

Reproducible Preservation of Databases through Executable Specifications

Ivar Rummelhoff, Thor Kristoffersen, Bjarte M. Østvold
Norwegian Computing Center

Abstract

We propose a new preservation method for relational data and a corresponding tool. The method involves writing a specification that can later be executed by the tool without user interaction, transforming the input files and databases into an encapsulated package suitable for archiving. Thus, the transformation steps become reproducible, which facilitates automation by reusing the specifications and allows for an iterative process, where for each iteration the specification is extended or adjusted and then executed to check that the result is closer to fulfilling future access requirements.

Submitted 8 February 2024 ~ Accepted 27 May 2025

Correspondence should be addressed to Ivar Rummelhoff. Email: ivar.rummelhoff@nr.no

The *International Journal of Digital Curation* is an international journal committed to scholarly excellence and dedicated to the advancement of digital curation across a wide range of sectors. The IJDC is published by the University of Edinburgh on behalf of the Digital Curation Centre. ISSN: 1746-8256. URL: <http://www.ijdc.net/>

Copyright rests with the authors. This work is released under a Creative Commons Attribution License, version 4.0. For details please see <https://creativecommons.org/licenses/by/4.0/>



Introduction

Much of the world's information is stored in relational databases. Our work is concerned with the preservation of the digital records in such databases in a way that ensures both consistent quality and easy access to the information. More precisely, we focus on how to transform such data into digital artifacts suitable for archiving—possibly tailored to the expected future access needs. When such steps are carried out manually, it can lead to unpredictable data quality and put future data access at risk. To address this, we contribute a novel method for digital preservation of databases with the following components:

- An *executable specification language*, *DbSpec*, for describing the extraction, migration, transformation and validation of relational data and metadata from databases and other sources.
- An *interpreter* for the specification language, i.e., a computer programme that can automatically and deterministically execute the steps in a specification.

These contributions fill a gap in the existing tool landscape for database preservation in that they make explicit and reproducible the steps for producing database archives and encapsulated packages, including integration with other tools and scripts.

The DbSpec interpreter is free and open-source software (FOSS) available at <https://github.com/immortalvm/dbspec>. This repository also contains a language reference and some examples.

Background and Motivation

This work is motivated by the following question: Is there a method for extracting and transforming relational data for future access that is *efficient*, *reliable*, *transparent* and *reproducible*? The method should be useful even in situations where the available time and resources are scarce. This makes efficiency a priority, but also reliability, as unreliable tools can be costly. Transparency and reproducibility are important for provenance and trust, much as in science (Munafò et al., 2017). These qualities should also be reflected in the preserved metadata, since knowing *how* the data has been produced is often necessary to be sure what it means. This can also lead to savings, because the preservation steps can be more easily reused.

Scope

Relational databases are used for many purposes; and sometimes the data we want to preserve is relational in nature even though it does not currently reside in a relational database management system (RDBMS). Conversely, such systems also support non-relational data such as XML or JSON. Functionality for managing these data types has even been included in the SQL standard. In addition, databases often contain data in custom and/or proprietary formats, whether as “large objects” stored within the RDBMS or as references to external files.

We are concerned with the extraction, migration, transformation and validation of relational data for future access. For the most part, this is what we shall mean by the term *preservation*. In the DCC Curation Lifecycle Model (Higgins, 2008), such steps would sort under Ingest, Preservation and Transformation actions. We shall not make this distinction, but whereas in the lifecycle model a transformation usually describes an action to produce new data after accessing the preserved data, we are only interested in the transformations happening before or as part of the preservation process.

Database preservation projects can be very complex and require both cross-disciplinary and cross-organisational collaboration. In other cases, it is just the IT department being asked to produce a simplified extract of some data expected to have future value. This might happen when an IT system is about to be decommissioned, in which case the preservation effort will be a one-off event. But even in such scenarios, it is valuable to be able to rerun the preservation steps without human involvement.

We make no assumptions about the nature of the data or the reasons for preservation. Therefore, it is mostly outside our scope how the resulting digital preservation artifacts should be managed. For instance, public records should perhaps be deposited in a certified repository (CRL & OCLC, 2007), and research data might be uploaded to a service such as Zenodo.¹ In such cases, it makes sense to include the submission of data and metadata in the list of preservation steps. This is especially true if it involves specifying additional metadata. However, in order to automate such steps, the process must be scriptable—for example, using a public API.

Data Extraction, Formats and Standards

The crudest and cheapest way of preserving a database is to simply store a disk image or a database backup;² and even when a database has been preserved by other means, it is common to retain the original, raw data as a precaution. However, it is widely recognised that it can be difficult to interpret such data in the future. In order to preserve the *information*, complex extraction steps are sometimes needed that require a deep understanding of the originating system, the database schema or specific technologies.

