AURORA: An Automated Database Schema Change Logging System

Bradley Camilleri
Department of Computer Information Systems
Faculty of ICT, University of Malta
Msida, Malta
e-mail: bradley.camilleri.22@um.edu.mt

Joseph G. Vella
Department of Computer Information Systems
Faculty of ICT, University of Malta
Msida, Malta
e-mail: joseph.g.vella@um.edu.mt

Abstract— **Database schema changes pose a critical challenge when updating Computer Information Systems (CIS) since they require careful synchronization between codebase updates and the database. Traditional approaches to schema evolution often lack automated tracking of schema changes, leading to duplicated coding, data loss, inconsistency, and increased downtime. This paper presents AURORA, an automated solution that captures the Data Definition Language (DDL) and Data Manipulation Language (DML) operations performed by the data designer. The captured data is then used to drive client-side database schema upgrades. By leveraging Database Management System (DBMS) event triggers, this system ensures that any relevant DML and DDL events are logged and used for the generation of a schema upgrade script. This script includes the schema changes performed by the data designer and the respective pre and post data checks to ensure database consistency and integrity at the client's database. This client-side script also includes a rollback mechanism to reverse a schema upgrade in the case of failure. AURORA was evaluated through a set of real-world scenarios which highlighted its practicality, coverage and validity.**

Keywords-schema evolution; computer information systems; databases; software deployment; software upgrades.

I. INTRODUCTION

A Computer Information System (CIS) implementation, e.g., codebase and database structures, is static in contrast to the dynamic environment in which its clients operate. Therefore, for a CIS to remain relevant for its clients, it is subject to updates which are orchestrated by the software provider [1][2]. However, if an application update includes database changes, an issue arises since the client's database must undertake the appropriate script to migrate it to the desired state while ensuring that its data is preserved. This script must be distributed to all clients alongside the updates to the application program and must be validated at each client-side database prior to running the updated version of the software. This local action is required as most applications have a portion of their codebase with a hard-coded representation of the schema in use [3].

Generating these database upgrade scripts is challenging since, if the Structured Query Language (SQL) queries performed on the database during development are not explicitly logged, they are lost. As a result, this leads to the tedious and time-consuming task of redoing the queries to generate a change script for the schema upgrade. Other challenges include ensuring that each change construct has an undoing action, in case an update cannot proceed at the client, and that schema changes that cause data loss, e.g., delete operations, are supported with redundant structures to ensure that these operations remain undoable during the upgrade process.

Moreover, one must account for any additional artefacts that a client has added independently of the software provider. For example, a view created on the old schema can become nonsensical when an upgrade script changes the base tables it depends on.

Upgrades to database schemas are not taken lightly. Firstly, there is the issue of application availability, i.e., the system is offline whilst a client is executing an upgrade. Secondly, an upgrade must not purge nor make data unreachable for the client. Thirdly, errors during the upgrade process can leave the database in an inconsistent state. Therefore, an adequate mechanism to roll back the database to a stable checkpoint is required. An example of an upgrade going wrong is the case of Revolut's authentication system [4] where the deletion of, what seemed to be, an unused table column, caused the entire authentication system to fail.

AURORA addresses the issue of generating database upgrade scripts by implementing a mechanism that automatically tracks the Data Definition Language (DDL) and Data Manipulation Language (DML) operations performed by a data designer. This automation also generates the respective pre, post, and undoing code fragments for the captured DDL and DML constructs.

AURORA also aims to make it easier for the client-side execution of a schema upgrade by generating an upgrade script that only requires minimal manual database administrator (DBA) intervention even when the upgrade process fails, or the upgrade process cannot proceed. Moreover, the client-side script keeps the client's existing data and adapts it to the new schema, and it also allows the client's DBA to easily reverse a database upgrade, therefore reducing downtime and addressing recoverability.

At the software provider, AURORA tracks changes to schemas, tables (for both DML and DDL changes), views, functions, and table triggers. To achieve this, AURORA requires the DBMS to support DDL and DML triggers since these are the mechanisms which are used to track the schema changes. Any application code that is external to the database, e.g., front end code, is outside the system's scope.

