



Una estrategia genérica de evolución de bases de datos

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:

Antonio Pérez Serrano

Tutor/es:

Jesus Joaquín García Molina

José Ramón Hoyos Barceló

29 de Mayo de 2025



**Facultad
Informática
Universidad
Murcia**

Una estrategia genérica de evolución de bases de datos

OrionFlow: Una herramienta CI para bases de datos

Autor

Antonio Pérez Serrano

Tutor/es

Jesus Joaquín García Molina

Departamento Informática y Sistemas

José Ramón Hoyos Barceló

Departamento Informática y Sistemas



Grado en Ingeniería Informática



DIS
Departamento de
Informática y Sistemas

UNIVERSIDAD DE
MURCIA



Murcia, 29 de Mayo de 2025

Preámbulo

El presente Trabajo Fin de Grado nace de la necesidad de abordar de forma sistemática la evolución de esquemas en entornos de persistencia heterogénea, con el propósito de unificar y automatizar las migraciones de modelos relacionales y NoSQL. Su alcance incluye tanto la extensión al modelo relacional de un lenguaje de operaciones de evolución (Orion), así como su integración en flujos DevOps mediante GitHub Actions y análisis estático con CodeQL.

Durante el desarrollo de este trabajo se han llevado a cabo tareas de generación de transformaciones (módulos M2T para MySQL y CodeQL), pruebas de funcionamiento de cada operación y la creación de un caso de estudio propio (Umugram). En todo momento hemos contado con la colaboración y el asesoramiento de los tutores del proyecto y del resto de miembros del grupo de investigación ModelUM, cuyos debates y revisiones han sido fundamentales para garantizar los principios y fundamentos de la propuesta.

Agradecimientos. Este trabajo es parte del proyecto PID2020-117391GB-I00 financiado por el Ministerio Español de Ciencia, Innovación y Universidades y la Agencia Estatal de Investigación (MICIU/AEI/10.13039/501100011033) y ERDF, EU.

Agradecimientos

Desde el verano pasado mi vida académica —y, en gran medida, personal— sufrió un giro inesperado. Llegué al Grado desmotivado, incapaz de hallar energía ni sentido a lo que hacía; deseaba que los días del calendario pasasen sin más. Por mucho que trabajara, una voz interna —alimentada por unas calificaciones que parecían recordarme a diario mis carencias— insistía en que nunca era suficiente, encerrándome en un bucle de inseguridad y agotamiento emocional.

Entonces apareció **Fabián**. Me propuso formar equipo y, aunque no entendía por qué me había elegido, acepté. Trabajando codo con codo vimos que, juntos, superábamos cualquier barrera: el talento se multiplica cuando se comparte y las metas, por altas que parezcan, dejan de estar fuera de alcance. Mis notas empezaron a remontar y, sobre todo, recuperé las ganas de aquel chaval que un día soñó con comerse el mundo. A este impulso se sumó **Jesús Joaquín García Molina**, que me ofreció colaborar con el grupo de investigación MODELUM.

Me gustaría decir gracias,

A **Jesús**, le debo la oportunidad de unirme al grupo, su confianza y la paciencia con la que me ha guiado. Ha sido el empuje definitivo para volver a creer en mis capacidades.

A **Fabián**, gracias por ser el motor diario que me anima a dar siempre una mejor versión de mí mismo. Las horas de código, café y debates me han recordado lo divertido que es programar acompañado.

A **José Ramón**, cuyo criterio claro y acompañamiento este verano han sido esenciales para ganar experiencia en las tecnologías en las que este Trabajo de Fin de Grado está basado.

Al equipo de MODELUM, por acogerme desde el primer día, por los comentarios constructivos y por demostrar que la investigación es, ante todo, un trabajo colectivo.

Por último y no menos importante, gracias a mi familia y a mis amigos de siempre. Su apoyo incondicional ha sido el suelo firme sobre el que he reconstruido mi motivación.

A quienes confiaron en mí incluso cuando yo dudaba: este trabajo es para vosotros.

*La vida no se trata de encontrarte a ti mismo,
sino de crearte a ti mismo.*

Bernard Shaw.

Declaración firmada sobre originalidad del trabajo

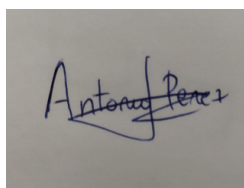
D./Dña. **Antonio Pérez Serrano**, con DNI **24468696P**, estudiante de la titulación de **Grado en Ingeniería Informática** de la Universidad de Murcia y autor del TF titulado “**Una estrategia genérica de evolución de bases de datos**”.

De acuerdo con el Reglamento por el que se regulan los Trabajos Fin de Grado y de Fin de Máster en la Universidad de Murcia (aprobado C. de Gob. 30-04-2015, modificado 22-04-2016, 28-09-2018 y 28-06-2022), así como la normativa interna para la oferta, asignación, elaboración y defensa de los Trabajos Fin de Grado y Fin de Máster de las titulaciones impartidas en la Facultad de Informática de la Universidad de Murcia (aprobada en Junta de Facultad 27-11-2015)

DECLARO:

Que el Trabajo Fin de Grado presentado para su evaluación es original y de elaboración personal. Todas las fuentes utilizadas han sido debidamente citadas. Así mismo, declara que no incumple ningún contrato de confidencialidad, ni viola ningún derecho de propiedad intelectual e industrial.

Murcia, a 29 de Mayo de 2025

A handwritten signature in blue ink, reading "Antonio Pérez Serrano", is centered within a square gray box.

Fdo.: Antonio Pérez Serrano
Autor del TF

Resumen

En el desarrollo de software, la evolución de esquemas de bases de datos es una necesidad constante derivada de cambios en el negocio, la aparición de nuevas tecnologías o mejoras funcionales en las aplicaciones. Sin embargo, estos cambios en el esquema de la base de datos, pueden generar errores si no se gestionan adecuadamente, tanto en el código fuente de las aplicaciones que interactúan con ella como en las pruebas del sistema que utilizan los desarrolladores.

Con las prácticas de integración continua (CI), ha surgido la tendencia de gestionar de manera más rigurosa estos cambios. Herramientas como Flyway o Liquibase permiten registrar y versionar scripts de migración, integrándolos en el flujo de desarrollo. Sin embargo, estas herramientas se enfocan en bases de datos relacionales y su soporte para bases de datos NoSQL es escaso. Además, no contemplan la adaptación del código fuente que se ve afectado por los cambios.

La complejidad se incrementa cuando se combinan varias tecnologías de bases de datos (relacionales, documentales, grafos, etc.), lo que impide generalizar los cambios en el esquema. Esta problemática muestra la necesidad de herramientas que permitan manejar la evolución de esquemas de forma unificada, independientemente de la tecnología utilizada.

Para enfrentar estos problemas, se ha llevado a cabo este Trabajo fin de Grado (TFG), que busca minimizar esos riesgos automatizando tanto la generación de los cambios como la detección de su impacto en el código. El TFG se enmarca dentro de un proyecto de investigación del grupo ModelUM financiado por el Ministerio de Ciencia, Innovación y Universidades, y que busca conseguir un enfoque ágil para la evolución de esquemas de bases de datos, con soporte para migración de datos y código.

En los últimos años, el grupo de investigación ModelUM ha centrado su trabajo en la definición de lenguajes y herramientas genéricas para tareas comunes en sistemas de bases de datos, aplicando Ingeniería del Software Dirigida por Modelos (MDE). Para ello, ha creado el metamodelo unificado U-Schema que permite representar tanto los modelos de esquemas de bases de datos relacionales como de los principales modelos NoSQL (documentos, columnares, key-value y grafos), permitiendo abstraerse de las particularidades de cada tecnología subyacente.

Por otro lado ha definido dos lenguajes específicos de dominio (DSL): uno llamado Athena, que es el encargado de definir los esquemas utilizando una notación textual expresiva apoyándose en el metamodelo unificado U-Schema, y otro llamado Orion, que permite definir los cambios en el esquema de forma agnóstica a la base de datos.

Este TFG está enfocado en la creación de una herramienta llamada OrionFlow,

que permita automatizar la adaptación del esquema de la base de datos y facilite la adaptación del código fuente de las aplicaciones que se verá afectado por este cambio de esquema. La herramienta utilizará como entrada un script de Orion en el que se especifican los cambios que se quieren llevar a cabo en el esquema de la base de datos.

En el trabajo realizado se ha extendido el lenguaje Orion, para que pueda generar automáticamente scripts MySQL que den soporte a los cambios en el esquema definidos con el lenguaje. Junto con la transformación del esquema, se ha incorporado un sistema de análisis que permite revisar automáticamente el código fuente y detectar fragmentos que podrían verse afectados por los cambios. Este análisis se realiza utilizando reglas definidas en un lenguaje de consultas sobre la semántica del código llamado codeQL, y se integra con herramientas populares como Github en los flujos de trabajo actuales. De este modo, cada vez que se propone un cambio en el esquema, se evalúa automáticamente su posible impacto sobre el código.

Toda la solución se ha integrado en un flujo de trabajo basado en GitHub Actions, lo que permite realizar las comprobaciones y aplicar los cambios desde el propio repositorio del proyecto. Así, los desarrolladores pueden proponer transformaciones, recibir alertas si algo falla y actualizar los elementos necesarios para actualizar la base de datos.

El sistema se ha validado mediante un caso práctico llamado Umugram, elaborado con las tecnologías Spring Boot y Spring Data JPA donde existe un fuerte acoplamiento entre los datos y el código fuente. Para las diferentes pruebas, se generaron las consultas codeQL donde se detectaron fragmentos del código que debían revisarse. También se ha desarrollado una herramienta de versionado para gestionar los cambios, lo que facilita el mantenimiento del sistema a lo largo del tiempo.

Como posibles líneas de trabajo futuras, se plantea ampliar el soporte a más tipos de bases de datos, mejorar las capacidades del análisis del código para cubrir más patrones, y explorar mecanismos que permitan sugerir correcciones automáticas. En conjunto, esta propuesta contribuye a avanzar hacia sistemas más robustos y confiables en lo que respecta a la evolución de esquemas de datos en entornos de desarrollo continuo.

Extended Abstract

This Final Degree Project (TFG) focuses on creating a practical and reliable solution for managing changes to database structures in software applications. In today's fast-paced software development landscape, applications are constantly evolving. Developers add new features to enhance user experiences, businesses adjust their strategies to remain competitive, and technical teams optimize systems to handle growing data volumes or improve performance. Each of these changes often requires updates to how data is stored—whether in traditional relational tables, document-based collections, or graph-based systems. These updates, known as schema evolution, are a critical but challenging aspect of software development. Without proper management, they can lead to errors, data inconsistencies, and significant delays in development.

When database changes are handled informally—through manually written scripts or one-off fixes applied directly in production—problems are almost inevitable. The application code may fail to find the data it expects, leading to runtime errors. Data relationships can break, compromising the integrity of the system. Perhaps most concerning, these issues introduce uncertainty about the reliability and quality of the data, which can erode trust in the application. Beyond technical problems, these challenges impact development teams, who must dedicate significant time to diagnosing issues, rewriting queries, and recovering lost or corrupted data. This not only affects the application's performance but also reduces overall productivity, as skilled developers are pulled away from building new features to fix preventable errors.

Modern software development relies heavily on continuous integration and continuous deployment pipelines, which automate and streamline the process of updating and deploying code. Ideally, database schema changes should be part of these pipelines, treated with the same care as code updates. Tools like Flyway and Liquibase attempt to address this by storing database change scripts alongside code in version control systems like Git. This approach improves traceability and ensures that the database structure aligns with the application's needs. However, these tools have significant limitations: they primarily support relational databases and offer little assistance for updating the application code affected by schema changes. Additionally, as organizations increasingly adopt a mix of database technologies—such as relational databases like MySQL and NoSQL systems like MongoDB or Neo4j—managing schema changes across these diverse systems becomes a complex task, as each database type follows its own rules and syntax.

To tackle these challenges, this project builds on the innovative work of the ModelUM research group, which has developed a framework called U-Schema. This framework

provides a unified way to describe database structures, whether they are relational or NoSQL, making it easier to manage schema changes across different technologies. The group has also created two specific languages: Athena, which allows developers to define database structures in a clear, textbased format, and Orion, which enables high-level descriptions of schema changes without tying developers to the specific syntax of a particular database. This project extends Orion to support relational databases, specifically MySQL, and integrates it into modern CI/CD workflows, ensuring that schema changes are handled efficiently and reliably.

Schema evolution is a routine task in software development, but it's fraught with challenges. When the structure of a database changes—say, a table is renamed, a column is removed, or a new relationship is added—the application code that relies on the database must be updated to match. If these updates are not carefully coordinated, the application may fail to find the expected data, resulting in errors during runtime. For example, a query looking for a renamed column will fail, or a missing table could cause the application to crash. These mismatches can also break data integrity rules, such as foreign key constraints, leading to inconsistent or unreliable data. Over time, these issues can undermine confidence in the system, especially in critical applications where data accuracy is paramount.

Historically, schema evolution has been managed through ad-hoc methods, such as writing custom scripts or making direct changes in production environments. These approaches are error-prone and difficult to track, especially in large projects with frequent updates. Developers may forget to update related code, or manual changes may introduce subtle bugs that are hard to detect until they cause significant problems. In modern development environments, where teams push updates multiple times a day, these manual methods are unsustainable. They lead to wasted time, as developers must troubleshoot errors, rewrite broken queries, and restore corrupted data, all of which divert resources from building new features or improving the application.