For common formats, the threat of “digital obsolescence” appears to be low (Rosenthal, 2010), especially for those that have open-source implementations. Even the proprietary backup formats of major commercial RDBMSs will presumably be accessible for the foreseeable future, but there might be high licensing fees. Thus, the preservation steps are mainly there to (1) ensure that we preserve the right data and metadata and (2) lower the cost of future use. Traditionally, three approaches to the preservation of relational data have been considered: migration, emulation and XML (Digital Preservation Testbed, 2003). In our work, we have concentrated on preservation using XML—which we believe makes the most sense for long-term preservation—and on the following standards:

Software Independent Archiving of Relational Databases (SIARD) is an open format designed for archiving relational databases in a vendor-neutral form (DILCIS Board, 2021c). A SIARD file is a ZIP64 file containing data, data types, table structure, relations, and views stemming from an SQL:2008-compliant database and serialised as XML. SIARD 2.2 was approved in 2021. This is the latest version and the one we have been using. In a conventional archiving scenario, the SIARD file along with a description of the data content and metadata will be enclosed in a *Submission Information Package* (SIP) and submitted to a repository. Information packages were defined in the somewhat abstract *Open Archival Information System* (OAIS) Reference Model. A standardised file format has been proposed by the E-ARK project and the DILCIS Board (DILCIS Board, 2021b). They have also defined how to include SIARD files in such packages (DILCIS Board, 2021a). These formats build upon the metadata standards METS and PREMIS, both from the US Library of Congress.

Pitfalls

Even when using standardised preservation formats, there are things that can go wrong:

Missing information. Databases often contain data one does not wish to preserve, for example, because they are obsolete, duplicates, of low value, or contain sensitive information. There is a risk, however, that some information is left out that would be valuable in the future.

¹ Zenodo, operated by CERN: <https://zenodo.org/>

² This would be a form of “bit-level preservation”.

Low-quality or missing metadata. To ensure a low cost barrier for future access, it is important to include high-quality metadata and documentation. If the quality is poor or if these elements are missing entirely, it may significantly raise the cost of future access.

Dependencies. Real-life databases often contain dependencies on external systems. If these dependencies are not properly handled and resolved in some way, there is a risk that the preserved data will be rendered partially or completely unusable at some point in the future, when those external systems are gone.

Weak authenticity. The value of a preserved database is limited if it is unknown whether it is a faithful representation of information contained in the original sources. Thus, the archive should include information on how it was produced, its *provenance*.

Manually preserving a database so that the information remains accessible may require a complex series of extraction, transformation and migration steps. In the end, it might not be clear even to experts involved exactly what steps were taken—which undermines authenticity. It is also a practical problem in case the process must be repeated, for example, because a defect has been found.

E-ARK information packages may contain *event* elements describing the preservation actions taken. These follow the PREMIS standard (PREMIS Editorial Committee, 2015). There is also a provenance framework, PROV by W3C, and extensions such as PROV-IO⁺ (Han et al., 2023) that allow for more details. However, neither representation is *executable*, so reproducibility is not guaranteed.

Databases and Datasets

A preserved relational database may often be referred to as a *dataset*, but in most cases a dataset is just a set of tables or multi-dimensional arrays. Databases usually have a more complex structure, and the database schema can be valuable as metadata expressing the underlying assumptions of the system that produced the data. Nevertheless, one approach to preserving databases involves migration to a “dimensional model” (Ur, Muzammal, David, & Ribeiro, 2015). This means applying techniques such as denormalization that are most commonly used for creating data warehouses. The goal is to make the information in the database more accessible, but the result—which is more like a dataset—has less structure and more redundant data.

The DbSpec Method

Overview

The solution we propose is to specify each preservation step in a formal language, DbSpec, and execute these steps using a software tool known as an *interpreter* (see Figure 1). Thus, the job of the experts goes from *performing* manual steps to *expressing* those steps through the specification. At various points, draft specifications can be executed and the result checked using tests that are themselves part of the specification. This is repeated until the result is satisfactory. As the specification embodies all the work done by the experts, these iterations have low marginal cost, although there may be waiting time involved, especially for large databases. The process also becomes more transparent, making the specification (and how it evolves over time) visible to all project members; but most importantly, the specification can be included in information packages as provenance information, showing precisely how the contents were produced.

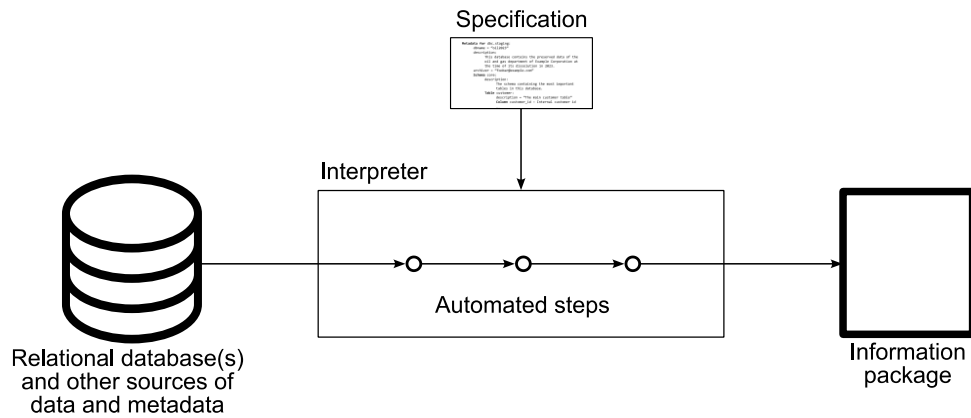


Figure 1. Automated preservation process.

Specification Language

While readable for humans, a DbSpec specification can be seen as a computer programme. As such, the language is inspired by the scripting language Python, but it is simpler and geared towards the preservation of relational data. For the most part, a DbSpec specification describes a series of steps to be executed sequentially; but in order to facilitate reuse, the language also has conditional branching (if-then-else) and a basic looping construct.