This paper is divided into six sections. Section one provides the reader with an introduction to the area, its challenges and the work in this paper. Section two provides

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

the reader with an overview of some of the relevant literature in the area, and a few existing vendor-specific solutions. In Section three, the requirements and design of AURORA are presented, and Section four details the implementation. Section five includes a real-world evaluation of the system, and Section six concludes this paper.

II. BACKGROUND & LITERATURE REVIEW

Version Control (VC) software allows development teams to manage and keep track of their source code revisions [5], allowing them to access, integrate and back track to older versions of the code as needed. VC has been used to track changes in code through tools like Git [6] and to track document and file updates through cloud storage solutions like OneDrive [7], Google Drive [8] and Dropbox [9]. Moreover, numerous tools exist to perform VC on the database during schema evolution [10]–[15].

Schema evolution in database design involves the modification of a schema artefact through a sequence of schema changes (see Table 1) while retaining the consistency of existing constraints and data [16]. Schema evolution implies codebase development (since queries that depend on the changed artefacts need to be examined for continued validity), but not all codebase development results in schema changes. Software providers deploy application program upgrades to their clients much more frequently than database updates since the latter are usually delayed and performed in a batch. This occurs due to the sensitive nature of schema changes and the fact that, as discussed, a schema change results in an application program update.

Apart from the application's codebase, schema evolutions also effect the Extract, Transform and Load (ETL) [17] scripts which are used to generate Online Analytical Processing (OLAP) systems and the global schema of a multi-database system. The freshness rate of Hybrid transactional/analytical processing (HTAP) systems is also determined by the recency of the schema in use [18]. Therefore, understanding how a schema has evolved over time is crucial to ensure that these scripts and systems are adequately updated with the newest version of the schema.

To better manage schema evolution during application development, Curino et al. developed PRISM [2][19] which allows developers to specify schema changes through Schema Modification Operators (SMOs). PRISM then uses these SMOs to update a schema, and its data, to the target version. PRISM can also re-write a sub-set of queries to match the target schema, ensure data preservation and create undo operations from each SMO [19]. However, PRISM limits the development team to using the provided SMOs, which, as highlighted by Herrmann et al. [20], is not a complete coverage. Moreover, the development team must manually audit any changes performed on the database through these SMOs since an SMO can result in many DML and DDL operations on the database. Finally, the development team must manually write these SMOs, and PRISM does not consider additional database constructs which were added by the client (e.g., views and functions).

In addition, numerous commercial solutions [10]–[12] exist to manage schema evolution with Oracle [13], IBM [14] and Microsoft [15] each having their own tools to handle schema changes on their own database management systems (DBMS). However, the development team must still manually log the queries executed on the database as the 'diff' [21] comparison-based functionality provided by these tools does not consider the changes that happened in between two schema versions in a similar notion to performing a diff between two text files.

TABLE I. A SUBSET OF SCHEMA CHANGES WHICH TARGET A TABLE

Granularity	Description
Add Table	
Table Level	With Indexes
	Without Indexes
Attribute Level	Without Constraints
	With Constraints
Purge Table	
-	Drop Table
-	Covert table to many tables, e.g., 1-to-Many relation
-	Purge table and move its data to another table
Amend Table	
Table Level	Add/Alter/Delete Comment
	Add/Alter/Delete Indexes
	Enable/Disable Triggers
Attribute Level	Change Datatype
	Add/Alter/Delete Constraints

Given two database schemas, the schema diff algorithm returns the SQL statements that must be executed to synchronize both schemas. However, this algorithm does not consider the steps taken to arrive at the target point. For example, consider the schema change presented in Figure 1. The algorithm would generate the SQL operations found in Figure 2. These operations would result in the loss of all user addresses since the 'Address' column is dropped without copying its data to the new table. Moreover, the schema diff algorithm links the 'AddressID' column in the destination schema to the 'AddressID' column of the 'Address' table found in the source schema being compared, i.e., `sch_b`, rather than using the newly generated 'Address' table in the destination schema, i.e., `sch_a`, which does not represent the developer's original intention.

Another technique to perform schema evolution is temporal versioning which stores the entire schema for each schema change that occurs. This requires more annotations but has powerful version reasoning constructs that allow the DBA to move from one version to another and it also allows for a few concurrent schema versions, e.g., possibly depicting different functionalities.