The rise of CI/CD pipelines has transformed software development by automating code testing, building, and deployment. However, database schema changes are often left out of these automated workflows. Tools like Flyway and Liquibase help by managing database scripts in version control, but they focus almost exclusively on relational databases and provide little support for updating the application code affected by schema changes. This is a significant gap, as schema changes often ripple through the codebase, requiring updates to queries, data access logic, or object mappings. Moreover, as organizations adopt diverse database technologies—combining relational databases with NoSQL systems like document stores, key-value stores, or graph databases—the complexity of schema management increases. Each database type has its own syntax and rules, making it difficult to create a unified approach to schema evolution.

This project proposes a comprehensive solution for managing database schema evolution, making it easier to update database structures while keeping the application code in sync. The solution builds on the ModelUM group's U-Schema framework, which offers a technology-agnostic way to describe database structures. Whether the database

is a relational system like MySQL or a NoSQL system like MongoDB, Cassandra, or Neo4j, U-Schema provides a consistent representation that abstracts away the technical details. This unified approach simplifies schema management across diverse database types, a critical need in modern software architectures where hybrid setups are common.

At the heart of the solution are two domain-specific languages developed by ModelUM: Athena and Orion. Athena allows developers to define a database's structure in a simple, text-based format that works across different database technologies. For example, a developer can describe a table or collection without worrying about the specific syntax of MySQL or MongoDB. Orion, meanwhile, enables developers to specify changes to the database structure at a high level. Instead of writing low-level SQL or NoSQL commands, developers can use Orion to write commands like “add entity,” “remove attribute,” or “rename relationship.” These commands are independent of the target database, making it possible to apply the same change logic to different systems.

The Orion system works by validating each change against the current database structure, ensuring it's valid before applying it. It updates an in-memory model of the database and generates the specific code needed to execute the change—such as SQL statements for MySQL, aggregation pipelines for MongoDB, or Cypher queries for Neo4j. Until now, Orion primarily supported NoSQL databases, which left relational databases—a mainstay of many industries—unsupported. This project addresses that limitation by extending Orion to generate SQL scripts for MySQL, ensuring it can handle both relational and NoSQL environments. This extension makes the solution more versatile, as it can now support the hybrid database setups commonly found in enterprise applications.

The solution also integrates schema evolution into CI/CD pipelines, ensuring that changes are thoroughly tested and validated before reaching production. Using GitHub Actions to orchestrate tasks and CodeQL to analyze code, the system identifies parts of the application that might break due to schema changes. For example, if a column is renamed, CodeQL can flag queries or code that still reference the old name. This proactive approach reduces the risk of errors in production and helps developers address issues early in the development process.

A key feature of this project is its integration of schema evolution into modern CI/CD workflows. By embedding database change management into the same pipelines that handle code updates, the solution ensures that schema changes are treated with the same rigor as code changes. This is achieved using GitHub Actions, which automates tasks like running tests or deploying code, and CodeQL, a powerful tool for static code analysis. Together, these tools help catch potential issues before they reach production, making the development process more reliable and efficient.

The Orion engine, provided by the ModelUM group, is central to this integration. It starts with an Athena file describing the current database structure and an Orion script listing the desired changes. These are processed in two steps. First, a model-to-model transformation updates the internal representation of the database, producing

a new Athena file that reflects the changes. Second, a model-to-text transformation generates the scripts needed to apply the changes to the actual database. For MySQL, this means creating SQL scripts to add tables, modify columns, or update relationships. The system ensures that these scripts are precise and follow the correct order to avoid errors.

To maintain data integrity, the solution enforces strict rules for schema changes. For example, every table must have a primary key, and identifiers are checked against MySQL's reserved words to prevent conflicts. Complex changes, such as those involving multiple steps or data integrity checks, are implemented using stored procedures. These procedures make the changes reusable, allow dynamic queries to manage constraints, and ensure that if an error occurs, the database is rolled back to a consistent state. This approach minimizes the risk of partial or inconsistent updates, which could otherwise lead to data corruption.

The integration with CI/CD pipelines is achieved through two synchronized workflows. The first workflow uses CodeQL to analyze the impact of Orion scripts on the application code. When a developer submits an Orion script, the system clones the project, identifies the script, and generates CodeQL queries to check for potential issues, such as outdated queries or object mappings. The results are displayed in GitHub's Code Scanning section, with color-coded alerts showing which parts of the code need attention. Developers can then resolve, ignore, or defer these alerts based on their context.

The second workflow manages the versioning of Orion scripts. It activates when a commit message starts with the keyword "orion" and specifies whether to accept or reject a change. If accepted, the script is archived in a dedicated directory, renamed for clarity, and logged in a Markdown-based version history table with details like the version number, description, timestamp, and commit hash. If rejected, the script is removed from the repository. This process uses empty commits to avoid unnecessary code changes while maintaining a clear record of decisions, ensuring traceability and accountability.

To validate the solution, a prototype application called Umugram was developed—a simplified social network built with Java 17, Spring Boot 3, and MySQL 8. Umugram was created using Spring Initializr, which provided the basic project structure, dependencies (including Spring Data JPA), and configuration. The application's data model centers on a User entity with attributes like email, username, password, and profile information (e.g., biography, avatar, and website). Users can follow each other, creating self-referential relationships, and post content like text, photos, videos, or live streams. These posts are linked to a Comment entity, allowing for rich interactions.

The persistence layer was implemented using Spring Data JPA's JpaRepository interfaces, which simplify common database operations like creating, reading, updating, and deleting data. For more complex queries, a custom repository was created using JPA's persistence context and native query support. This setup allowed the application to handle both standard and advanced data access patterns, making it a realistic

testbed for schema evolution.

Multiple test scenarios were conducted to evaluate the solution. For example, the team renamed or removed entities, such as changing a table's name or deleting a column. In each case, the CodeQL-based analysis accurately identified affected code, such as queries or mappings that referenced the old structure. The OrionMySQL generator was also tested by creating the Umugram database from an Orion script, verifying that the generated SQL scripts correctly set up tables, relationships, and constraints. These tests confirmed that the solution could handle real-world schema changes while maintaining alignment between the database and the application code.

The solution provides a strong foundation for managing schema evolution, but there are several opportunities for enhancement:

- **Enhanced Versioning:** Improve the version history table to support undo and redo operations, allowing developers to roll back or reapply changes as needed. This would make the system more flexible for iterative development.
 - **Legacy Compatibility:** Adapt CodeQL queries to support older versions of Java's persistence framework (javax.persistence), ensuring the solution works with legacy systems still in use by many organizations.
 - **Advanced Code Analysis:** Incorporate data flow analysis into CodeQL, enabling the system to track how values move through the code and identify issues in dynamically built queries. This would improve the accuracy of impact detection.
 - **Automated Suggestions:** Develop a feature that suggests fixes for code affected by schema changes, such as updating a query or modifying an annotation. This would reduce the manual effort required and speed up the development process.
-

Índice general

Resumen	xiii
Extended Abstract	xv
Lista de Acrónimos y Abreviaturas	xxix
1. Introducción	1
1.1. Contexto y Motivación	1
1.2. Objetivo	3
1.3. Metodología	5
1.4. Organización de la memoria	6
2. Fundamentos	9
2.1. Ingeniería Dirigida por Modelos (MDE)	9
2.2. U-Schema	10
2.3. Athena	11
2.3.1. Orion	13
2.3.2. CodeQL	15
2.4. GitHub Actions	16
2.5. Spring	17
3. Estado del arte	19
3.1. Herramientas existentes de evolución de esquemas CI	19
3.2. Trabajo de investigación relacionado	20
4. Visión general de la solución propuesta	23
5. Actualización del esquema y de los datos	27
5.1. Mapping SCOs Orion a cambios en esquema relacional	27
5.2. Implementación	31
5.2.1. Arquitectura General	31
5.2.2. Flujo de ejecución	32
5.2.3. Decisiones de diseño	33
6. Actualización del código	37
6.1. Workflow: CodeQL	37

6.2. Workflow: Migration Schema	39
7. Validación	43
7.1. Caso de estudio: Umugram	43
7.2. Resultados obtenidos en Umugram	45
7.2.1. Generación en MySQL de Umugram con Orion	45
7.2.2. Integración continua en Umugram	48
8. Conclusiones y líneas futuras	53
8.1. Conclusiones Generales	53
8.2. Limitaciones del enfoque y trabajo futuro	53
8.2.1. Limitaciones de la transformación con Orion	53
8.2.2. Trabajo futuro sobre la herramienta predictora	54
Bibliografía	55
A. Anexo I	59
A.1. Flyway	59
A.2. Liquibase	60
B. Anexo II	63
B.1. Plantillas de texto en Xtend (<i>String Templates</i>)	63
B.2. Ejemplo de operacion <code>dispatch</code>	64
B.3. Procedimientos auxiliares	65
C. Anexo III	79
C.1. Scripts orion	79
C.2. Código MySQL generado	83

Índice de figuras

2.1. Metamodelo U-Schema	10
2.2. Metamodelo Athena	13
2.3. Taxonomía de SCO de Orion, tomado de [1]	14
4.1. Arquitectura del motor Orion con integración continua y ampliación del Data Updater.	23
4.2. Funcionamiento interno de los workflows que conforman la integración continua.	25
5.1. Diagrama de clases UML para el DataUpdater relacional	31
6.1. Alerta en la sección de code-scanning de Github.	38
6.2. Visión integral del workflow automatizado en Orion, combinando análisis CodeQL y control de versiones de scripts.	41
7.1. Diagrama de clases UML del caso de estudio Umugram	44
7.2. Scripts generados tras hacer uso del generador	45
7.3. Workflow codeql tras recibir el script orion	49
7.4. Alertas mostradas en la sección code-scanning de Github	50
7.5. Tabla de versiones vista desde Github tras aceptar diferentes scripts . .	51

Índice de tablas

1.1. Estimación de horas dedicadas a la elaboración del TFG	7
5.1. Scripts para la base de datos MySQL a partir de las operaciones de Orion	29
5.2. Procedimientos almacenados más relevantes en MySQL	35
6.1. Historial de versiones de scripts Orion en el fichero <code>version_table.md</code>	40
A.1. Ejemplo parcial de la tabla <code>flyway_schema_history</code>	60
A.2. Simulación de la tabla <code>DATABASECHANGELOG</code>	62

Índice de Códigos

2.1. Ejemplo de operación EntityAddOp en Orion	14
5.1. Método principal empleado en la generación de las operaciones a MySQL	32
7.1. Repositorio con extensión JpaRepository en <i>Umugram</i>	43
7.2. Repositorio con métodos del EntityManager en <i>Umugram</i>	44
7.3. Script Orion para añadir Post	46
7.4. SQL generado para MySQL de la entidad Post	46
7.5. Fragmento en memoria de Post en el esquema .athena	47
7.6. Operación promote del atributo revision en Post	47
7.7. Consulta CodeQL para localizar una consulta con anotación @Query .	50
A.1. Script de creación de tabla con Flyway	59
A.2. Ejemplo de <i>changelog</i> Liquibase	60
B.1. Ejemplo mínimo de plantilla	63
B.2. Iteración con FOR para tipos colección	64
B.3. Generación de SQL a partir de EntityRenameOp	64
B.4. Procedimientos generados por la transformación M2T Orion-MySQL .	65
C.1. Script 1 – Operaciones <i>Umugram</i>	79
C.2. Script 2 – Operaciones avanzadas <i>Umugram</i>	82
C.3. Script SQL generado para el script 1 C.1	83
C.4. Script SQL generado para el script 2 C.2	87

Lista de Acrónimos y Abreviaturas

CRUD	Create, Read, Update, Delete.
DBCL	DATABASECHANGELOG.
DSL	Domain-Specific Language.
EMF	Eclipse Modeling Framework.
M2T	Model-to-Text.
MDE	Model-Driven Engineering.

1. Introducción

1.1. Contexto y Motivación

En el ámbito del desarrollo de software, los esquemas de bases de datos rara vez permanecen estáticos. Esto se debe a la necesidad de las organizaciones de adaptarse a las nuevas tecnologías, transformaciones en su modelo de negocio o la incorporación de mejoras funcionales y técnicas. Como resultado, se produce una evolución del esquema de la base de datos, que suele implicar la adición, eliminación o modificación de entidades, propiedades y relaciones. Estos cambios tienen un impacto directo en el resto del sistema, incluyendo los datos, el código fuente, en particular las consultas de acceso a los datos o las validaciones y pruebas.

Tradicionalmente, a pesar de ser una tarea común, la evolución del esquema de base de datos no siempre se ha realizado de forma sistemática ni se ha integrado adecuadamente con el resto del desarrollo. Con frecuencia, se creaban nuevas tablas manteniendo las anteriores, lo cual podía acarrear problemas como consultas obsoletas o fragmentos de código huérfanos que dependen de elementos eliminados del modelo de datos.

Con la aparición de los entornos actuales de integración continua y despliegue [2], se observa una tendencia creciente a aplicar buenas prácticas en la evolución del esquema, de modo que, además del código fuente, los cambios al esquema se integran dentro del propio flujo de desarrollo, por ejemplo, mediante el uso de repositorios para el control de versiones (como Github [3] o Gitlab [4]). Es el caso de herramientas como Flyway [5] o Liquibase [6] que, a la vez que se guarda una versión con los cambios que se producen en el código fuente del proyecto, almacenan los scripts necesarios para la modificación de la base de datos, permitiendo una trazabilidad en la evolución del esquema. Este enfoque busca mantener la coherencia entre el modelo de datos, la implementación y el despliegue, ante los inevitables cambios que se producen a lo largo del ciclo de vida del software. Todavía, la adopción de estas herramientas no está muy extendida, no ofrecen facilidades para la adaptación del código, y la mayoría sólo soportan bases de datos relacionales pero no sistemas NoSQL [7] (en algunos casos ofrecen un soporte muy limitado de sistemas NoSQL, como es el caso de Flyway para MongoDB).