When a specification is executed, many steps will be delegated to other tools. Conceptually, this is similar to languages for automating sequences of invocations of command-line tools, but the DbSpec mechanism is to use *embedded scripts*. These come in two flavours: SQL and other executable scripts. While SQL statements and queries will be sent to the relevant RDBMS for execution, other scripts are executed in subprocesses by external tools (e.g., Bash or Python). In both cases, there are simple ways to provide input and receive output. This seamless integration of embedded scripts means that a specification can provide a unified view of fundamentally different steps.

The DbSpec language also contains other elements, such as statements for establishing database connections, making assertions, and logging. Assertions should be used to check representation invariants that are not guaranteed by database constraints. This is especially important when planning to reuse the specification with other data sources.

The Structure of a Typical Specification

While we think DbSpec will prove useful in a range of settings, the focus so far has been on preserving data from relational databases as SIARD archives. First, a *staging database* is created for the database schema and contents we want to preserve. When possible, we suggest restoring a database backup. Next, additional data can be imported using embedded scripts. The staging database is then transformed, and there should be assertions expressing pre- and postconditions. Additional metadata may be associated with the database objects, and a SIARD archive is produced, both using special language constructs. Finally, an information package is compiled using another embedded script.

Figure 2 shows some of the steps in a simple specification. Notice how the embedded scripts are separated from the other parts of the specification using indentation only. When advanced extraction and transformation steps are needed, the specification will be more complex. DbSpec does not eliminate the need for technical expertise. There are no high-level constructs or graphical user interfaces that try to guess what you want. Instead, this must be expressed in precise terms—using the same or similar tools and technologies as system developers and database administrators.

```

#!/usr/bin/env dbspec
Parameters:
    host - PostgreSQL hostname
    ...
Log "Restore dump to database ${db}"
Execute using "/bin/bash":
    set -e
    ...
    psql -v ON_ERROR_STOP=1 -c "CREATE DATABASE ${db}"
    pg_restore --no-owner --role=${user} -d ${db} dvdrental.tar
    ...
Set dbc = connection to "jdbc:postgresql://${host}:${port}/${db}" with:
    user = user
    password = "Use parameter instead"
    ...
Log "Delete customer data"
Execute via dbc:
    DROP VIEW customer_List;
    ...
    DROP TABLE customer;
    DELETE FROM address WHERE address_id NOT IN (
        SELECT address_id FROM staff
        UNION SELECT address_id FROM store);
    ...
Log "Check that every film has at least one category"
Set without_category = result via dbc:
    SELECT title FROM film WHERE film_id NOT IN (SELECT film_id FROM film_category)
Assert without_category.size == 0
    ...
Log "Add metadata"
Metadata for dbc:
    description = "Example DVD rental database"
    Schema public:
        Table film:
            description:
                Film data such as title, release year, etc.
            Column film_id - Internal primary key
            ...
        View sales_by_store:
            ...
Log "Produce SIARD file"
Output dbc to "${db}.siard"
    ...

```

Figure 2. Excerpts from a simple DbSpec file.

Reproducibility

The production of information packages and other digital artifacts using DbSpec is repeatable in the sense that re-executing the specifications with the same input in an identical operating environment should yield similar results. Ideally, only preservation metadata should change—such as timestamps—but embedded scripts may also introduce randomness. Avoiding this is the responsibility of those writing the specification. Even in special cases when, say, one only wants to preserve a random selection of entries, this can be made deterministic by using a pseudo-random generator with a fixed seed value.

The next challenge is how to ensure that the environment is the same. For instance, we recommend that the RDBMS used for the staging database is the exact same version as in the originating system. One solution might be to use container technologies (discussed below). However, in complex data extraction scenarios, the environment might have to include a complete, running system—or even other systems, when there are external dependencies. In these situations, we recommend that the specification be divided into two stages, where the full environment is only needed for the first stage. This will produce the input necessary for the second stage, which should contain most of the preservation steps.

Splitting the specification is also a way to ensure future access to input that is close to the original in cases when the complete input to the first stage cannot be preserved, for example, for

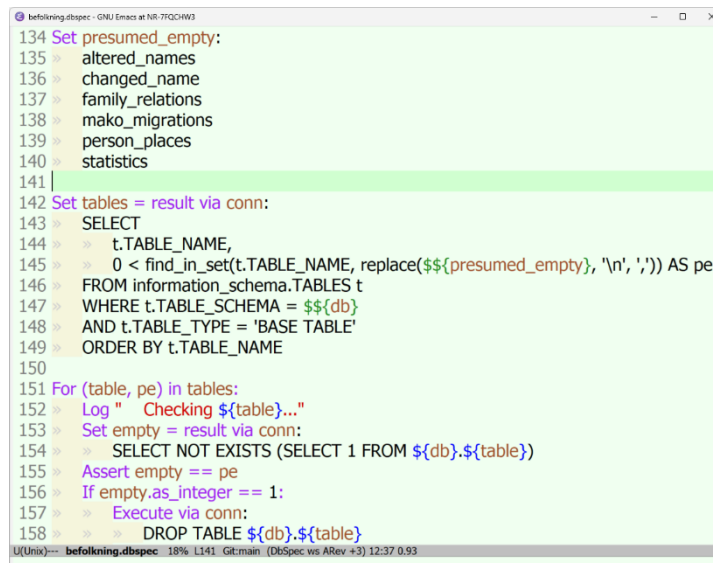
legal reasons: we then recommend that the output from the first stage as well as any additional input to the second stage be included in the information package together with the specifications and container definitions for each stage.