A delicate part of the management of schema changes is their recording, and this is mostly developer input. A tenable method to automate the recording of schema changes is to trigger actions on DDL constructs, as is done with DML operations. In PostgreSQL, this can be achieved through event triggers [22]. Event triggers can be fired either before

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

Copyright (c) IARIA, 2025. ISBN: 978-1-68558-265-4

or after a DDL event has occurred and they can also be triggered by specific DDL events, e.g., only on delete operations. This mechanism enables the automated recording of schema changes without requiring any further explicit action from the developer.

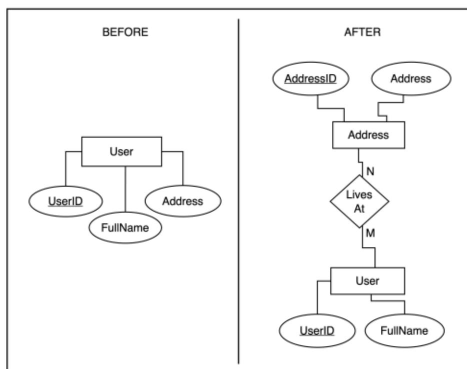

Figure 1. A schema change with data loss when using a schema diff algorithm.

```
CREATE TABLE IF NOT EXISTS sch_a."Address"
(   "AddressID" text NOT NULL PRIMARY KEY,
"Address" text NOT NULL );

ALTER TABLE sch_a."User"
DROP COLUMN IF EXISTS "Address";

ALTER TABLE sch_a."User"
ADD COLUMN "AddressID" text NOT NULL;

ALTER TABLE sch_a."User"
ADD CONSTRAINT "Users_AddressID_fkey"
FOREIGN KEY("AddressID")
REFERENCES sch_b."Address" ("AddressID");
```

Figure 2. The SQL queries generated by a schema diff algorithm (found within pgAdmin [23] version 8) for the schema change in Figure 1.

Software vendors can distribute code changes over-the-air, e.g., through package managers or Docker [24], or through manual techniques [25], e.g., file transfer. When run at the client, the database upgrade script generated by the vendor must do the necessary pre-checks (to ensure that none of the changes affect the client's data), schema updates, and post-checks (to ensure that no data was lost during the upgrade process). If any pre-checks fail, the database is not compatible with the upgrades defined by the vendor and, if any post-checks fail, this implies that the schema changes resulted in an inconsistent database state and the old state needs to be restored.

III. REQUIREMENTS & DESIGN

A software vendor that develops database-centric application programs requires a system that automatically transcribes schema changes over the database in an upgrade script. This script must then be packaged and applied to the client's database when the application program is upgraded at the client. The software vendor also requires that the schema change has its respective undo action sequence in case the upgrade needs to be reverted. Finally, apart from creating the upgrade and undo script, the system must also ensure that these changes were performed correctly on the client's database. If the client's database instance loses consistency, any data is lost from the client's database, or the update is not what was expected, then the client's database needs to be recovered to the state before the process started.

The software vendor is accepting that the system is based on a single DBMS, e.g., PostgreSQL, and that a reasonable subset of SQL's DDL and DML operations are available. Furthermore, the system must maximize the facilities provided by the DBMS, i.e., programming interfaces and data dictionaries. Moreover, the script generated is to have adequate security profile requirements and should be executed efficiently on client-side set-ups. Finaly, data consistency, and availability need to be catered for as well.

Schema changes can be captured in a few ways. One approach is to use a rule-based system, based on triggers and event triggers, to capture the DML and DDL queries generated by the data designer. This method attaches DDL triggers to the schemas that need to be tracked and DML triggers to the tables that need to have their data changes tracked. Once a trigger action is fired, it stores the difference in metadata between the current version and the proposed change, thus encoding schema evolution [16] through trigger action and the underlying data dictionary.

Another approach is to use a schema diff algorithm. This algorithm could be used in one of two ways – one can either compare the initial schema with the final schema, after all the changes have been done, or one can apply the schema diff algorithm incrementally after each schema change. The former provides a succinct upgrade script but loses the actions that happened in between the major versions while the latter essentially generates the SQL query which was input by the data designer.

When comparing the two approaches, it was decided to adopt a rule-based system for AURORA. Such a decision was taken as, to execute the schema diff algorithm incrementally after each schema change, a system of DDL and DML triggers is still required. Moreover, as discussed in Section 2, the queries generated by the schema diff algorithm are not adequate as these may lead to erroneous schema upgrades.