La complejidad a la que se enfrentan los desarrolladores se incrementa cuando la organización utiliza múltiples tecnologías de bases de datos (relacional, documental, grafos, etc...), ya que cada modelo de datos tiene sus propias reglas de estructuración y operaciones, lo que dificulta la generalización de los cambios y obliga a un mayor esfuerzo de integración y mantenimiento. Con la creciente adopción de las bases de

datos NoSQL y el papel predominante que seguirán teniendo las bases de datos relacionales, se hace evidente la conveniencia de disponer de herramientas que faciliten abordar la evolución de una forma genérica, de modo que se pueda abordar la evolución del esquema independientemente de la tecnología subyacente, por ejemplo, expresar las operaciones de cambio de una forma independiente del modelo de datos.

En los últimos años, el grupo de investigación ModelUM ¹ ha centrado su trabajo en la definición de lenguajes y herramientas genéricas para tareas comunes en sistemas relacionales aplicando *Ingeniería del Software Dirigida por Modelos* (MDE, por sus siglas en inglés) [8]. Para ello, se ha creado el metamodelo unificado U-Schema que puede representar esquemas de los modelos relacionales y los modelos NoSQL (documentos, columnares, key-value y grafos) [9]. Alrededor de U-Schema, se han abordado varios proyectos, como la definición de esquemas con el lenguaje Athena [10], la definición de datos sintéticos con el lenguaje Deimos [11], y la evolución del esquema [12]. El trabajo realizado en este *Trabajo Fin de Grado* (TFG) se enmarca en el último proyecto. Una taxonomía de operaciones de cambio de esquemas (SCO, en inglés *Schema Changes Operations*) junto al lenguaje Orion que la implementa fueron presentados en [12], donde se describe cómo el motor de Orion actualiza el esquema y los datos para los tres sistemas NoSQL más populares ² : MongoDB (documentos), Cassandra (columnar) y Neo4J (grafos), que corresponden a diferentes modelos de datos NoSQL. El motor de Orion recibe como entrada un esquema U-Schema y un script Orion, los convierte en los correspondientes modelos conformes a los metamodelos U-Schema y Orion, respectivamente, y aplica dos transformaciones de modelos: una transformación modelo-a-modelo (m2m) para actualizar el esquema y una transformación modelo-a-texto (m2t) para generar un script de código que adapta la base de datos al nuevo esquema. Estos scripts incluyen operaciones en las APIs de MongoDB o Neo4J, o el lenguaje de consultas de Cassandra (CQL).

Dado que las bases de datos relacionales siguen siendo claramente dominantes, la extensión de Orion para ofrecer su soporte era necesaria. Junto a esta extensión, la construcción de una herramienta que aprovechara Orion para proporcionar una evolución ágil, similar a la que ofrecen Flyway o Liquibase pero facilitando la adaptación de código, era una tarea pendiente en un proyecto I+D financiado por el Ministerio de Ciencia, Innovación y Universidades ³.

Disponer de esas dos capacidades es esencial para equipos de desarrollo que trabajan con frameworks de aplicaciones que usan bases de datos relacionales, como Spring Boot [13] que usa anotaciones JPA [14] (Java Persistence API) y consultas JPQL (Java Persistence Query Language). En estos contextos, cualquier modificación en el esquema puede derivar en errores en tiempo de compilación o, incluso de manera más problemática, en tiempo de ejecución. Dichos errores pueden manifestarse en consultas

¹<https://modelum.github.io/index.html>

²como se puede observar en <https://db-engines.com/en/ranking>

³PID2020-117391GB-I00, Proyectos I+D de Generación de Conocimiento, Un enfoque ágil para la evolución de esquemas de bases de datos NoSQL: migración de datos y código

inválidas o mal tipadas, en clases que referencian entidades inexistentes o en fallos de validaciones internas de Spring Data JPA [15] que no encuentran ciertos campos o relaciones. Además, en escenarios complejos, algunos errores pueden ocultar a otros, dificultando la depuración y ralentizando el desarrollo.

Esta fragilidad del código ante cambios en el modelo de datos expone la necesidad de herramientas que no solo gestionen la adaptación del código en la evolución del esquema, sino que también sean capaces de predecir y localizar automáticamente el impacto que estos cambios tendrán en el código fuente. Hasta donde alcanza nuestro conocimiento, este tipo de análisis predictivo no está contemplado en los entornos de desarrollo convencionales ni en las herramientas existentes de evolución ágil del esquema. Los desarrolladores deben confiar en su conocimiento del sistema, en herramientas de validación genéricas como SonarQube [16] o en la ejecución de tests para detectar inconsistencias. Esto puede resultar costoso y propenso a errores. Una estrategia de adaptación de código para Orion fue presentada en [17] pero que solo llegaba a mostrar cómo el lenguaje de consultas de código CodeQL [18] era una buena opción para encontrar los puntos afectados por cambios del esquema expresados con Orion, pero no se llegó a abordar el diseño y construcción de una estrategia CI que contemplase la adaptación tanto de los datos como del código.

1.2. Objetivo

El trabajo de este TFG surge de los dos objetivos mencionados arriba para el proyecto I+D de evolución ágil de esquemas relacionales y NoSQL: la extensión de Orion para soportar sistemas relacionales y el diseño e implementación de una estrategia de evolución ágil. La herramienta creada, que hemos denominado *OrionFlow*, debe permitir la adaptación de datos y código cada vez que se desee ejecutar un script Orion. Cada vez que se realice un *commit* que incluya un script Orion, se generarán consultas CodeQL que analizarán el código fuente para detectar de forma automática y anticipada las partes del código de las aplicaciones que se verán afectadas si se produce un determinado cambio en el esquema. Además, como se ha comentado previamente, existe un creciente interés por automatizar la evolución del esquema dentro de los pipelines de CI, por lo que OrionFlow será parte de un proceso CI en GitHub aprovechando la posibilidad de definir workflows de automatización de tareas con GitHub Actions [19]. Gracias al análisis previo, el desarrollador podrá anticipar el alcance de los cambios necesarios en el código y decidir si la nueva versión del esquema es viable antes de su despliegue.

Se ha escogido GitHub como plataforma de control de versiones y GitHub Actions como su sistema de integración continua, por su popularidad en la industria y amplia adopción por equipos de desarrollo. Y también hay que destacar que CodeQL está integrado de forma nativa en GitHub.

La validación de OrionFlow se ha llevado a cabo por medio de una aplicación muy

sencilla, que hemos denominado UmuGram desarrollada con Spring Boot y Spring Data JPA.

Para poder alcanzar el objetivo, el trabajo realizado se ha organizado en tres tareas principales que se han abordado en el orden en que se muestran:

- **Ampliar el DSL Orion:** Permitir que Orion admita bases de datos relacionales y pueda generar scripts de migración para MySQL [20].
- **Detección de impacto en el código:** Diseñar y generar consultas CodeQL a partir del script Orion de entrada que localicen en el código fuente de la aplicación las entidades, propiedades o relaciones afectadas por cada operación de evolución.
- **Integración continua:** Dirigir todo el proceso CI en un workflow de GitHub Actions que, ante cada commit con un script `.orion`, genere y analice las consultas a partir del script y genere alertas que avisen al desarrollador. Según la acción tomada por el desarrollador, administrar su decisión a partir de otro workflow que lleve a cabo el control de versiones sobre el esquema afectado.

De forma más concreta, los objetivos específicos del trabajo son:

- Diseñar un flujo automatizado (*workflow*) en GitHub que reciba como entrada un script de evolución Orion, compuesto por una secuencia de operaciones sobre el esquema Athena.
 - Para cada operación del script, generar automáticamente consultas CodeQL específicas que permitan identificar advertencias y errores en el código fuente asociado (consultas JPQL, anotaciones JPA, clases afectadas, etc.) en caso de realizar dichos cambios.
 - Mostrar los resultados del análisis en formato estandarizado (*SARIF*), resaltando el código que deberá ser modificado si se desea aplicar la evolución del esquema.
 - En caso de aprobación del cambio, generar un nuevo commit con el nombre del script, y registrar su versión en una tabla de versionado que contabilice las evoluciones realizadas, siguiendo un enfoque similar al utilizado por herramientas como Flyway.
 - Desde la herramienta de escritorio que realiza transformaciones modelo-a-texto M2T (Orion-Bases de datos), adaptar el motor Orion para que sea capaz de generar scripts de migración MySQL. El desarrollador tendrá la elección de transformar el código afectado y regenerar el esquema de la base de datos o orientarse por el código generado en la transformación para realizar la migración.
-

En definitiva, el objetivo central del TFG es proporcionar a los desarrolladores un mecanismo automatizado para anticipar el impacto que tendría una modificación del esquema sobre el código fuente, permitiéndoles valorar si el número y la criticidad de las zonas afectadas justifican aplicar el cambio. Tras su evaluación, podrá optar por aceptar la migración —con la opción de modificar los datos y el código en los próximos cambios— o bien rechazarla y descartarla por completo.

1.3. Metodología

Es preciso señalar que la complejidad y objetivos del trabajo han exigido un esfuerzo superior al que se supone a un TFG y se ha realizado a lo largo de unos 9 meses, con diferente intensidad, en el marco del proyecto I+D antes mencionado, y parte de la dedicación estuvo cubierta con un contrato de técnico-especialista de unos 2 meses de duración. En realidad, el trabajo ha abarcado más tareas que las relacionadas con el TFG explicadas en esta Memoria.

EL trabajo requería de conocimientos que se pueden dividir en cuatro áreas: MDE, los lenguajes Athena y Orion junto al metamodelo U-Schema, bases de datos NoSQL, CodeQL, Spring Boot y Github Actions. Por ello, lo primero fue adquirir una base sólida de MDE: metamodelado, la plataforma EMF (Eclipse Modeling Framework) y el lenguaje utilizado para escribir transformaciones Xtend. Todo esta formación fue recibida en varios seminarios impartidos por el tutor Jesús J. García Molina. Después, fue preciso la lectura de algunos trabajos de investigación publicados por parte del grupo ModelUM, en especial el artículo de Alberto Hernández-Chillón et al. [17], así como las tesis doctorales de Alberto Hernández Chillón [1] y Carlos J. Fernández Candel [21]. La primera para conocer los DSL Orion y Athena en profundidad, y la segunda para conocer bien el metamodelo unificado U-Schema. Durante esta etapa inicial, se tomó la decisión de orientar el proyecto hacia esquemas relacionales implementados con el framework *Spring Data JPA* y el conocimiento de esta plataforma se consiguió fundamentalmente con un texto de introducción [15]. Y en este punto se realizó una primera incursión en CodeQL a partir de la documentación oficial proporcionada por GitHub.

A partir de ese estudio inicial, el trabajo realizado se puede organizar en las tres tareas principales que abajo se detallan, cada una con varias actividades, y se indica el orden en que se llevaron a cabo entre paréntesis.

- *Diseño*: Diseño de la adaptación de los datos (1) y de la estructura de las consultas CodeQL para cada operación de la taxonomía de Orion (2). Definición de los flujos de trabajo y su integración con Github Actions. Diseño del caso de uso Umugram para las pruebas (4).
- *Implementación*: Desarrollo de la transformación m2t que genera código SQL para la adaptación de los datos a partir de scripts Orion (3) y creación del pipeline automatizado que ejecuta las consultas y gestiona el versionado en GitHub (7).

- *Validación*: Desarrollo del caso de estudio Umugram (5), donde se aplicaron diversas operaciones de evolución en el esquema para analizar sus efectos en el código, permitiendo evaluar el funcionamiento de la herramienta OrionFlow. Llevar a cabo la validación de la transformación m2t (6) y de los workflows CI (8).

Durante todo el desarrollo, se produjeron reuniones presenciales o virtuales con los tutores con una frecuencia quincenal (a veces, semanal). Además, se mantuvo un canal de comunicación constante mediante WhatsApp y correo electrónico, lo que permitió resolver dudas de forma casi inmediata y mantener un feedback continuo y enriquecedor durante todo el trabajo.

La dedicación total del trabajo se estima en 500 horas, distribuidas según la planificación mostrada en la Tabla 1.1.

1.4. Organización de la memoria

El Capítulo 2 recoge los fundamentos teóricos y tecnológicos necesarios para comprender el trabajo desarrollado que se explica en el resto de la Memoria. En él, se introducen los conceptos clave MDE, el metamodelo U-Schema, los DSLs Orion y Athena, así como las tecnologías utilizadas: CodeQL y GitHub Actions, entre otras.

El Capítulo 3 presenta el trabajo relacionado en el ámbito de la evolución de esquemas, comparando nuestro enfoque con herramientas existentes y trabajos de investigación previos relevantes.

En los capítulos 4, 5 y 6 se describe la solución propuesta. El primero aporta una descripción general que se va detallando en los dos siguientes capítulos, donde se incluye la adaptación de las operaciones de evolución de Orion al contexto relacional, la construcción de consultas de análisis estático para detectar código afectado y el diseño de los flujos de trabajo para lograr la integración continua.

En el Capítulo 7 se aplica el enfoque desarrollado a un caso de estudio, la aplicación Umugram. Se describe el esquema inicial y se prueban las dos herramientas desarrolladas: la transformación a scripts MySQL y el funcionamiento de la herramienta predictiva. Finalmente, el Capítulo 8 expone las conclusiones obtenidas, se plantean posibles limitaciones del enfoque propuesto y el trabajo futuro relacionado con un posible diseño de la segunda versión de la herramienta OrionFlow.