Testing and Evaluation

The DbSpec language and interpreter has gone through three test phases, with increasing realism. This has led to many improvements and lent weight to the usefulness of our approach, even though the testing so far has not been done independently of this project or as part of an actual preservation effort.

Initial Example Specifications

During the initial development, we produced two basic examples of DbSpec specifications using freely available PostgreSQL databases.³ However, we quickly realised that while readable, the way DbSpec uses indentation makes it difficult and error-prone to write specifications without editor support. Fortunately, we were able to repurpose the parser into an extension for the text editor Emacs, which we have bundled with the interpreter. This highlights DbSpec keywords, syntactical structure and errors, but it lacks such support for SQL and other languages used in embedded scripts (see Figure 3). In some cases, the preferred solution has been to develop embedded scripts using other tools before copying the text into the DbSpec file.



```

134 Set presumed_empty:
135   altered_names
136   changed_name
137   family_relations
138   mako_migrations
139   person_places
140   statistics
141
142 Set tables = result via conn:
143   SELECT
144     t.TABLE_NAME,
145     0 < find_in_set(t.TABLE_NAME, replace(${presumed_empty}, '\n', ',')) AS pe
146   FROM information_schema.TABLES t
147   WHERE t.TABLE_SCHEMA = ${db}
148   AND t.TABLE_TYPE = 'BASE TABLE'
149   ORDER BY t.TABLE_NAME
150
151 For (table, pe) in tables:
152   Log "  Checking ${table}..."
153   Set empty = result via conn:
154     SELECT NOT EXISTS (SELECT 1 FROM ${db}.${table})
155   Assert empty == pe
156   If empty.as_integer == 1:
157     Execute via conn:
158       DROP TABLE ${db}.${table}
  
```

Figure 3. Excerpt from example DbSpec specification in the Emacs editor.

We also discovered that the software we currently rely on for generating SIARD files did not always work as expected. For example, when using PostgreSQL for the staging database, the metadata of views do not contain their defining queries, and materialised views are completely ignored.⁴ In one of the examples, we have therefore shown how such information can instead be included using embedded scripts, but this is not ideal.

³ These have been made available alongside the source code of the interpreter. See <https://github.com/immortalvm/dbspec/tree/main/examples>, folders ‘dvdrental’ and ‘adventureworks’.

⁴ Bug report: <https://github.com/sfa-siard/SiardGui/issues/65>

DbSpec Evaluation

A limited evaluation of the DbSpec approach—including the language, interpreter and recommendations—was performed during the autumn of 2024. As part of the evaluation, the evaluator wrote a DbSpec specification for the example database Northwind Traders for Microsoft SQL Server. While generally positive, the evaluation also observed that no testing had yet been performed with realistic production databases. Those may have hundreds of tables and be very hard to understand, in which case, writing DbSpec specifications seems to require a lot of potentially error-prone manual work. The evaluation also revealed several software errors that have since been fixed.

Case Study (2025)

Lastly, we have used DbSpec to curate and preserve parts of the actual production database of the Historical Population Register in Norway (Norwegian: *Historisk befolkningsregister*, HBR for short).⁵ Figure 3 shows a part of this specification. Whereas the size of this database was moderate (22 GiB) compared to really large databases, it was significantly larger than the example databases used earlier. It also involved a different RDBMS (MariaDB) and had some issues that are common in real-world database systems:

- Partially outdated documentation and almost no descriptions of the tables and columns in the database itself.
- Columns and whole tables that are no longer or not yet in use.
- Unclear representation invariants; few constraints other than primary keys.
- Secondary keys used as foreign keys.
- No database objects other than tables (e.g., views or functions).
- Inconsistent naming conventions (e.g., mixing Norwegian and English).
- Large variation in data quality.
- Dependence on other data sources (e.g., through unexplained region codes).

On the other hand, the number of tables was quite low (24), and some of the developers were available to answer questions. Alas, due to limited time and resources, we had to prioritise the testing of DbSpec over the preservation outcome; and since the work was not prompted by the needs of the data owner or constrained by regulations, the preservation decisions were somewhat arbitrary, and we did not properly test the hypothesis that DbSpec can boost communication between stakeholders and experts. However, we did observe that when making substantial transformations, the amount of embedded SQL can make the specification unwieldy and less suitable for such exchanges.

The case study prompted a number of simplifications and other improvements to the language semantics and the interpreter.⁶ We also identified some issues that could not be resolved “on the fly”. In particular, the size of the database made it impractical to execute the whole specification for every update, but the current alternatives are not very good either (e.g., commenting out parts of the specification or splitting it into multiple files). The size also slowed down the process of understanding the database and deciding what to do with it. For example,

⁵ The database contains public information which is made available at <https://histreg.no/> (which is powered by this database), and we have made the DbSpec script available alongside the other examples, but the database dump it relies on is not public.

⁶ See <https://github.com/immortalvm/dbspec/releases> for details.

SQL queries for checking (presumed) representation invariants could take several minutes. On the positive side, the SIARD extraction worked better than for the PostgreSQL example databases. All in all, DbSpec made it easier to properly structure and document the preservation steps, but the process is still slow and labour-intensive.