PostgreSQL has DDL triggers and makes use of two data dictionaries, the SQL `information_schema` and the Postgres-specific `pg_catalogue`. AURORA uses the `pg_catalogue` as this allows it to interpret implementation-specific queries that the database designer may run.

Moreover, the `pg_catalogue` allows the system to reference objects using a robust naming scheme across the instance, i.e., the OID. Having a robust naming scheme is crucial since it allows the system to accurately track an object throughout its lifespan, from creation to deletion. One must note that database objects can be renamed, and they may also be dropped and re-created using the same name. Therefore, the object's schema-qualified name, which the `information_schema` uses to identify objects, is not adequate to identify an object throughout its entire lifespan.

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

IV. IMPLEMENTATION

AURORA was implemented and tested on PostgreSQL 15. At the software vendor, the application's database includes a dedicated `version_control` schema which includes all the tables and functions which are needed to track the DML and DDL changes performed on that database. DDL changes are tracked using event triggers while DML changes are tracked using triggers which are automatically attached to the version-controlled tables.

As depicted in Figure 3, when the database designer enables the system, a snapshot of all the tracked objects in the database is taken. Then, whenever a DML or a DDL query is executed, a snapshot of the data dictionary entry of the effected object(s) is re-taken. Each time a snapshot is taken, a new version of the database is said to have been generated.

When doing DDL operations, AURORA uses the OID provided by the event trigger to get the object that was directly modified; however, it also checks each database object individually to determine if any other object has been modified as a side-effect of the DDL operation. This is done by generating and comparing the SHA1 hash of each object's metadata as found in the data dictionary and in the object's latest snapshot in the system. If the hashes don't match, then the object has been modified, and a snapshot of the object is taken. If the hashes match, then the operation does not need to be recorded.

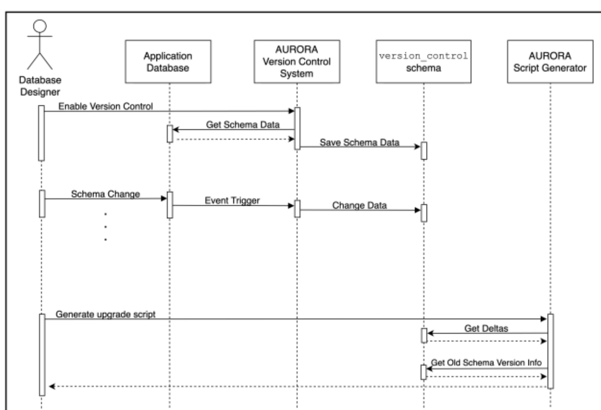

Figure 3. A UML Sequence Diagram depicting how AURORA is used by the developers.

With regards to DML operations, AURORA keeps a snapshot of the old and new version of the modified record(s) in JSON format. This allows all DML changes to be stored at the same table, even though each table has its own set of attributes.

Once the database designer performs all the changes to the schema, they can then use the provided Python program (referred to as the 'script generator') to generate the SQL upgrade script from the snapshots captured. The DBA at the client can then use another Python program (referred to as the 'script executor') to execute the SQL upgrade script on their database instance.

Given a specific range of versions, the script generator gets all the snapshots in that range in JSON encoding, and it compares each snapshot with the last snapshot of that object using a JSON 'diff' algorithm. Depending on the attributes that have changed between the snapshots, it then tries to generate an SQL query like the one that was executed by the data designer. This program also generates the respective undo queries to reverse the operation. This allows the DBA at the client to reverse the upgrade without needing to restore an entire database backup since this extends downtime.

Once the upgrade script is created, the script generator then generates a list of database objects that the script executor should find in the client's database instance. This list along with the SQL upgrade script and the script executor program are provided to the client to allow them to upgrade their database instance to the latest version.

The script executor is split up into three parts: the pre-checks, the execution of the SQL scripts and the post-checks. As part of its pre-checks, the script executor uses the list of objects generated by the script generator to ensure that the state of the client's database is as expected; otherwise, the SQL upgrade script will not be compatible with the database and might cause unexpected results.