Actividad	Descripción	Duración
Reuniones con el grupo ModelUM	Encuentros presenciales y virtuales para seguimiento, resolución de dudas y toma de decisiones técnicas.	47 h
Estudio de MDE y EMF	Aprendizaje de fundamentos teóricos, metamodelado, uso de Ecore, Xtend y lectura de papers del grupo ModelUM.	75 h
Formación en Spring Data JPA	Lectura del libro <i>Java Persistence with Spring Data and Hibernate</i> y experimentación con anotaciones JPA.	40 h
Estudio de CodeQL y GitHub Actions	Revisión de documentación oficial y experimentación con análisis estático y flujos de CI/CD.	70 h
Diseño del flujo de trabajo	Definición de la arquitectura de la solución, operaciones Orion, integración de herramientas, SARIF.	49 h
Implementación de la herramienta de escritorio	Desarrollo de la transformación M2T de Orion a scripts SQL para MySQL.	78 h
Desarrollo del workflow de GitHub	Implementación de las consultas CodeQL, integración del análisis y versionado en GitHub Actions.	56 h
Caso de estudio Umu-gram	Creación y evolución del modelo, aplicación de operaciones, análisis de impacto y documentación de resultados.	30 h
Redacción de la memoria	Escritura, revisión y estructuración del documento final del TFG.	55 h
Total		500 h

Tabla 1.1: Estimación de horas dedicadas a la elaboración del TFG

2. Fundamentos

2.1. Ingeniería Dirigida por Modelos (MDE)

La Ingeniería Dirigida por Modelos (Model-Driven Engineering, MDE) es un paradigma de desarrollo de software que promueve el uso de modelos como elementos centrales durante el proceso de desarrollo. Su objetivo principal es elevar el nivel de abstracción, permitiendo a los desarrolladores centrarse en los aspectos funcionales y conceptuales del sistema, mientras se minimiza la necesidad de manejar directamente detalles técnicos o de implementación mediante código.

En el contexto de la MDE, un modelo no se limita a ser una mera representación conceptual del sistema, sino que se convierte en un artefacto ejecutable a partir del cual es posible generar automáticamente código, validar restricciones, simular comportamientos o derivar configuraciones específicas del sistema.

El fundamento teórico de MDE se apoya en el concepto de metamodelado. Un meta-modelo es un modelo que define los elementos y relaciones válidas dentro de un lenguaje de modelado. Esta jerarquía se estructura comúnmente en una arquitectura de cuatro niveles:

- Nivel M0: El mundo real o los datos concretos del sistema.
- Nivel M1: Los modelos que representan dichos datos (por ejemplo, un modelo UML de clases).
- Nivel M2: El metamodelo que define los elementos posibles del lenguaje de modelado (por ejemplo, Ecore).
- Nivel M3: El meta-metamodelo, que define el lenguaje con el que se construyen los metamodelos (como MOF o Ecore en sí mismo).

En el desarrollo de lenguajes específicos de dominio (DSLs), los metamodelos juegan un papel esencial al definir la sintaxis abstracta del lenguaje. Esta definición especifica los conceptos estructurales del dominio —como clases, atributos, asociaciones o generalizaciones— y suele representarse mediante diagramas de clases. Para complementar la semántica de los modelos y garantizar su validez, es habitual utilizar lenguajes de restricciones como OCL (Object Constraint Language), que permiten expresar reglas de formación y condiciones que deben cumplirse dentro del modelo.

El entorno Eclipse, a través del proyecto EMF (Eclipse Modeling Framework), proporciona una de las infraestructuras más completas para el desarrollo MDE. EMF

incluye herramientas para la creación de metamodelos (Ecore), la definición de sintaxis concreta (textual o gráfica) y el desarrollo de transformaciones. Existen tres tipos de transformaciones: modelo a modelo (M2M), modelo a texto (M2T) y texto a modelo (T2M).

En el contexto de este TFG, se emplea MDE como tecnología de implementación dado que el lenguaje Orion está creado como un DSL que implementa la sintaxis abstracta como un metamodelo Ecore/EMF [22], a su vez, está basado en el metamodelo U-Schema. En este trabajo, se han implementado transformaciones modelo a código para la adaptación del esquema, los datos y el código.

2.2. U-Schema

U-Schema es un metamodelo utilizado para representar de forma unificada esquemas de bases de datos relacionales y no relacionales (documentos, columnares, key-value y grafo) permitiendo abstraerse de las particularidades de cada modelo de datos subyacente, que fue desarrollado en la tesis de Carlos J. Fernández Candel [21]. A diferencia de los modelos de datos específicos, U-Schema ofrece una visión lógica unificada que facilita la creación de soluciones genéricas para manejar bases de datos.

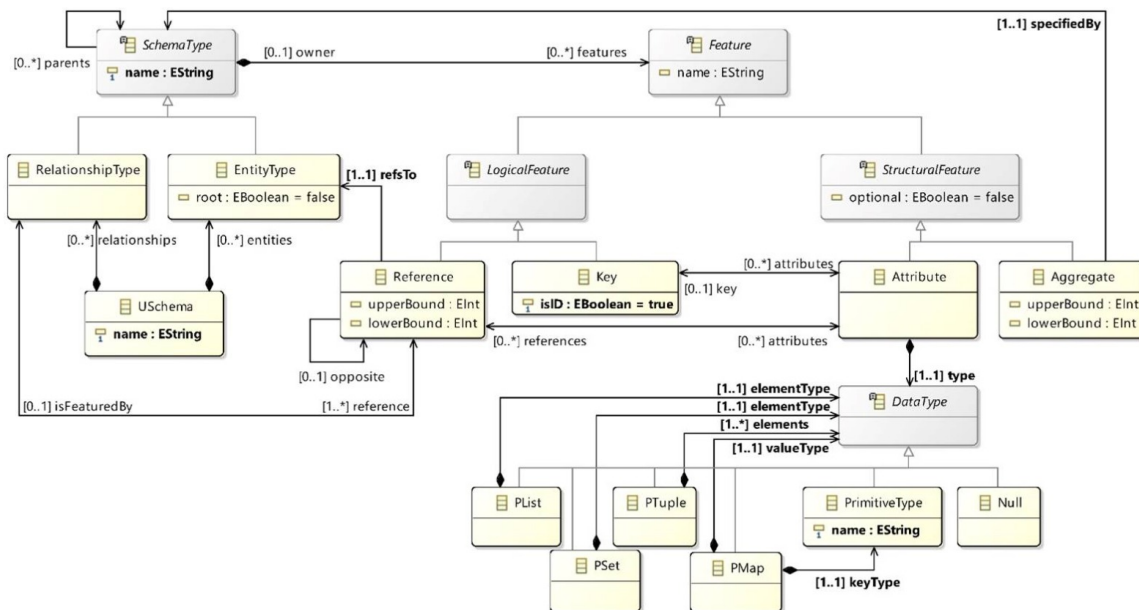


Figura 2.1: Metamodelo U-Schema

En la figura 2.1 podemos observar el metamodelo U-Schema creado como un metamodelo Ecore/EMF. Este define una serie de elementos que desglosaremos a continuación:

- **USchema:** Representa el modelo completo del esquema de datos.

- **SchemaType**: Cada USchema está compuesto por múltiples *SchemaTypes*, los cuales pueden clasificarse como *EntityType* o *RelationshipType*.
- **EntityType**: Corresponde a una entidad del dominio de la aplicación.
- **RelationshipType**: Representa relaciones entre entidades. Estas relaciones pueden tener atributos propios, y modelan tanto relaciones M:N en bases de datos relacionales como aristas en grafos.
- **Feature**: Cada *SchemaType* se compone de un conjunto de propiedades llamadas *Features*, que pueden ser de dos tipos: *LogicalFeature* y *StructuralFeature*.
- **LogicalFeature**: Incluye elementos como *Key* y *Reference*, que permiten definir claves y referencias a otras entidades respectivamente.
- **StructuralFeature**: Agrupa propiedades de tipo *Attribute* y *Aggregate*.
- **Attribute**: Es una propiedad con un tipo de dato asociado. Puede ser un tipo primitivo (por ejemplo, **String**) o un tipo compuesto como **PList**, **PSet**, **PTuple** o **PMap**.
- **Aggregate**: Define una entidad contenida dentro de otra, anidando una subentidad especificada por un *EntityType*. La propiedad **root** indica si se trata de una entidad principal o embebida. Además, las propiedades **lowerBound** y **upperBound** definen su cardinalidad.
- **Key**: Permite identificar de forma única a una entidad. Una clave puede estar compuesta por uno o más atributos, y la propiedad **isID** permite marcarla como identificador principal.
- **Reference**: Indica una relación hacia otra entidad mediante la propiedad **refsTo**. Puede especificar su cardinalidad a través de los atributos **lowerBound** y **upperBound**, y también puede ser definida a través de un *RelationshipType*.

2.3. Athena

Athena es un lenguaje genérico para especificar esquemas de bases de datos, el cual esta basado en el metamodelo U-Schema y fue desarrollado como parte de la tesis de Alberto Hernández Chillón [1]. El lenguaje Athena se ha diseñado para ser fácil de usar por desarrolladores con conocimientos de modelado de datos (por ejemplo, ER o OO) y favorecer la concisión y legibilidad.

Athena está construido con el framework *Xtext*, que permiten definir su gramática y generar automáticamente un editor textual y el inyector de modelo: combinación de un parse que genera el árbol sintáctico abstracto (AST) con un generador que lo recorre para crear el modelo Ecore que conforma al metamodelo del lenguaje.

En el contexto de este TFG, se utiliza Athena para definir los esquemas relacionales que serán sujetos a un proceso de evolución con Orion, y transformados automáticamente en scripts SQL adaptados a MySQL. Gracias a su sintaxis compacta, Athena facilita también el versionado, la edición manual y el análisis de los cambios estructurales a nivel de modelo.

Como se observa en la Figura 2.2, Athena define un conjunto de elementos clave que permiten modelar esquemas complejos con variaciones estructurales y relaciones entre entidades:

- **AthenaSchema:** Es el elemento raíz del modelo. Contiene un conjunto de *EntityType* y *RelationshipType*, así como posibles referencias a otros modelos mediante la propiedad `imports`.
 - **EntityType y RelationshipType:** Representan entidades y relaciones respectivamente. Las entidades pueden ser principales (`root=true`) o embebidas, mientras que las relaciones permiten modelar vínculos entre entidades con sus propios atributos.
 - **StructuralVariation:** Representa una variante de una entidad o relación. Cada variación contiene un conjunto de *Feature* que pueden diferir entre instancias de un mismo tipo.
 - **Feature:** Son propiedades que describen los datos de una variación. Se dividen en *Key*, *Reference*, *Attribute* y *Aggregate*, representando diferentes formas de estructurar la información.
 - **Key:** Permite identificar una instancia de entidad de forma única. Puede estar compuesta por múltiples atributos y marcarse como identificador principal mediante la propiedad `isID`.
 - **Reference:** Define una relación hacia otra entidad. Incluye cardinalidades mediante `lowerBound` y `upperBound`, y puede estar vinculada a una relación declarada explícitamente.
 - **Aggregate:** Define una entidad embebida dentro de otra. Hace referencia a otra *EntityType* mediante la propiedad `specifiedBy`, y permite establecer su cardinalidad.
 - **Attribute:** Propiedad con un tipo de dato asociado. Puede ser primitivo (`PrimitiveType`) o compuesto, incluyendo estructuras como `PList`, `PSet`, `PTuple` o `PMap`.
 - **DataType:** Es la superclase de todos los tipos de datos válidos en Athena. Abarca desde tipos básicos hasta estructuras complejas como listas, mapas o tuplas.
-

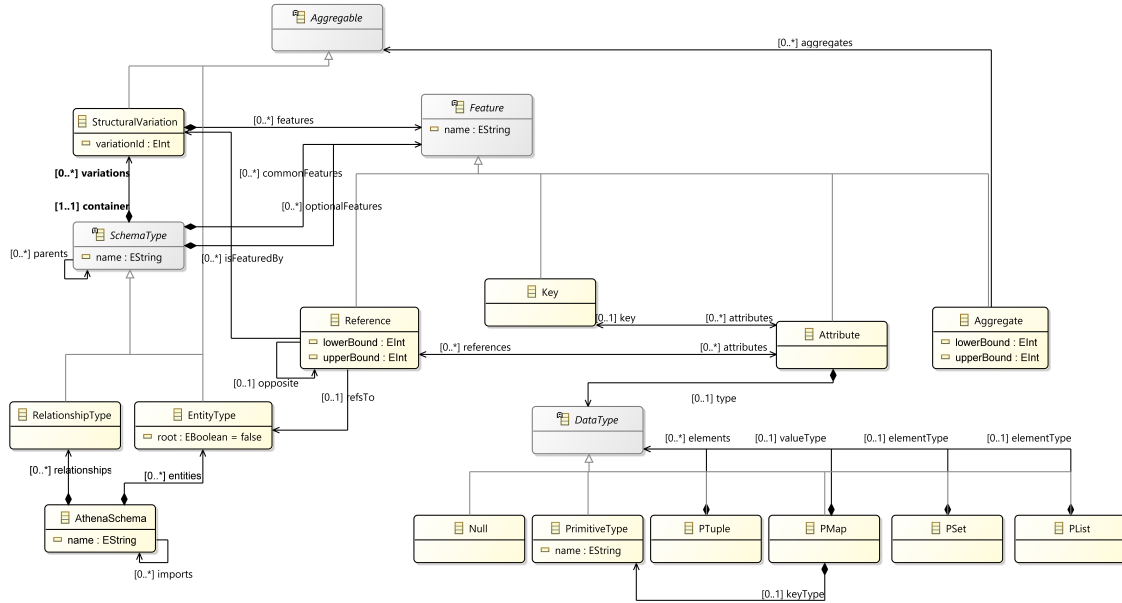


Figura 2.2: Metamodelo Athena

2.3.1. Orion

Orion es un lenguaje genérico para especificar operaciones de cambio (SCO) que fue desarrollado en la tesis de Alberto Hernández. Este lenguaje se basa en una taxonomía de cambios de esquema definida sobre el metamodelo U-Schema. En la figura 2.3 se puede observar el conjunto de SCOs de la taxonomía extraído de la tesis de Alberto Hernández [1]. Estas operaciones están encapsuladas en un DSL propio, similar a lo visto con el DSL Athena, también desarrollado con tecnologías como Xtext y OCL, garantizando su validación.

