and comparing their results to the original run.

Navigating Provenance Execution Traces. Next, we will demonstrate how combined execution traces help a user to understand how data items (files or tuples in a database) have been produced and their interrelationships. We will use a customized graph visualizer that will render the execution trace of application *APP3*. The objective will be to show how Bob can understand Alice's text extraction process, and conduct his curation task of tracing, i.e., which source data has lead to invalid or inaccurate database entries and correct these errors. Recall that this application extracts properties from text files and stores them in a database. Using the visualizer, Bob can select elements in the execution trace to get additional information (e.g., values of tuples or inspect file contents). Our system also highlights the provenance (data dependencies) for selected elements. Figure 5 shows a screenshot of the dependencies for an invalid database tuple t_5 produced by Alice's application. For instance, this helps Bob to understand that the erroneous tuple t_5 depends on data stored in file f_2 (connected through highlighted edges with t_5), but not on, e.g., file f_1. Using our viewer, Bob can then inspect the content of file f_1 (or another intermediate result file in t_5's provenance). Note that if two elements are connected in the execution trace this does not necessarily imply that they depend on each other. For instance, a process P may have written a file B and afterwards read from a file A. Files A and B will be connected through P, but B's content obviously does not depend on A. LDV uses temporal information like this which is recorded in the execution trace and fine-grained database provenance to exclude spurious data dependencies.

Finally, if Bob sees incorrect tuples in the database, he may wonder which version of Alice's programs are producing the incorrect data. Therefore he might be interested in determining the version of the program that was used to create this particular version of the tuple. If a Github-like version control system is available on the OS side and/or transaction time support is available on the database side, then our visualizer takes this version information into account and shows versions of files and tuples and their relationships.

6. CONCLUSIONS

In this demonstration we introduce the attendants to the challenges of sharing DB applications, present several types of use cases for sharing such applications, and show how light-weight virtualization with provenance can be used to address the challenges faced by these use cases. Furthermore, attendants will be able to get a look behind the curtains and see how our LDV system realizes repeatability.

7. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grants ICER-1440327, ICER-1343816 and SES-0951576, and by the US Department of Energy under contract DE-AC02-06CH11357.

8. REFERENCES

[1] P. J. Guo et al. CDE: Using System Call Interposition to Automatically Create Portable Software Packages. In *USENIX*, 2011.

[2] L. Moreau and P. Missier. PROV-DM: The PROV Data Model. *http://www.w3.org/TR/2013/REC-prov-dm-20130430/*, 2013.

[3] Q. Pham, T. Malik, B. Glavic, and I. Foster. LDV: Light-weight Database Virtualization. In *ICDE*, pages 1179–1190, 2015.

[4] M. Riondato. Jails. `https://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/jails.html`, 2009.