If additional constructs are found in the database, e.g., views and functions that are created by the client, and they do not depend on any object that will be modified by the generated SQL upgrade script, they are left untouched. However, if the pre-checks determine that these objects will be affected by the upgrade script, the process is stopped and the user is informed that these objects need to be maintained for the script to run.

Before running the upgrade script, another pre-check is performed to ensure that any new integrity constraints do not affect any of the client's data in the database. If this is the case, the client's DBA is asked to remove the violating data from the appropriate table.

Once all the pre-checks are done, the script executor gets the number of records in all the tables and a hash for each record is generated. This information will then be used at the end of the upgrade to ensure that no data loss occurred (apart from that which was expected).

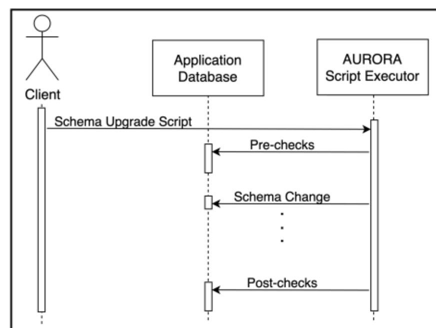

Figure 4. A UML Sequence Diagram depicting how AURORA is used on the client-side.

Before starting the upgrade itself, the script executor generates a backup of the entire database. This is done to ensure that no data is lost if something goes wrong and the

Copyright (c) IARIA, 2025. ISBN: 978-1-68558-265-4

database needs to be restored. Once the backup is generated, the script executor starts running the upgrade script on the client's database. As each query is being executed, an audit is kept of which queries were run. This allows the script executor to know which queries have been run and which reverse queries to execute if the upgrade fails. Once the upgrade is done, the script executor ensures that no data loss occurred based on the data recorded before the database was upgraded. This process can be seen in Figure 4.

To make it easier to reverse an upgrade, the script executor puts a 'tombstone' on any objects that need to be deleted. In this way, if a drop operation needs to be reversed, the tombstone is simply removed and the object, and all its data, is once again accessible.

AURORA's code was thoroughly tested with standard methods to ensure that it works as expected. Numerous DML and DDL queries with different variations were tested to ensure that the triggers were working as expected and to ensure that the system was correctly logging the objects' changes from the data dictionary. The script generator was tested by generating a set of JSONs, which represented both valid and invalid schema changes, and ensuring that the correct output, i.e., an SQL forward and backward query or error, is provided by the program. The script executor was tested by ensuring that any additional constructs generated by the client are detected, that any missing database objects are detected, that the script can adequately undo its actions if an error occurs, and that any unexpected data loss is detected.

V. Evaluation

AURORA was evaluated by simulating how it would be used in real-world scenarios. This was done by performing a set of schema changes on one database to simulate the changes performed at the vendor and then running the resulting script on another database to simulate the client's context. The goal of this evaluation was to ensure that:

1. The client's database ended up in the same state as that of the vendor, i.e., the vendor's DML and DDL operations were correctly recorded and executed;
2. All the client's data, in the unchanged schema objects, is preserved;
3. If the client's database has additional constructs which were created by the client (and which do not depend on the objects modified by the vendor), these constructs are preserved after the upgrade.

This evaluation was designed to cover as much of the schema changes that are tracked by AURORA. This was achieved by running the DML and DDL operations needed to generate the Scott schema [26], by going from version 7d4ca07595c6 [27] to 193312356621 [28] of the WikiMedia database and by upgrading a custom-made database [29].

The Scott schema was used to ensure that AURORA can handle a basic set of CREATE TABLE, ALTER TABLE and INSERT commands. The WikiMedia schema was used to test AURORA with a real-world database upgrade. Finally, a custom-made database schema was used to evaluate AURORA's performance on the remaining database objects that were not considered with the previous cases.

VI. Conclusion

Version Control plays a crucial role in software development; however, its support in databases is lacking as, although numerous systems exist to perform some form of version control on a database, schema changes need to be manually recorded by the development team. This increases the risk of missing key operations while creating the database upgrade script. Therefore, this paper introduced AURORA which automatically tracks the DDL and DML operations performed on a database. Using this tracked data, the system can automatically generate a database upgrade script along with a set of undo queries to rollback unacceptable upgrades. The script generated is then given to the client to apply the changes to their own database, reducing downtime while ensuring data integrity throughout the upgrade process.