Uno de los aspectos clave de este trabajo es que Orion no solo se utilizará para transformar esquemas, sino también como base para analizar el impacto que estos cambios pueden tener en el código fuente de aplicaciones. A través de un flujo automatizado en Github, el sistema toma como entrada un script Orion y genera consultas específicas en CodeQL que permiten detectar fragmentos de código que quedarían obsoletos o inconsistentes tras aplicar los cambios.

El fragmento 2.1 ejemplifica varias operaciones definidas en la taxonomía de Orion. En primer lugar, se muestra una operación **EntityAddOp**, que añade una nueva entidad **User** al esquema. A continuación, se utiliza una operación **AggregateAddOp** para asociar a dicha entidad un agregado llamado **profile**, que encapsula información estructurada como nombre, avatar, descripción y sitio web. Posteriormente, se define la entidad **Post**, también mediante **EntityAddOp**, y finalmente se incorpora una operación **ReferenceAddOp** que establece una relación entre **Post** y **User** a través de una clave foránea `user_id`.

MongoDB		t_M	Cassandra	t_C	Neo4j	t_N
Entity type Operations						
Add	createCollection(),\$addFields	0.01	CREATE table	0.21	CREATE,MATCH,SET node	0.37
Delete	drop()	0.01	DROP table	0.37	MATCH,DELETE node	1.12
Rename	renameCollection()	0	2×(COPY,DROP,CREATE) table	2.86	MATCH,REMOVE,SET node	1.89
Extract	\$project,\$out	0.31	2×COPY table,CREATE table	2.25	MATCH,CREATE node	2.92
Split	2×(\$project,\$out),drop()	0.63	(4×COPY,2×CREATE,DROP) table	4.44	MATCH,2×CREATE,DELETE node	6.55
Merge	2×\$merge,2×drop()	4.95	(4×COPY,CREATE,2×DROP) table	4.72	2×MATCH,CREATE,2×DELETE node	8.29
Delvar	remove()	0.38	—	—	MATCH,DELETE node	1.42
Adapt	\$unset,\$addFields	0.70	—	—	MATCH,REMOVE,SET node	1.95
Union	\$addFields	1.40	—	—	MATCH,SET node	8.44
Relationship type Operations						
Add	—	—	—	—	—	—
Delete	—	—	—	—	MATCH,DELETE rel	2.43
Rename	—	—	—	—	MATCH,apoc.refactor.setType rel	0.22
Extract	—	—	—	—	MATCH,CREATE rel	5.43
Split	—	—	—	—	MATCH,2×CREATE rel	10.99
Merge	—	—	—	—	2×MATCH,CREATE,2×DELETE rel	13.84
Delvar	—	—	—	—	MATCH,DELETE rel	0.84
Adapt	—	—	—	—	MATCH,REMOVE,SET rel	0.76
Union	—	—	—	—	MATCH,SET rel	15.93
Feature Operations						
Delete	\$unset	1.08	DROP column	0.22	MATCH,REMOVE field	0.78
Rename	\$rename	1.22	2×COPY table,DROP column,ADD column	2.07	MATCH,SET,REMOVE field	2.03
Copy	\$lookup,\$addFields,\$addFields,\$out	4.06	2×COPY table,ADD column	2.21	2×MATCH,SET field	2.24
Move	\$lookup,\$addFields,\$addFields,\$out,\$unset	5.09	2×COPY table,ADD column,DROP column	2.30	2×MATCH,SET,REMOVE field	3.31
Nest	\$rename	1.27	—	—	—	—
Unnest	\$rename	1.30	—	—	—	—
Attribute Operations						
Add	\$addFields	1.35	ADD column	0.21	MATCH,SET field	0.76
Cast	\$set	1.31	2×(COPY,DROP,CREATE) table	3.06	MATCH,SET field	1.50
Promote	—	—	2×(COPY,DROP,CREATE) table	3.08	CREATE constraint UNIQUE	4.12
Demote	—	—	2×(COPY,DROP,CREATE) table	3.08	DROP constraint	0.03
Reference Operations						
Add	\$lookup,\$addFields,\$out	4.09	ADD column,2×COPY table	2.04	2×MATCH,CREATE rel	5.39
Cast	\$set	1.46	2×(COPY,DROP,CREATE) table	3.07	—	—
Mult	\$set	1.41	—	—	—	—
Morph	\$lookup,\$addFields,\$out,\$unset	4.95	—	—	—	—
Aggregate Operations						
Add	\$addFields	1.43	CREATE type,ADD column	0.24	—	—
Mult	\$set	1.45	—	—	—	—
Morph	insert(),save()	34.08	—	—	—	—

Figura 2.3: Taxonomía de SCO de Orion, tomado de [1]

Código 2.1: Ejemplo de operación EntityAddOp en Orion

```

1 ADD ENTITY User {
2   +id: Identifier,
3   name: String,
4   !email: String
5 }
6
7 ADD AGGR User::profile: {
8   nameProfile: String,
9   avatar_url: String,

```

```
10  description: String,  
11  website: String  
12 }&  
13  
14 ADD ENTITY Post: {  
15   +post_id: String,  
16   caption: String,  
17   description: String,  
18   created_at: Timestamp,  
19   updated_at: Timestamp  
20 }  
21  
22 ADD REF Post::user_id : String& TO User
```

2.3.2. CodeQL

CodeQL es una herramienta de análisis estático desarrollada por GitHub que permite representar el código fuente de una aplicación como una base de datos consultable. A través de un lenguaje de consultas, los desarrolladores pueden navegar por el árbol de sintaxis abstracta (AST) del programa y definir reglas que detecten patrones, vulnerabilidades o errores comunes. Este enfoque transforma el código en una colección de relaciones entre elementos (clases, métodos, expresiones, etc.), lo que facilita el análisis estructurado y permite automatizar la detección de inconsistencias de forma precisa. Gracias a esta capacidad, CodeQL resulta especialmente útil en tareas de verificación de calidad, revisión automática de código y análisis de impacto frente a cambios estructurales.

Los resultados del análisis se representan utilizando el formato *SARIF* (Static Analysis Results Interchange Format), un estándar promovido por la industria para reportar resultados de análisis estático. SARIF define un esquema JSON estandarizado que incluye:

- **Metadatos de la herramienta:** Identificador, versión y configuración empleada.
- **Catálogo de reglas:** Un array de *rules*, cada una con su ID, descripción, severidad y ayuda asociada.
- **Resultados del análisis:** Un array de *results*, cada entrada indicando:
 - Ubicación exacta en el código fuente (archivo, línea y columna).
 - Mensaje descriptivo del hallazgo.
 - Severidad o nivel de confianza.
 - Referencia a la regla infringida.

- **Runs:** Cada ejecución de análisis corresponde a un **run**, que agrupa sus propios metadatos, catálogo de reglas y resultados.

Gracias a esta estructura, los IDE y plataformas de integración continua pueden:

1. **Navegación contextual:** Al hacer clic en un resultado, saltar directamente a la línea afectada en el editor.
2. **Agrupación y filtrado:** Organizar hallazgos por regla, severidad o componente.
3. **Quality gates:** Definir condiciones de fallo en CI/CD (por ejemplo, bloquear builds si aparecen vulnerabilidades críticas).

GitHub Code Scanning, por ejemplo, consume directamente informes SARIF en cada push o pull request, marcando en la interfaz de revisión los problemas detectados por CodeQL y enlazando a la documentación de la regla correspondiente.

En el contexto de este trabajo, CodeQL se utiliza como mecanismo para analizar el impacto que las operaciones de evolución del esquema, definidas en Orion, pueden tener sobre el código fuente de aplicaciones desarrolladas con Spring Boot y Spring Data JPA. Para cada operación del script Orion, se generan dinámicamente consultas CodeQL adaptadas a la estructura y convenciones del proyecto objetivo. Los resultados se almacenan en formato SARIF y se envían al apartado de Github Code Scanning para mostrarlos como alertas.

2.4. GitHub Actions

GitHub Actions es la plataforma de integración y entrega continua (CI/CD) integrada en GitHub, que permite definir flujos de trabajo automatizados en respuesta a eventos como commits, pull requests o publicaciones de versiones. Mediante la definición de archivos YAML, los desarrolladores pueden configurar tareas que se ejecutan en entornos virtuales, facilitando la construcción, prueba, análisis o despliegue automático del software.

En este trabajo, GitHub Actions se utiliza para dirigir el proceso de comprobación del impacto del código en la modificación del esquema. Cada vez que se detecta un nuevo script de evolución Orion en el repositorio, se activa un workflow que ejecuta una serie de pasos encadenados:

1. Validación del script de evolución mediante las reglas definidas en el DSL Orion.
 2. Construcción de consultas CodeQL específicas asociadas a las operaciones del script.
 3. Ejecución de las consultas sobre el código fuente del proyecto.
-

4. Generación del informe en formato SARIF con las advertencias o errores encontrados.
5. Opción de registrar la versión del esquema en una tabla de versionado y generar un nuevo commit si se aprueba el cambio.

Gracias al uso de GitHub Actions, todo este proceso se ejecuta de forma transparente y automatizada, integrándose con el flujo de desarrollo habitual. Esta automatización garantiza que cualquier cambio en el modelo sea analizado de forma rigurosa antes de ser aplicado, permitiendo a los desarrolladores tomar decisiones informadas.

Además, el uso de esta infraestructura facilita la posterior intervención manual y asegura la repetibilidad del proceso, contribuyendo a la fiabilidad del sistema en escenarios reales de evolución continua.

2.5. Spring

Spring y su módulo **Spring Data JPA** han sido el framework utilizado en el caso de estudio. Estas tecnologías permiten definir el esquema relacional directamente en las entidades Java mediante anotaciones como `@Entity`, `@Table`, `@Column` o `@OneToMany`, y provisión automática de CRUD (Create, Read, Update, Delete) a través de interfaces de **Repository**. Además, las consultas ad hoc se expresan en JPQL o en métodos con `@Query`, lo que introduce cadenas literales estrechamente acopladas al modelo de datos. En tiempo de ejecución, Spring Data JPA genera dinámicamente las sentencias SQL correspondientes y gestiona el mapeo objeto-relacional.

Esta fusión de esquema y código conlleva varios riesgos cuando el modelo evoluciona:

- **Errores en tiempo de compilación:** campos renombrados o entidades eliminadas provocan que las consultas generadas por métodos derivados (`findBy...`) dejen de compilar.
- **Fallos en tiempo de ejecución:** consultas JPQL embebidas en `@Query` pueden lanzarse con excepciones `IllegalArgumentException` o `PersistenceException` si apuntan a columnas o relaciones obsoletas.
- **Cobertura limitada de pruebas:** muchas rutas de acceso a datos se ejercitan solo en integración, por lo que las roturas pueden pasar desapercibidas en fases tempranas.

Estas particularidades refuerzan la necesidad de un análisis predictivo capaz de detectar de forma estática y temprana las referencias a elementos del esquema que dejarán de existir o habrán cambiado de nombre tras aplicar una migración.

3. Estado del arte

Este capítulo no aspira a presentar un exhaustivo estado del arte, más propio de una tesis de máster o tesis doctoral, sino a mostrar en qué medida los objetivos de nuestro trabajo pretenden superar algunas limitaciones de las dos herramientas comerciales de evolución ágil de bases de datos más populares en la actualidad y también se incluirán algunos trabajos de investigación relacionados que ya se consideraron en el artículo del grupo ModelUM que introdujo la idea de usar CodeQL para detectar los puntos del código afectados por operaciones SCO y analizar el archivo SARIF generado, pero no abordó el proceso CI aquí tratado.

3.1. Herramientas existentes de evolución de esquemas CI

Flyway y Liquibase son las dos herramientas más usadas por los equipos de desarrollo. A continuación comentamos sus características más relevantes. Estas características estarán extendidas en el Anexo A

- **Definición de cambios:** Cada `changeSet` puede incluir cualquier instrucción DDL o DML (por ejemplo, `createTable`, `addColumn`, `insert`).
- **Ejecución única:** Liquibase mantiene una tabla interna (`DATABASECHANGELOG`) en la que anota cada `changeSet` aplicado, evitando duplicidades.
- **Rollback:** Opcionalmente, cada `changeSet` puede llevar asociada una sección de `rollback` que describa cómo revertir los cambios aplicados.
- **Ventaja principal:** Soporta múltiples formatos de definición, integra comprobaciones de checksum y permite generar diffs automáticos del esquema.

Flyway es otra herramienta muy popular y más minimalista enfocada en migraciones SQL versionadas. Su filosofía se basa en scripts numerados y ordenados de forma estricta.