Discussion

In this section, we will discuss in more detail the rationale behind DbSpec, some issues that have not yet been fully resolved, and a few ideas for future work.

Avoiding the Pitfalls

Above we mentioned four things that can go wrong when preserving relational data. Here are some ways these risks can be reduced when using DbSpec:

Missing information. DbSpec promotes engaging with the data instead of simply attempting to preserve everything as is. This will often involve throwing some data away, but this might actually increase the *information* in the archive—for example, by resolving inconsistencies—at least if an initial, native backup is also preserved. When some of the input data should not be preserved in any form or when it must be extracted from external data sources, we recommend splitting the specification (cf. ‘Reproducibility’ above).

Low-quality or missing metadata. Metadata statements make it easy to add documentation strings and other metadata to the generated SIARD archives. Moreover, DbSpec specifications may contain transformations that make the database schema more valuable as metadata.

Dependencies. DbSpec also facilitates transformations that eliminate dependencies on external data sources, making the archive self-contained. (Nevertheless, one should keep the external references—especially if the external data source contains relevant information that cannot be included in this archive, e.g., for legal reasons.)

Weak authenticity. We express the preservation steps in terms of an executable specification that can be preserved alongside the data (see ‘Reproducibility’ above, and also ‘Data Protection and Provenance’ below).

SIARD is not a Backup Format

A transformation action in the DCC Lifecycle Model is described as a future event: when new data is created from what was preserved. The idea is perhaps that what we preserve should be as close to the original as possible. Every transformation can lead to loss of information and therefore also of authenticity. This might also be the reason why tools for creating SIARD archives seem geared towards preserving the original database as is—postponing any transformations to the future, after the database has been restored from the SIARD archive to a suitable RDBMS. However, this approach can be risky. The SIARD file may not contain all the relevant data and metadata. Perhaps the data was not even in the original database, but in external files or systems that are no longer available. Also, when the original system is gone, we may no longer be able to extract correct information from the data we do have.

Instead, we strongly believe that some transformations should be considered preservation steps, executed in an environment as close to the original system as possible, and specified using DbSpec. For reproducibility, we also recommend including in the information package a “native” backup of the original database as well as any other input files. This is in accordance with the CITS SIARD standard, where such files can form a separate “representation” (DILCIS Board, 2021a).

Containerisation

For reproducibility, it is not always enough to know the input and the preservation steps. The operating environment is important as well. To some extent, this can be specified using assertions, but a better option is arguably to use so-called “OS-level virtualisation” with isolated and fully specified environments called *containers*. Using containers, one can ensure that each specification is executed in the intended operating environment. Since a container definition is itself an executable specification, it should be included in the information package alongside the DbSpec file—as “preservation metadata”.

In the initial examples (discussed above), we have shown how to achieve this with the help of the containerisation software Docker.⁷ With Docker, each container is an instance of an *image*, and what we referred to as a container definition is a set of instructions on how to build one or more images, known as a *Dockerfile*. We build two images.⁸ The first starts by setting up the operating environment for the preservation steps. This must include the DbSpec interpreter and all the input files. Next, the specification is copied into the image, the RDBMS is started, and the specification is executed. The second image copies the DbSpec output from the first so that this output can be saved to disk as part of building the images. In other words, there is no reason to create actual containers other than to inspect the current state when developing specifications iteratively. If executing the draft specification from start to finish is slow, it can be split into multiple sub-specifications during development in order to exploit the caching mechanism of Docker.

Unfortunately, virtualisation is not always an option. As mentioned above (under ‘Reproducibility’), some preservation steps might have to be executed in an environment which is not easily reproduced; but in some cases, it can still be done if instead of lightweight containers, we use full-fledged virtual machines, for example, specified and managed through the ‘libvirt’ virtualisation API.⁹

Iterative Development and Executable Specifications

Successful database preservation is contingent on informed decisions regarding what to preserve, correct and efficient execution of these decisions, and validation. This is similar to software development projects, where iterative and incremental development is the norm; even more so when the implementation is in terms of executable specifications, which we have shown facilitate such iterations. Notice, however, that in software engineering, executable specifications are usually understood as requirements expressed in such a way that they can also be executed as automated acceptance tests—usually in terms of concrete examples (Adzic, 2011). This would also be valuable in many database preservation projects, but we have not considered how such tests might be integrated with DbSpec.

In many cases, the preservation effort is not a one-off event. If the whole process must be repeated at regular intervals, one should be able to use the previous specification as a starting point, adjusting only for structural changes, for example, to the database schema. Even if the preservation is not done very often, we recommend keeping the specification in sync with the database and re-executing it as part of the testing when there are changes.

Why Another DSL?

DbSpec is a domain-specific language (DSL), but in principle one could use a general-purpose scripting language instead (e.g., Python), or even a compiled language. Another option would be

⁷ Docker, Inc.: <https://www.docker.com/>

⁸ In fact, we build six images in total: first, one containing the DbSpec interpreter, next, one for the parts that can be shared between the two examples (inheriting from the first), and finally, two for each example (defined in one Dockerfile per example).

⁹ See <https://libvirt.org/>.

to define a library/extension for a programming language, a so-called “internal DSL”. In complex data extraction scenarios, the best option might be to use the programming language of the originating system—at least for the initial preservation steps—as this would make it easier to reuse or adapt existing source code and libraries. However, using an “external” DSL has some advantages:

Readable syntax. Since the new syntax does not have to be compatible with an existing language, we can optimise for readability—that is, make sure that the preservation steps can be expressed clearly and succinctly.