AURORA was implemented on PostgreSQL, and it has been shown to reliably handle schema evolutions. This system's techniques offer a significant advantage when compared to techniques based on a schema diff algorithm since this system offers a finer granularity of changes. It also represents a significant step forward in database version control as it has the ability to automate schema evolution while minimizing downtime and preserving data consistency, making it a valuable tool for both developers and clients. As a result, AURORA can effectively improve the workflow and the quality of software deployments for both the software house and its client.

Unlike Git, which is a distributed and decentralized version control system [6], AURORA only works on a centralized database. If decentralized database support is required, a causal consistency [30][31] mechanism is indicated as it ensures a partial ordering between a schema change and any operation that depends on it while providing reasonable performance.

AURORA does not modify the DDL and DML operations that were executed by the software provider. As a result, there is the possibility that the upgrade script includes redundant DDL and/or DML operations. For example, if the data designer creates a table, deletes it and re-creates it with the same name (without doing any other actions to the table), the first two create and delete operations can be omitted from the script as their net effect is nil. Future work involves optimizing the upgrade script to address such redundant processes.

Lastly, this system integrates with previous works such as PRISM. By modifying AURORA to automatically generate the Schema Modification Operations based on the DML or DDL operations performed, one would be able to take advantage of PRISM's features, such as its query re-writing facilities. Further query re-writing facilities can also be provided to re-write SQL specific constructs, e.g., the SQL query in a subset of Common Table Expressions (CTEs). However, even if such query re-writing facilities were provided in AURORA, the queries (to be re-written) would have to be provided explicitly by the developer since automatically identifying the SQL queries in the application's codebase would require a full scan of the entire

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

Copyright (c) IARIA, 2025. ISBN: 978-1-68558-265-4

source code and would require numerous heuristics, e.g., to differentiate a string which is holding an SQL query against a string which is holding a user prompt. Moreover, the application may generate SQL queries during runtime, meaning that the application's codebase would only have partially written SQL queries, which cannot be re-written.

REFERENCES

[1] C. A. Curino, H. J. Moon, L. Tanca, and C. Zaniolo, "Schema Evolution in Wikipedia - Toward a Web Information System Benchmark," in *ICEIS2008 - Proc. 10th Int. Conf. Enterprise Inf. Syst.*, SciTePress, 2008, pp. 323–332, doi: 10.5220/0001713003230332.

[2] C. A. Curino, H. J. Moon, A. Deutsch, and C. Zaniolo, "Update rewriting and integrity constraint maintenance in a schema evolution support system: PRISM++," *Proc. VLDB Endow.*, vol. 4, no. 2, pp. 117–128, Nov. 2010, doi: 10.14778/1921071.1921078.

[3] D.-Y. Lin and I. Neamtiu, "Collateral evolution of applications and databases," in *Proc. Joint Int. and Annu. ERCIM Workshops Principles Softw. Evolution (IWPSE) and Softw. Evolution (Evol) Workshops*, in IWPSE-Evol '09. New York, NY, USA: Association for Computing Machinery, 2009, pp. 31–40, doi: 10.1145/1595808.1595817.

[4] D. Lucia. "Revolut app issues — 30th October. What happened, and what we did to fix it," https://web.archive.org/web/20200812131530/https://blog.revolut.com/revolut-app-issues-30th-october-what-happened-and-what-we-did-to-fix-it/ (accessed Feb. 04, 2025).

[5] M. J. Rochkind, "The source code control system," *IEEE Transactions on Software Engineering*, vol. SE-1, no. 4, pp. 364–370, 1975, doi: 10.1109/TSE.1975.6312866.

[6] "Git Portal," Git. https://git-scm.com/ (accessed Oct. 07, 2024).

[7] "OneDrive Portal," https://www.microsoft.com/en-us/microsoft-365/onedrive/online-cloud-storage (accessed Feb. 08, 2025).

[8] "Google Drive Portal," Google. https://workspace.google.com/products/drive/ (accessed Feb. 08, 2025).

[9] "Dropbox Portal," Dropbox. https://www.dropbox.com (accessed Feb. 08, 2025).

[10] "Atlas Portal," Atlas. https://atlasgo.io/ (accessed Dec. 17, 2024).

[11] "Liquibase Portal," Liquibase. https://www.liquibase.com/ (accessed Dec. 17, 2024).