- **Naming convention:** Cada script de migración debe nombrarse con el prefijo V, seguido de un número de versión y una breve descripción, por ejemplo:

`V3_add_user_table.sql`

- **Versionado y checksum:** Flyway registra en una tabla interna (`schema_history`) la versión aplicada y el checksum del fichero, garantizando que sólo se ejecute una vez y detectando modificaciones accidentales.
- **Migraciones undo (opcional):** Permite definir scripts `U#_...` para revertir migraciones, aunque desaconseja su uso por los riesgos de inconsistencias parciales.
- **Repeatable scripts:** Los archivos con prefijo `R_` se reejecutan siempre que cambie su checksum, facilitando la actualización de vistas, procedimientos o datos de referencia.
- **Ventaja principal:** Extremadamente sencillo de integrar, sólo requiere ficheros SQL numerados y no necesita configuraciones complejas.
- **Limitación clave:** Carece de un metamodelo de evolución; no predice ni señala en el código de aplicación los lugares donde las migraciones podrían quebrar consultas SQL estáticas.

Liquibase y Flyway no ofrecen ningún soporte relacionado con la adaptación del código tal como la facilidad de OrionFlow de detectar y predecir los puntos en el código de la aplicación afectados por las SCO, por ejemplo, podrían romper sentencias SQL embebidas o referencias a columnas renombradas. Soportan un gran número de sistemas relacionales y NoSQL, incluidos los que su uso está más extendido, pero no ofrecen un único lenguaje que soporte una taxonomía de cambios tan completa como la de Orion.

3.2. Trabajo de investigación relacionado

A lo largo de las últimas décadas se han propuesto diversos enfoques para automatizar la evolución de esquemas y su co-evolución con el código. En el ámbito relacional, Curino *et al.* presentaron PRISM++ [23], que define un conjunto de *Operadores de Modificación de Esquema* (SMOs) y *Operadores de Modificación de Restricciones* (ICMOs). Para cada SMO, PRISM++ genera instrucciones SQL que adaptan la estructura de la base de datos, migran los datos existentes y reescriben automáticamente las consultas afectadas (frecuentemente mediante vistas), garantizando la consistencia de las aplicaciones. Se trata de una gran contribución y la herramienta académica está disponible para su uso. Su limitación es centrarse en el modelo relacional.

DB-Main [24] ofrece un entorno MDE para bases de datos relacionales y orientadas a objetos tempranas. Se basa en tres pilares—(i) un metamodelo genérico GER, (ii) un proceso de ingeniería por transformaciones y (iii) el mantenimiento de un historial de cambios—y soporta ingeniería inversa, forward engineering y evolución de esquemas. Sin embargo, no contempla las particularidades de los sistemas NoSQL ni ofrece a los desarrolladores la posibilidad de conocer cómo el código será afectado.

En el terreno de la persistencia poliglota, el proyecto Typhon [25] introdujo TyphonML para definir esquemas y TyphonQL para consultas. Cuando el esquema evoluciona, las consultas TyphonQL se reescriben según un mapeo de tuplas <SCO, esquema, consulta> a manejadores específicos, dando como resultado cuatro posibles acciones: (i) sin cambio, (ii) reescritura segura, (iii) reescritura con advertencia, o (iv) rechazo con feedback al desarrollador. No obstante, exige un lenguaje de consulta propio y un runtime dedicado.

Para bases de datos de documentos, Darwin [26] construye un grafo de historia de esquemas y aplica migración de datos *perezosa*, permitiendo coexistir múltiples versiones de una misma entidad. Su algoritmo de reescritura genera subconsultas por versión y las unifica. Aunque efectivo, su metamodelo es simple y no incluye variaciones estructurales ni relaciones explícitas de grafos.

Para concluir este capítulo, señalamos las principales contribuciones del enfoque presentado en este TFG.

- Se trata de un enfoque genérico que usa Orion para expresar los cambios. Orion implementa una taxonomía definida sobre el *metamodelo unificado* U-Schema. Esta taxonomía abarca cada construcción de U-Schema: entidades, tipos de relación, variaciones, atributos, referencias y agregados, mediante operaciones atómicas (añadir, eliminar, renombrar, castear, promover, morfismo, etc.).
- Incluye un *sistema de análisis de impacto* que genera consultas CodeQL a partir de scripts Orion e integra los resultados en GitHub Actions, identificando con precisión usos de API y fragmentos de código afectado por las SCO.
- Motores de ejecución para MySQL, MongoDB, Cassandra y Neo4j que generan scripts nativos, asegurando que la misma especificación Orion dirige la evolución de esquemas en múltiples tecnologías. El análisis de impacto sólo se ha implementado para MySQL

Con ello se supera el enfoque limitado a SQL de PRISM++, el metamodelo cerrado de DB-Main, las dependencias de lenguaje de Typhon y la simplicidad de Darwin. La propuesta presentada unifica evolución de esquemas, migración de datos y adaptación de código en un único paradigma, proporcionando a los desarrolladores una solución integral, segura y que puede ser trazable en entornos de persistencia poliglota.

4. Visión general de la solución propuesta

Para facilitar la comprensión de la estrategia que se describe en detalle en los dos siguientes capítulos, en este capítulo se presenta un esquema general que ofrece una visión completa pero con menor nivel de detalle. Primero se introducirá una explicación básica del funcionamiento del motor Orion, responsable del proceso de generación de scripts para actualizar el esquema y los datos. A continuación, se introducirá la herramienta OrionFlow.

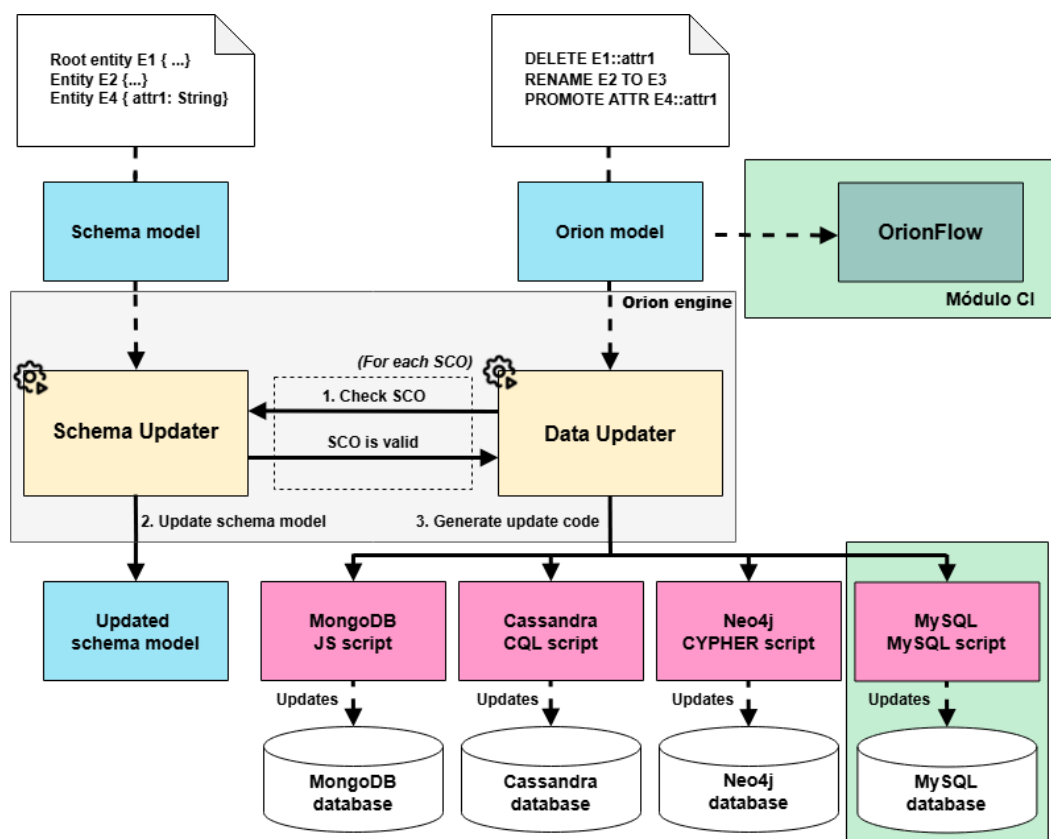


Figura 4.1: Arquitectura del motor Orion con integración continua y ampliación del Data Updater.

La figura 4.1 muestra la arquitectura del motor Orion (**Orion Engine**) conectada a OrionFlow. Como entrada recibe el esquema —ya sea un script Athena o un script del lenguaje específico del sistema de bases de datos— y el script de evolución escrito en el lenguaje Orion, que especifica la secuencia de SCOs a aplicar. Un paso inicial consiste en convertir el esquema y script Orion en modelos Ecore/EMF [22] que son conformes a los metamodelos creados para expresar la sintaxis abstracta de cada uno de los dos lenguajes.

Los modelos Orion y Athena se procesan por dos componentes esenciales: el *Schema Updater*, responsable de validar y modificar el esquema, y el *Data Updater*, encargado de generar el código específico para actualizar cada sistema gestor de bases de datos, MySQL en el caso del Data Updater creado en este TFG. El OrionEngine trabaja siguiendo 3 etapas.

1. *Validación de las operaciones de cambio de esquema (SCO)*: cada operación definida en el script Orion se verifica para garantizar que cumple con las restricciones del metamodelo Athena cargado en memoria.
2. *Actualización del modelo del esquema*: tras la validación, para cada operación, el esquema se modifica para incorporar los cambios especificados por las operaciones, generando una nueva versión del modelo manteniendo de este modo la coherencia entre operaciones.
3. *Generación automatizada del código de migración de datos*: finalmente, se produce el código específico para modificar la base de datos real, incluyendo scripts para bases de datos NoSQL y ahora relacional.

Existía soporte para MongoDB, Cassandra y Neo4j, y se ha extendido a MySQL en este TFG. En la figura 4.1 se muestran dos zonas verdes que señalan las partes que han sido desarrolladas en el TFG. Una de ellas corresponde al Data Updater que genera scripts de código SQL específicos de MySQL. La otra zona verde representa a OrionFlow que es responsable de dirigir todo el flujo automático una vez que se sube o se acepta un script Orion.

En la figura 4.2 se ilustra el comportamiento interno de OrionFlow. Hay dos workflows principales, llamados *CodeQL* y *Migration Schema* que realizan las siguientes tareas:

1. *CodeQL workflow*: a partir del modelo Orion, el generador se encarga de recorrerlo y obtener automáticamente un conjunto de consultas CodeQL (archivos .ql) correspondientes a cada operación de cambio de esquema. Estas consultas se guardan en el repositorio de scripts y, a continuación, el motor de CodeQL compila el código fuente de la aplicación para construir su base de datos interna. Entonces, ejecuta todas las consultas sobre esa base de datos y exporta los resultados en un informe SARIF. Este fichero permite mostrar, en la sección de
-

Code scanning alerts, las líneas de código afectadas por el cambio de esquema, con mensajes claros que ayudan al desarrollador a identificar y corregir errores antes de aplicar la migración.

2. *Migration Schema workflow*: toma el mismo script Orion como punto de partida con la finalidad de gestionar la tabla de versionado y la publicación de los scripts de migración en un repositorio de versiones. Primero, comprueba si el desarrollador ha aprobado el cambio cuando realiza el commit; en caso afirmativo, se agrega a la carpeta de migraciones del repositorio y se registra una nueva entrada en la tabla de versiones (similar a Flyway), de modo que quede trazada la evolución del esquema. Finalmente, se genera un commit automático que incluye el script almacenado, la versión del esquema incrementada y la tabla de versiones actualizada. En caso de que el desarrollador no acepte, el script Orion se descarta.

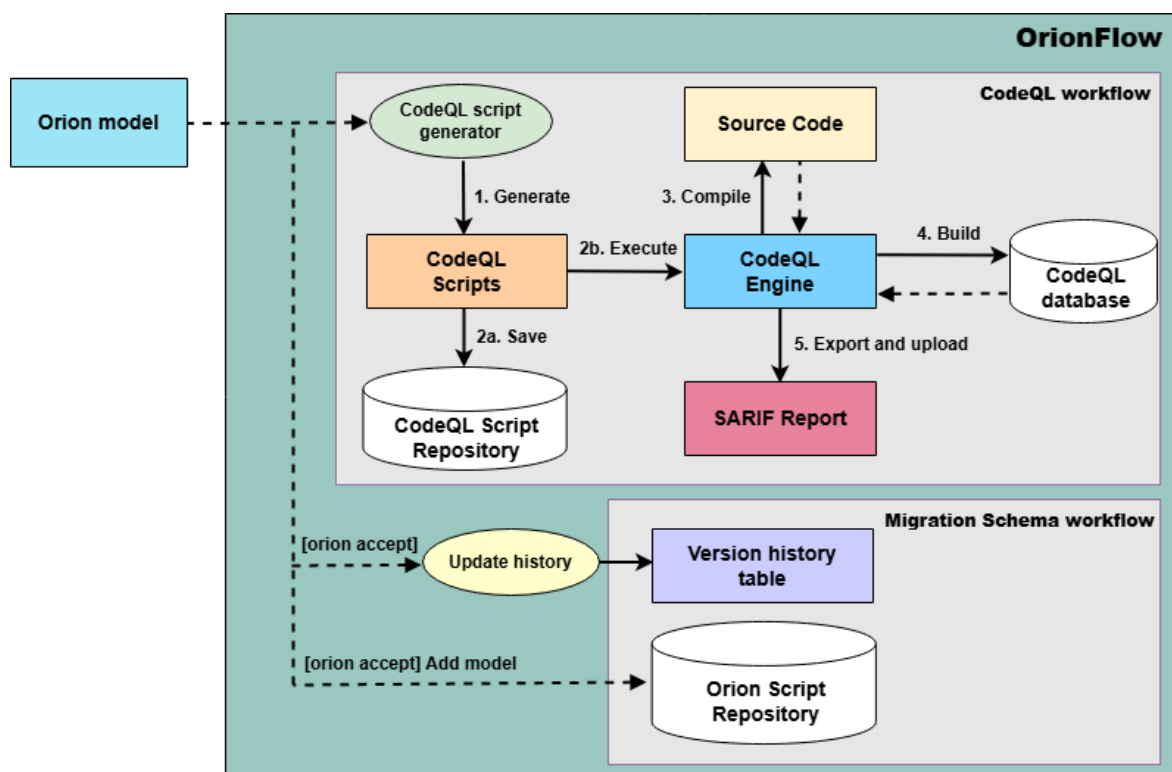


Figura 4.2: Funcionamiento interno de los workflows que conforman la integración continua.