Simple overall structure. Whereas the embedded scripts can be arbitrarily complex, DbSpec itself is very simple and easy to understand. This also makes it easier to develop tool support for the language, for example, within IDEs and text editors other than Emacs.

Language agnostic. Except for special handling of SQL, DbSpec treats every scripting language the same. Mixing languages is fine, too.

Easy to extend and adjust. With a small DSL, it is easier to adjust the syntax or extend the language with new constructs that are also readable.

Taken together, we think using DbSpec can improve communication both with future users of preserved databases and within ongoing database preservation projects, possibly even within a broader database preservation community. This facilitates shared understanding and informed discussions, even in preservation projects that require cross-disciplinary and cross-organisational collaboration. Thus, we hope that DbSpec specifications may serve as a form of “boundary objects” (Francis & Kong, 2014). That being said, testing has shown that while DbSpec promotes a simple overall structure, this is less apparent when the specifications get long, with possibly quite complex embedded scripts. As a remedy, we are considering adding new language constructs for indicating sections or moving parts of the specification to separate files.

SIARD Output and Metadata

Even though DbSpec might also be used for other database transformation and preservation tasks, the language is currently geared towards creating SIARD archives—with special syntax for producing archive files and for specifying additional SIARD metadata. A challenge when producing these archives is that the format is based on the SQL:2008 standard, which all major RDBMSs diverge from in some ways. Thus, the process requires RDBMS-specific conversions of both the data and the database schema. In the DbSpec interpreter, this has been solved by (re)using components from *SIARD Suite* (Swiss Federal Archives, 2023), an existing software tool for producing database archives of existing databases.

Relying on external SIARD components limits the control we have over the resulting archive. For example, we are not able to limit the extraction to some database schemas¹⁰ or to set certain metadata fields, such as ‘*producerApplication*’. Furthermore, the automatic conversions and “best effort” approach of SIARD Suite is at odds with our recommended strategy of making such decisions explicit. This was highlighted during testing when it was discovered that SIARD Suite does not extract any view queries from PostgreSQL databases. Going forward, we see two options: either incorporate the SIARD extraction fully into the DbSpec interpreter—possibly using the SIARD Suite source code as a starting point—or remove the special syntax for producing SIARD archives altogether, leaving this to embedded scripts.¹¹

Preserving data has limited value if no one knows what they mean or whether the information can be trusted. For relational data, one first needs to know the structure of the database—the schema. This is mandatory metadata in SIARD archives. In addition, there can be textual descriptions of each database object, information about the database as a whole, and some information about the archival process—such as who carried it out. When creating SIARD

¹⁰ Instead, the DbSpec user must make sure that the database user/account only has access to the database objects they want to preserve.

¹¹ Such scripts might invoke SIARD Suite explicitly via `SiardFromDb/siard-from-db`.

archives using DbSpec, it is even possible to include the specification itself,¹² but it makes more sense to wrap both the specification and the SIARD file in an information package. DbSpec does not have built-in support for compiling such packages or the METS/PREMIS metadata inside, but it can be done manually using embedded scripts.¹³

Data Protection and Provenance

The SIARD archives created by DbSpec include information about (SQL) users, roles and privileges. We consider adding an option to prevent this. However, these features can be useful for managing access to sensitive data. The SIARD format does not support other access control features such as row-level security, which is not yet part of the SQL standard. Database transformations must be used for such information to be preserved, such as adding a column for the access permissions.

The SIARD format does not enforce any access restrictions, meaning anyone with access to the archive can read all the data. While there might be ways to solve this using encryption (assuming proper key management), a more straightforward solution would be to create multiple versions of the archive. For example, one version could contain all data, while another, with sensitive information removed, could be made publicly available. This can be achieved with a single specification.

Additionally, we aim to ensure that the final output—presumably one or more information packages—(1) was produced by a trusted entity, (2) has not been tampered with, and (3) is indeed the result of running DbSpec in the right environment and with the correct input. While (1) and (2) can be achieved using digital signatures, this is not enough for (3). Ideally, one should be able to execute the specification and get the same result (cf. ‘Reproducibility’). How to achieve sufficient assurance when this is not possible (e.g., for an initial preservation phase) is beyond the scope of this text; but one idea would be to adapt the framework SLSA,¹⁴ which tries to solve similar problems for software artifacts.

Limitations and Future Work

Despite recent progress, the DbSpec language and interpreter should be considered prototypes. Testing has been limited—for example, the interpreter has mostly been tested under Linux. There are also some inconvenient limitations, such as too little control over the generated SIARD files and no built-in support for creating information packages. Many common tasks must be handled using embedded scripts. Also, even though combining DbSpec with Docker works reasonably well, more could be achieved with tighter integration.¹⁵

From a theoretical perspective, DbSpec can be seen as a tool for *data exchange*, that is, “the problem of taking data structured under a source schema and creating an instance of a target schema that reflects the source data as accurately as possible” (Fagin, Kolaitis, Miller, & Popa, 2005). Since that seminal article, several tools and formalisms have been put forward for expressing such transformations precisely and ensuring correctness. Rather than formulating the transformation steps directly using SQL, the desired relationship between input and output is expressed declaratively—using examples or logical expressions—and correct SQL is generated automatically. This is perhaps more useful in a migration scenario where the target schema is known in advance, than for database preservation where one often has to deal with nonstandard

¹² SIARD archives may contain “additional files, such as style sheets” in the ‘header’ directory.