[12] "Redgate Flyway Community Portal," Redgate. https://www.red-gate.com/products/flyway/community/ (accessed Dec. 17, 2024).

[13] "Enterprise Manager Lifecycle Management Administrator's Guide," Oracle Help Center. https://docs.oracle.com/cd/E24628_01/em.121/e27046/change_management.htm#EMLCM11767 (accessed Dec. 17, 2024).

[14] "Db2 Object Comparison Tool for z/OS 13.1.0 Documentation," IBM Documentation. https://www.ibm.com/docs/en/db2objectcompare/13.1?topic=131-overview (accessed Dec. 17, 2024).

[15] "SQL Server Data Tools Documentation," Microsoft Learn. https://learn.microsoft.com/en-us/sql/ssdt/sql-server-data-tools?view=sql-server-ver16 (accessed Feb. 06, 2025).

[16] J. F. Roddick, "A survey of schema versioning issues for database systems," *Information and Softw. Technol.*, vol. 37, no. 7, pp. 383–393, 1995, doi: https://doi.org/10.1016/0950-5849(95)91494-K.

[17] T. Spiteri Staines and J. G. Vella, "High level architectural modelling for representing the extract, transform and load process," in *Int. Conf. Inf. Syst. and Manage. Sci. (ISMS 2018)*, Valletta, Malta, Mar. 2018, pp. 1–10,

[18] C. Camilleri, J. G. Vella, and V. Nezval, "HTAP With Reactive Streaming ETL," *J. Cases Inf. Technol.*, vol. 23, no. 4, pp. 1–19, 2021, doi: 10.4018/JCIT.20211001.oa10.

[19] C. A. Curino, H. J. Moon, and C. Zaniolo, "Graceful database schema evolution: the PRISM workbench," *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 761–772, Aug. 2008, doi: 10.14778/1453856.1453939.

[20] K. Herrmann, H. Voigt, J. Rausch, A. Behrend, and W. Lehner, "Robust and simple database evolution," *Inf. Syst. Front.*, vol. 20, no. 1, pp. 45–61, Feb. 2018, doi: 10.1007/s10796-016-9730-2.

[21] J. W. Hunt and M. D. Mcilroy, "An algorithm for differential file comparison," *Computer Science*, 1975, Available: http://www.cs.dartmouth.edu/%7Edoug/diff.pdf. [Accessed: Feb. 08, 2024]

[22] "Overview of Event Trigger Behavior," PostgreSQL Documentation. https://www.postgresql.org/docs/16/event-trigger-definition.html (accessed Jul. 04, 2024).

[23] "pgAdmin Portal," pgAdmin. https://www.pgadmin.org/ (accessed Feb. 11, 2025).

[24] "Docker Portal," Docker. https://www.docker.com/ (accessed Feb. 11, 2025).

[25] J. Humble and D. Farley, "The Problem of Delivering Software," in *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, Boston, MA, USA: Addison-Wesley Professional, 2010, pp. 5–7.

[26] "SCOTT Schema," Pastebin. https://pastebin.com/NcMVAfpL (accessed Feb. 11, 2025).

[27] "Merge 'resourceloader: Fix line indent in FileModuleTest'," https://phabricator.wikimedia.org/rMW7d4ca07595c67f4fbe9ac5fbce9ce9837cc2cd70 (accessed Feb. 09, 2025).

[28] "Merge 'schema: Add cl_target_id and cl_collation_id to categorylinks'," Phabricator. https://phabricator.wikimedia.org/rMW193312356621f82996a0351715f7d8074405f53f#change-CgIXeHctgWRI (accessed Feb. 09, 2025).

[29] "Custom Upgrade Script," Pastebin. https://pastebin.com/3SvthrP3 (accessed Feb. 19, 2025).

[30] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto, "Causal memory: definitions, implementation, and programming," *Distrib. Comput.*, vol. 9, no. 1, pp. 37–49, Mar. 1995, doi: 10.1007/BF01784241.

[31] C. Camilleri, J. G. Vella, and V. Nezval, "D-Thespis: A Distributed Actor-Based Causally Consistent DBMS," in *Transactions on Large-Scale Data- and Knowledge-Centered Systems LIII*, A. Hameurlain and A. M. Tjoa, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2023, pp. 126–165.

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org