Por tanto, el trabajo realizado en este TFG ha servido para que el motor Orion ofrezca soporte a bases de datos relacionales, en concreto MySQL, y lo que ha sido más complejo ha sido crear la herramienta OrionFlow que predice el impacto de cada SCO de un script Orion en el código de la aplicación.

5. Actualización del esquema y de los datos

En este capítulo se describe la extensión del motor Orion para soportar bases de datos relacionales. Primero se discute el mapping de SCO de la taxonomía Orion a operaciones de cambio de esquema y, después, la implementación del mapping. También se comentarán las SCOs que requieren un cambio en los datos. Por ejemplo, renombrar una columna no requiere ninguna adaptación de los datos, pero dividir una tabla en dos sí lo requiere.

5.1. Mapping SCOs Orion a cambios en esquema relacional

La Tabla 5.1 muestra la correspondencia entre las operaciones Orion soportadas y la actualización requerida en el esquema SQL. Nótese que este mapping es independiente del sistema relacional. Por otro lado, las SCO que requieren una modificación de los datos son marcadas con un símbolo asterisco. Los procedures que aparecen son descritos en la tabla 5.2.

A continuación se describe el mapping organizado en las diferentes categorías de SCO en la taxonomía. En la siguiente sección se explicarán los detalles de implementación, las posibles dificultades y las decisiones de diseño tomadas. Por cuestiones de tiempo, algunas SCO no han sido tratadas, por ejemplo, llevar un agregado a la entidad raíz. Cabe destacar que muchas operaciones no solo modifican la estructura del esquema, sino también los datos existentes. Por ejemplo, operaciones destructivas como borrar un atributo o una entidad eliminan directamente columnas o tablas, y por tanto implican pérdida de datos. Otras, como el renombramiento, son puramente estructurales y no alteran el contenido almacenado, limitándose a cambiar identificadores sin afectar a la información subyacente.

Sin embargo, existe un tercer grupo de operaciones —como extraer una tabla o promover un atributo— que requieren una transformación activa de los datos. En estos casos, el motor genera sentencias `INSERT INTO ... SELECT ...` y emplea procedimientos almacenados con uniones (`JOIN`) para copiar y reorganizar los datos entre tablas intermedias. Aquellas donde se haga una manipulación entre tablas de los datos están marcadas con un asterisco (*).

Entity Operations

- *Add E*: Se empieza creando la tabla principal. Dependiendo del tipo de atributo pueden ocurrir dos cosas: si es un tipo primitivo hay una correspondencia directa con una columna pero si es un tipo compuesto (Set, Map, o Tuple), se crea una tabla auxiliar que simula el comportamiento esperado del tipo.
- *Delete E*: Para su realización se invocan de forma secuencial dos procedimientos: uno para eliminar las entidades débiles dependientes, y otro que se encarga de eliminar las claves foráneas que referencian a E. Cuando todas las dependencias se han resuelto, se elimina la tabla principal.
- *Rename E*: Renombra la tabla principal además de los agregados y tipos especiales ligados a esta.
- *Extract** / *Split**: Estas operaciones descomponen la tabla original en nuevos esquemas junto con la migración de los datos. En *Extract*, se genera una única tabla con el subconjunto de columnas seleccionado y la tabla origen permanece intacta. En *Split*, en cambio, se crean dos tablas hijas, cada una con su propio conjunto de columnas y se descarta la tabla original de la misma forma utilizada en *Delete E*.

Feature Operations

- *Delete E.f*: Se invoca un procedimiento que elimina las restricciones sobre la columna y posteriormente se elimina. En caso de ser un tipo compuesto, se elimina la tabla correspondiente.
 - *Rename E.f*: Renombra la columna o tabla auxiliar correspondiente; si *f* es de un tipo con restricciones de integridad especiales (enumerado, rango, expresión regular), primero se invoca al mismo procedimiento que en la operación de borrado de esta categoría y, a continuación, se procede a renombrar la propiedad.
 - *Copy** / *Move**: Estos métodos son similares a los métodos *Extract* y *Split* pero aplicado a columnas. Ambas operaciones trasladan un atributo de una tabla origen a otra destino, generando primero la columna y migrando los datos correspondientes. En *Copy*, la columna original se mantiene intacta, mientras que en *Move* se elimina una vez completada la migración.
-

Tabla 5.1: Scripts para la base de datos MySQL a partir de las operaciones de Orion

Operación Orion SCO	Acciones en base de datos MySQL
Entity Operations	
Add E	CREATE <i>table</i> , n ^o $E.f \in st^1 \times$ (CREATE <i>subtable</i>)
Delete E	PROCEDURE <i>dropWeakEntities</i> , PROCEDURE <i>dropFKsFromTable</i> , DROP <i>table</i>
Rename E	RENAME <i>table</i> . n ^o $E.f \in st \times$ (RENAME <i>subtable</i>)
Extract E into E'	CREATE <i>table</i> , INSERT INTO ... SELECT
Split E into E', E''	2 x (CREATE <i>table</i> , INSERT INTO... SELECT), PROCEDURE <i>dropWeakEntities</i> , PROCEDURE <i>dropFKsFromTable</i> , DROP <i>table</i>
Relationship type Operations	
Add R	CREATE <i>table</i>
Delete R	DROP <i>table</i>
Rename R	RENAME <i>table</i>
Feature Operations	
Delete $E.f$	PROCEDURE <i>dropConstraintsForColumn</i> , DROP (<i>column</i> <i>table</i>) dependiendo si $f \in st$
Rename $E.f$	PROCEDURE <i>dropCheckConstraintsForColumn</i> , RENAME (<i>column</i> <i>table</i>) dependiendo si $f \in st$
Copy $E_1.f$ to $E_2.f$	ADD <i>column</i> , UPDATE ... SET
Move $E_1.f$ to $E_2.f$	ADD <i>column</i> , UPDATE ... SET, PROCEDURE <i>dropConstraintsForColumn</i> , DROP <i>table</i>
Attribute operations	
Add $E.attr$	(ALTER <i>table</i> ADD <i>column</i> CREATE <i>TABLE</i>) dependiendo si $attr \in st$
Cast $E.attr$	ALTER <i>table</i> MODIFY <i>column</i>
Promote $E.attr$	CREATE <i>table_{aux}</i> , PROCEDURE <i>update_fk_schema</i> , PROCEDURE <i>update_fk_data</i> , DROP PRIMARY KEY, ADD PRIMARY KEY, DROP <i>table</i> , RENAME <i>table_{aux}</i>
Reference operations	
Add $E.ref$ to <i>Entity</i>	ADD <i>column</i> , ADD CONSTRAINT ... FOREIGN KEY
Add $E.ref$ to <i>RelShip</i>	CREATE <i>table</i> , PROCEDURE <i>dropFKsInTable</i> , DROP PRIMARY KEY, ADD PRIMARY KEY, 2 x ADD CONSTRAINT ... FOREIGN KEY
Cast $E.ref$	PROCEDURE <i>castColumnFKs</i>
Aggregate operations	
Add $E.aggr$	CREATE <i>table</i>

¹ st es un campo de tipo compuesto ($\in \{\text{Set}, \text{Map}, \text{Tuple}\}$). Entonces se trata como una tabla auxiliar no como una columna.

Attribute operations

- *Add E.attr*: Modifica el esquema creando una columna o una tabla dependiendo de si es un tipo compuesto.
- *Cast E.attr*: La columna se redefine con el nuevo tipo, siempre que el nuevo tipo de sea compatible con el tipo de dato anterior.
- *Promote* E.attr*: Para incorporar un nuevo campo a la clave primaria, primero se crea una tabla temporal que define como PK el campo adicional. A continuación, se migran todos los datos desde la tabla original a la temporal. Seguidamente, se ajustan las claves foráneas de las tablas dependientes para que referencien el nuevo conjunto de columnas PK. Por último, se elimina la tabla antigua y se renombra la temporal con su nombre.

Relationship Operations

Las operaciones sobre tipos de relación ($n : m$) se traducen de igual manera a las de entidad, pero sin necesidad de gestionar restricciones de integridad referencial en la eliminación.

Reference operations

- *Add E.ref to Entity*: Se añade una nueva columna que referencia a la otra entidad dependiendo de la multiplicidad. Si la relación es $1 : 1$ o $0 : 1$, se añade directamente la nueva columna de clave foránea a la tabla de origen. En cambio, si la relación es $1 : n$ o $0 : n$, la columna se crea en la tabla destino. Finalmente se establece la restricción de integridad referencial.
- *Add E.ref to Relationship*: Construye una tabla a la que se le añaden las dos referencias. Cada vez que se añade una nueva referencia se actualiza tanto la clave primaria como las claves foráneas.
- *Cast E.ref*: Se invoca a un procedimiento que modifica el tipo del atributo tanto en la columna actual como en las tablas donde es referenciado.

Aggregate operations

- *Add E.aggr*: Se crea una tabla con clave primaria y foránea que corresponden con la clave primaria de la tabla padre.
-

5.2. Implementación

El lenguaje Orion fue implementado en el framework Xtext [27] destinado a facilitar la creación de lenguajes en Eclipse/EMF, y proporciona Xtend para escribir transformaciones de modelos (m2m y m2t). Por tanto, la implementación del mapping se ha llevado a cabo en una copia del proyecto Xtext creado para Orion, donde se adaptaron clases existentes y se crearon nuevas para la transformación de scripts Orion a MySQL. La figura 5.1 presenta un diagrama de clases que incluye únicamente las clases añadidas, con sus atributos y métodos más relevantes. A partir de este diagrama explicaremos brevemente las diferentes clases, luego describiremos el flujo de ejecución en el siguiente apartado, y finalmente expondremos las decisiones de diseño adoptadas.

5.2.1. Arquitectura General

En este apartado presentamos el desglose interno de la extensión realizada en el componente Data Updater para entornos relacionales. La figura 5.1 muestra el diagrama de clases principales que conforman la extensión, y se describirá brevemente cada una de ellas y sus responsabilidades más relevantes.

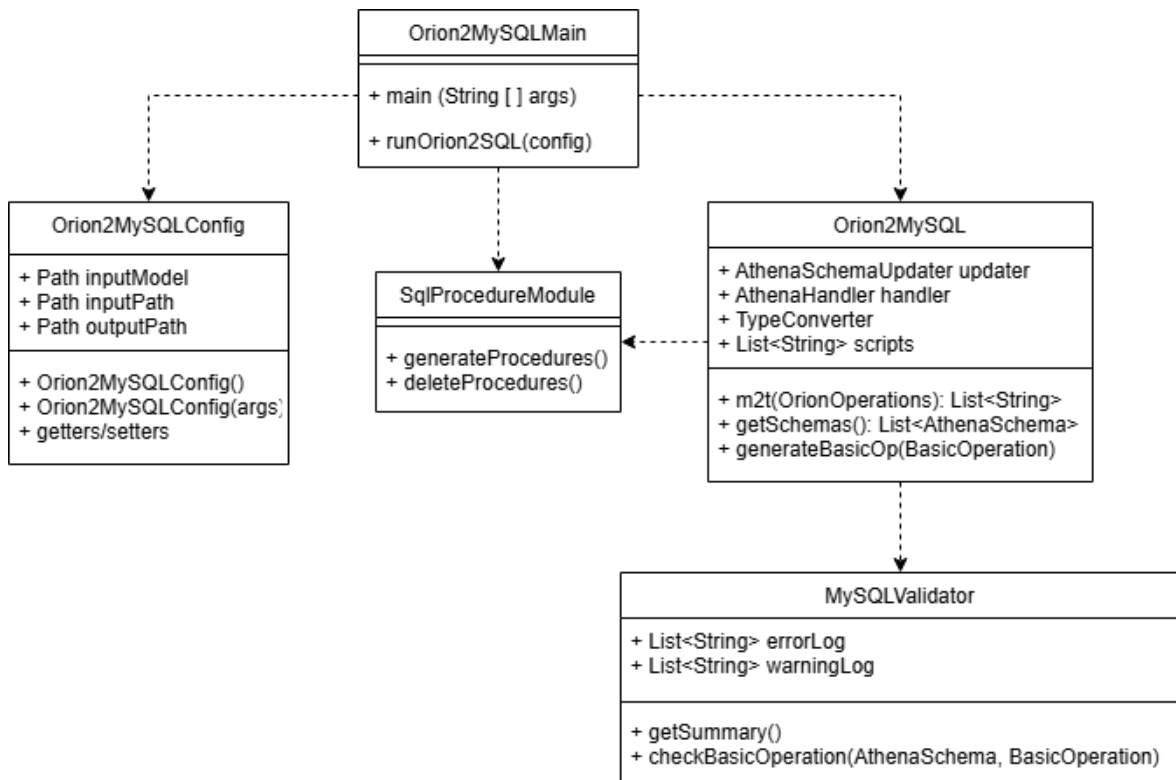


Figura 5.1: Diagrama de clases UML para el DataUpdater relacional

Orion2MySQLMain Punto de entrada a la aplicación; encargada de interpretar los parámetros recibido desde la consola de comandos en la clase *Orion2MySQLConfig* (encargada de parsear los ficheros y las rutas de entrada/salida) y arranca el servicio *Orion2MySQL*.