¹³ In *examples/adventureworks/package.dbspec*, we sketch how to do this using eArchiving Tool Box (see <https://github.com/E-ARK-Software/eatb>).

¹⁴ See <https://slsa.dev/> (Supply-chain Levels for Software Artifacts).

¹⁵ In the process we might switch to a different virtualization platform, such as Podman or ‘libvirt’.

features of the RDBMS.¹⁶ Nevertheless, it would be interesting to consider how declarative data exchange might be integrated into DbSpec.

Within the iDA project, DbSpec will also be used to specify interfaces for *retrieving* data from the preserved databases, potentially in the far future. For this purpose, the language also includes “*Command*” statements not discussed here. The DbSpec interpreter is available as open-source software under the CDDL licence at:

<https://github.com/immortalvm/dbspec>

This Git repository also includes a short manual, the Emacs extension, the DbSpec parser (as a submodule), and a few examples.

Related Work

Inspirations for our approach to database preservation have been the discussion of boundary objects in (Francis & Kong, 2014) and a draft report from the relational database archiving interest group of the DILCIS Board describing the use of SIARD in several European countries, (Groven et al., 2020). The DbSpec interpreter can be seen as an extension of existing software, most importantly SIARD Suite (Swiss Federal Archives, 2023), which can be used for producing database archives from existing databases. Database Preservation Toolkit (DBPTK) can be used for producing SIARD archives much in the same way (Ramalho, Ferreira, Faria, & Ferreira, 2020). It has some more advanced options and seems more flexible in general, but it has not yet been updated to the latest version of SIARD (2.2). As mentioned above, the DbSpec interpreter uses components from SIARD Suite for producing SIARD files.

In addition to the command-line tools for archiving (and restoring) databases, both DBPTK and SIARD Suite have desktop applications where one can inspect SIARD archives and edit their metadata. There is no support for transformations, data validation or combining data from multiple sources; but OpenRefine (Antonin Delpuch et al., 2023) does something similar for datasets, that is, individual tables of data. This is an application where the user can explore and transform their dataset and export the result in various formats. What makes it similar to DbSpec is that these transformations are all recorded and can be reused with other datasets (Li & Ludäscher, 2023). However, OpenRefine is not suitable for transforming whole databases; and the transformations supported are quite limited.

At the other end of the spectrum are the more general “scientific workflow systems”. Of these, it seems DbSpec has the most in common with Common Workflow Language (CWL) (see Crusoe et al., 2022). Database transformation and preservation workflows can be expressed more easily and directly in DbSpec due to its built-in constructs for database connections and embedded scripts, but scientific workflow systems with built-in support for performing tasks in parallel or in the cloud might be better suited when the amount of data is very large. Combining the two approaches—for example, via CWL scripts embedded in DbSpec—is something we would like to explore.

DbSpec is not the first computer language where one can embed code written in other languages. Other examples include inline assembly in C, HTML in PHP, and so forth. Syntactically, DbSpec is also similar to “literate programming” (Knuth, 1984), where the source code is embedded in a document intended for humans to read. In DbSpec, as in most programming languages, it is the other way around, with comments embedded in the source code. However, we are considering adjusting the language to give such texts a more prominent role. Ideally, they should not only describe the preservation steps, but also the rationale behind them. This might also serve as a foundation for delegating parts of the DbSpec specifications to large language models (LLMs) as in Shi et al. (2024).

¹⁶ When a database uses features of the RDBMS that are not part of the SQL standard, we recommend changing this before producing the SIARD representation. In such cases, the DbSpec specification will usually have to include embedded scripts with nonstandard SQL.

Conclusion

We have presented a new method for database preservation, by which we mean the extraction, transformation and validation of relational data for future access. The method involves composing executable specifications in a new domain-specific language, DbSpec. This has several advantages compared to more manual approaches. The process becomes more reliable, and the specification doubles as precise documentation of each preservation step—increasing the value of the preserved data. In most cases, it is also more efficient since the process is more easily repeated. Specifications written in DbSpec are easy to read and have a simple overall structure. This makes it easier to maintain a shared, high-level view of the preservation steps than when using a general programming language, other workflow languages or a collection of individual scripts. We also believe that it makes the specification more valuable as metadata.

From a quality assurance point of view, our method establishes a controlled, repeatable process suitable for integration into a comprehensive quality assurance programme. As long as the original database exists, the preservation steps can be reproduced at low marginal cost. A structured process of transformation and testing can be applied according to the organisation's quality standards. This reduces the probability of defects in the preserved database and hence the risk that data access will be hindered in the future, when the original database is no longer extant.

Acknowledgements

This work is part of the project Immortal Database Access supported by the Eurostars programme, project number E1622 iDA. We would also like to thank Arne-Kristian Groven for valuable input and discussions.

References

- Adzic, G. (2011). *Specification by example: How successful teams deliver the right software*. USA: Manning Publications Co.
- Delpuch, A., Morris, T., Huynh, D., Weblate (bot), Mazzocchi, S., ... Chandra, L. (2023). *OpenRefine/OpenRefine: OpenRefine v3.7.7*. Zenodo. doi: [10.5281/ZENODO.10220116](https://doi.org/10.5281/ZENODO.10220116)
- CRL, & OCLC. (2007). *Trustworthy Repositories Audit & Certification: Criteria and Checklist*. The Center for Research Libraries and Online Computer Library Center, Inc.
- Crusoe, M. R., Abeln, S., Iosup, A., Amstutz, P., Chilton, J., Tijanić, N., ... The CWL Community. (2022). Methods included: Standardizing computational reuse and portability with the common workflow language. *Communications of the ACM* 65(6), 54–63.
- Digital Preservation Testbed. (2003). *From digital volatility to digital permanence: Preserving databases*. The Hague: ICTU Foundation.
- DILCIS Board. (2021a). *CITS SIARD: E-ARK Content Information Type Specification for Relational Databases using SIARD*. Digital Information LifeCycle Interoperability Standards Board. Retrieved from <https://dilcis.eu/content-types/cs-siard>

- DILCIS Board. (2021b). *E-ARK SIP: Specification for submission information packages*. Digital Information LifeCycle Interoperability Standards Board. Retrieved from <https://earksip.dilcis.eu/>
- DILCIS Board. (2021c). *SIARD format specification*. Digital Information LifeCycle Interoperability Standards Board. Retrieved from <https://siard.dilcis.eu>
- Fagin, R., Kolaitis, P. G., Miller, R. J., & Popa, L. (2005). Data exchange: Semantics and query answering. *Theoretical Computer Science* 336(1), 89–124. doi: [10.1016/j.tcs.2004.10.033](https://doi.org/10.1016/j.tcs.2004.10.033)
- Francis, P., & Kong, A. (2014). Making the strange familiar: Bridging boundaries on database preservation projects. *Proceedings of the 11th International Conference on Digital Preservation*. Presented at the iPRES 2014.
- Groven, A.-K., Merenmies, M., Ovaska, V.-M., Hovdan, A. K., Alvik, L. J., & Rätsep, L. (2020). *Preserving databases using SIARD: Experiences with workflows and documentation practices*. DILCIS Board. Retrieved from <https://dilcis.eu/content-types/cs-siard>
- Han, R., Zheng, M., Byna, S., Tang, H., Dong, B., Dai, D., ... Wolf, M. (2023). *PROV-IO+: A Cross-Platform Provenance Framework for Scientific Data on HPC Systems*. arXiv. doi: [10.48550/arXiv.2308.00891](https://doi.org/10.48550/arXiv.2308.00891)
- Higgins, S. (2008). The DCC Curation Lifecycle Model. *International Journal of Digital Curation* 3(1), 134–140. doi: [10.2218/ijdc.v3i1.48](https://doi.org/10.2218/ijdc.v3i1.48)
- Knuth, D. E. (1984). Literate Programming. *The Computer Journal* 27(2), 97–111. doi: [10.1093/comjnl/27.2.97](https://doi.org/10.1093/comjnl/27.2.97)
- Li, L., & Ludäscher, B. (2023). On the reusability of data cleaning workflows. *International Journal of Digital Curation* 17(1), 6. doi: [10.2218/ijdc.v17i1.828](https://doi.org/10.2218/ijdc.v17i1.828)
- Munafò, M. R., Nosek, B. A., Bishop, D. V., Button, K. S., Chambers, C. D., Percie du Sert, N., ... Ioannidis, J. (2017). A manifesto for reproducible science. *Nature Human Behaviour* 1(1), 1–9.
- PREMIS Editorial Committee. (2015). *PREMIS Data Dictionary for Preservation Metadata, Version 3.0*. Library of Congress. Retrieved from Library of Congress website: <https://www.loc.gov/standards/premis/v3/premis-3-0-final.pdf>
- Ramalho, J. C., Ferreira, B., Faria, L., & Ferreira, M. (2020). Beyond Relational Databases: Preserving the Data. *New Review of Information Networking* 25(2), 107–118. doi: [10.1080/13614576.2021.1919398](https://doi.org/10.1080/13614576.2021.1919398)
- Rosenthal, D. S. H. (2010). Format obsolescence: Assessing the threat and the defenses. *Library Hi Tech* 28(2), 195–210. doi: [10.1108/07378831011047613](https://doi.org/10.1108/07378831011047613)
- Shi, K., Altnbükken, D., Anand, S., Christodorescu, M., Grünwedel, K., Koenings, A., Naidu, S., Pathak, A., Rasi, M., Ribeiro, F., Ruffin, B., Sanyam, S., Tabachnyk, M., Toth, S., Tu, R., Welp, T., Yin, P., Zaheer, M., Chandra, S., & Sutton, C. (2024). *Natural Language Outlines for Code: Literate Programming in the LLM Era*. [Preprint. To appear in *33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25)*]. ArXiv. doi: [10.48550/arXiv.2408.04820](https://doi.org/10.48550/arXiv.2408.04820)

Swiss Federal Archives. (2023). *SIARD suite*. Retrieved from <https://www.bar.admin.ch/bar/en/home/archiving/tools/siard-suite.html>

Ur, A., Muzammal, M., David, G., & Ribeiro, C. (2015). Database Preservation: The DBPreserve Approach. *International Journal of Advanced Computer Science and Applications* 6(12). doi: 10.14569/IJACSA.2015.061235