Orion2MySQL Presenta la mayor parte de la lógica: infiere o carga esquemas Athena, valida operaciones, convierte cada operación a SQL acumulando las conversiones en memoria mediante una lista de *scripts*.

SqlProcedureModule: Proporciona métodos de creación y destrucción de los procedimientos almacenados auxiliares necesarios para ciertas migraciones de esquema.

MySQLValidator Verifica que cada operación de Orion sea aplicable al esquema en memoria, registrando errores y advertencias.

5.2.2. Flujo de ejecución

Inicio y configuración La aplicación se inicia recibiendo como parámetros la ruta del script de entrada con extensión *.orion* y la carpeta de destino para los scripts de migración generados. Estos valores pueden suministrarse explícitamente al lanzar la herramienta o, si se omiten, emplean rutas por defecto preconfiguradas en el propio código.

Carga de módulos Athena y Orion Para que el programa pueda ejecutarse y recorrer los modelos de Athena y Orion, estos deben ser cargados previamente en memoria.

Generación de scripts de migración El fragmento de código 5.1 muestra el núcleo principal de la transformación M2T: por cada operación primero se valida su compatibilidad con el esquema Athena, luego se traduce inmediatamente a la instrucción SQL equivalente y, finalmente, se aplica ese cambio sobre el modelo Athena, de manera que el esquema siempre refleja el estado más reciente antes de procesar la siguiente operación.

Código 5.1: Método principal empleado en la generación de las operaciones a MySQL

```
1 def generateOperations(java.util.List<BasicOperation> operations) '''
2     «FOR op : operations SEPARATOR "\n"»
3         «{validator.checkBasicOperation(schemaUpdater.schema, op); "" }»
4         «generateBasicOp(op)»
5         «schemaUpdater.processOperation(op)»
6     «ENDFOR»
7 '''
```

Escritura en ficheros Una vez realizadas las transformaciones, se vuelcan en la ruta indicada en la fase de configuración los siguientes ficheros:

1. Un script `.sql` que contiene las operaciones de evolución traducidas a sentencias MySQL.
2. Un segundo script `.sql` con los procedimientos almacenados auxiliares necesarios para ciertas migraciones.
3. Un fichero `.athena` que refleja la versión actualizada del esquema, o bien define el esquema completo si partimos de un script de creación inicial en Orion.

5.2.3. Decisiones de diseño

Para la generación de código, se ha creado una transformación `m2t` en Xtend. Este lenguaje que forma parte del framework Xtext proporciona un potente mecanismo de text templates destinado a la generación de código. Además, proporciona una operación `dispatch` que es también muy útil: cuando recibes un parámetro que puede ser un objeto de diferentes subclases de una jerarquía, permite invocar automáticamente el método plantilla que corresponde sin necesidad de escribir manualmente grandes bloques de `if` o un `switch`. En el Anexo B se incluyen ejemplos de uso de ambas características.

Dado que en las bases de datos relacionales no existen tipos como conjuntos, mapas, tuplas o agregados, es preciso representar estas estructuras mediante tablas auxiliares y usando claves ajenas. Así se mantiene la flexibilidad de Athena y NoSQL, aceptando a cambio un mayor número de tablas y algo más de complejidad en la lógica.

Siguiendo las buenas prácticas de SQL, la herramienta obliga en la creación de ciertas operaciones a que cumplan una serie de requisitos:

- *Clave primaria obligatoria:* se rechaza la creación de tablas que no incluyan al menos un atributo marcado como clave, garantizando siempre la existencia de una PRIMARY KEY.
 - *Prohibición de claves primarias en agregados y relaciones n:m:* no se permite que estas tablas definan su propia clave primaria; en su lugar, la clave debe corresponder a la relación con la entidad padre o a las columnas que representan la asociación.
 - *Detección de palabras reservadas:* se emite una advertencia cuando un campo o nombre de una tabla utiliza identificadores reservados de SQL (USER, ORDER, GROUP, etc.), evitando futuros conflictos de sintaxis en consultas y definiciones DDL (Data Definition Language).
-

Para MySQL se ha optado por encapsular la lógica más compleja, como bucles o la desactivación temporal de restricciones, en procedimientos almacenados, de modo que el código de migración sea más limpio y fácil de mantener. A continuación, se muestran dos ejemplos que ilustran por qué en ciertos casos son una mejor alternativa a recorrer el modelo:

- *Iteración dinámica al eliminar restricciones:* En lugar de rastrear manualmente todas las entidades y claves foráneas que apuntan a una tabla para emitir individualmente un `ALTER TABLE ... DROP FOREIGN KEY`, un único procedimiento puede consultar `INFORMATION_SCHEMA` —la base de datos interna de MySQL que expone metadatos sobre esquemas, tablas, columnas y restricciones— y generar dinámicamente los comandos necesarios para eliminar todas las restricciones que referencian esa tabla. Además si se usa en varias partes del código simplemente basta con invocarlo con la llamada `CALL`.
- *Atomicidad en los procedimientos:* En el ejemplo anterior, al agrupar todos los pasos de eliminación de restricciones en un único procedimiento almacenado, si en cualquier momento falla alguno de los comandos (`DROP FOREIGN KEY` inexistente, clave aún en uso, etc.), se ejecuta automáticamente un `ROLLBACK` y todo el conjunto de operaciones se deshace, garantizando la coherencia del esquema.

En la tabla 5.2 se muestran los procedimientos más relevantes utilizados en las migraciones y un breve resumen de su función. Su definición completa puede consultarse en la sección B.3 del Anexo B.

Tabla 5.2: Procedimientos almacenados más relevantes en MySQL

Nombre del procedimiento	Descripción breve
<code>dropWeakEntities</code>	Elimina automáticamente tablas “débil” que referencian a una entidad padre.
<code>dropFKsInTable</code>	Elimina todas las claves foráneas definidas en una tabla objetivo.
<code>dropFKsFromTable</code>	Elimina todas las claves foráneas que apuntan a una tabla objetivo.
<code>dropConstraintsForColumn</code>	Elimina cualquier restricción (PK, FK, UNIQUE, CHECK...) que afecte a una columna específica de una tabla.
<code>dropCheckConstraintsForColumn</code>	Elimina todas las restricciones CHECK asociadas a una columna concreta.
<code>update_fk_schema</code>	Reconstruye automáticamente las claves foráneas cuando cambia la clave principal, añadiendo las columnas nuevas y recreando las restricciones.
<code>update_fk_data</code>	Copia los datos de las columnas promocionadas a todas las tablas que las usan, usando una sola sentencia <code>UPDATE ... JOIN</code> .
<code>castColumnFKs</code>	Cambia el tipo de datos de una columna y de todas las columnas que la referencian.

6. Actualización del código

En el Capítulo 4 se introdujo la arquitectura de la herramienta OrionFlow que ofrece el soporte a la integración continua de bases de datos, pero nos limitamos a dar una breve explicación de sus workflows componentes. En este capítulo se profundiza en sus detalles de implementación y las decisiones de diseño clave que permiten, por un lado, generar de forma automática un informe visual con los posibles impactos en el código y, por otro, mantener un historial preciso de las migraciones válidas. Tras analizar por separado ambos flujos —el de detección de riesgos mediante CodeQL y el de gestión de versiones de scripts Orion— se presenta un diagrama que muestra de forma práctica cómo un desarrollador interactúa con la herramienta durante su trabajo diario.

6.1. Workflow: CodeQL

Como se comentó anteriormente en la Figura 4.2, una vez se detecta un modelo Orion, el proceso CI se bifurca en dos flujos principales. El primero está dedicado a generar y ejecutar *on-the-fly* consultas CodeQL para cada SCO en un script Orion. Como paso previo a la ejecución, el motor CodeQL crea una base de datos de código compilada junto con el árbol de sintaxis abstracta (AST) para el código de la aplicación que se debe actualizar. Para generar las consultas CodeQL se ha creado otra transformación m2t, y para su ejecución, se adoptó la estrategia de empaquetar en un JAR ejecutable toda la lógica de extracción y recorrido del modelo Orion y el generador carga el modelo Orion en memoria y la transformación m2t para producir las consultas CodeQL específicas para cada operación del script que implica cambios sobre el código que deben ser notificados a los desarrolladores.

El generador sólo produce consultas CodeQL para aquellas SCO que modifican o eliminan elementos existentes en el código —es decir, cambios que efectivamente pueden provocar errores o inconsistencias si no se abordan—, mientras que las operaciones aditivas (por ejemplo, la creación de nuevas tablas o columnas sin referencia previa) quedan fuera de este proceso de predicción, debido a que no hay código que analizar.

Para que CodeQL pueda obtener una base de código compilada, es necesario compilar todos los módulos del repositorio que contienen código fuente. De este modo, CodeQL dispondrá de la versión compilada de cada módulo y podrá recorrer correctamente su estructura interna.

Para que el workflow pueda hacer uso de los dos componentes explicados previamente —el motor CodeQL y el generador de consultas—, en el repositorio debe existir un

directorio oculto `.github/codeql`. En su interior se encuentran:

- **queries/**: Carpeta donde se almacenan las consultas `.ql` generadas dinámicamente (una por cada operación detectada en el script Orion), junto con los ficheros auxiliares `utils.ql` y `suite.qls` que contienen metadatos y macros de apoyo. Estas consultas se regeneran y sobrescriben automáticamente en cada ejecución del workflow.
- **generator.jar**: El JAR ejecutable que implementa el analizador Orion2CodeQL mencionado anteriormente.
- **codeql-config.yml**: Archivo de configuración para la acción oficial de GitHub CodeQL, donde se indica el lenguaje (Java/Kotlin) y las rutas de origen a escanear donde se especifican que directorios se deben incluir o excluir durante el análisis.

De este modo, el workflow dispone de todo lo necesario para ejecutar el motor CodeQL. Un ejemplo visual de las alertas puede verse en figura 6.1. En ella se notifica al desarrollador que se hace uso de una query donde el nombre de la entidad será renombrada y puede provocar errores en tiempo de ejecución.



Figura 6.1: Alerta en la sección de code-scanning de Github.

A continuación, se ofrece una versión simplificada de las etapas dentro del workflow tratado en la sección:

1. *Disparador*: el workflow se dispara sobre cada **push** que modifique o añada un fichero `.orion`.
2. *Checkout*: se clona el repositorio para disponer del código y scripts.
3. *Extracción del nombre de fichero*: se lee el mensaje del *commit* para localizar el script Orion que se acaba de subir. si no existe realmente ese `.orion`, se aborta el análisis.

4. *Generación de consultas*: Invocación de la herramienta `generator.jar`.
5. *Inicialización de CodeQL*: se configura la acción oficial con nuestras consultas y el lenguaje (Java/Kotlin).
6. *Análisis CodeQL*: la acción oficial ejecuta el `analyze` y produce un informe SARIF.
7. *Publicación de resultados*: GitHub Code Scanning procesa el SARIF y subraya en colores (rojo/amarillo) los lugares del código propensos a fallar tras la migración.

6.2. Workflow: Migration Schema

El segundo flujo de integración continua se encarga de gestionar versiones históricas de los scripts Orion y de llevar un registro ordenado de cada migración aceptada. Para ello, se define un nuevo `workflow` que se dispara únicamente sobre la rama `main` cuando el mensaje de `commit` comienza con la palabra clave `orion`. El proceso interno tiene lugar mediante una serie de pasos:

1. *Checkout y contexto*: Se clona el repositorio para disponer de los cambios. Es necesario recuperar los commits anteriores para el paso siguiente.
 2. *Detección del script*: Extrae del mensaje de commit mediante un patrón de expresión regular el nombre del fichero `.orion` afectado y lo busca en el commit anterior dentro del árbol de versiones de GitHub.
 3. *Parsing de la acción*: Interpreta si la operación es `aceptar` o `cancelar`, así como la descripción opcional incluida tras la coma en el mensaje.
 4. *Preparación del directorio `orion/`*: Se produce una creación previa del directorio en caso de no existir y junto con el se añade:
 - `orion/orion_scripts/`: para archivar todas las versiones de los scripts orion utilizados durante el ciclo de vida del esquema de la base de datos.
 - `orion/orion_schema_version.txt`: con el contador de versiones. Se parte de la versión cero, y se incrementa en cada actualización de la tabla.
 - `orion/version_table.md`: En ella se mostrarán los datos más relevantes para obtener así la trazabilidad entre versiones.
 5. *Desarrollador acepta los cambios*
 - Incrementa en uno el valor de `orion_schema_version.txt`.
 - Extrae el script previo desde el commit anterior y lo mueve a al repositorio de scripts modificando su nombre para darle mayor legibilidad.
-

- Añade una fila en `version_table.md` con: versión, nombre de fichero, descripción, fecha y SHA abreviado.
6. *Desarrollador rechaza los cambios* Elimina completamente el fichero `.orion` referenciado..
 7. *Commit final*: Utiliza una acción de `git-auto-commit` para agrupar y subir todos los cambios (contador, tabla y scripts archivados) de vuelta a `main`.

Version	Fichero <i>.orion</i>	Descripción	Fecha	Commit SHA
0	V0_user_delete.orion	Se ha borrado la entidad Usuario	2025-05-25 04:27 UTC	d7a6b55
1	V1_add_livepost.orion	Se ha añadido la entidad LivePost	2025-06-02 11:15 UTC	a1b2c3d
2	V2_rename_post_userpost.orion	Se ha renombrado la entidad Post a UserPost	2025-06-10 09:42 UTC	e4f5g6h

Tabla 6.1: Historial de versiones de scripts Orion en el fichero `version_table.md`

Con esta estrategia, cada migración aceptada queda archivada y numerada de forma inmutable, mientras que las cancelaciones limpian el espacio de trabajo y evitan la acumulación de scripts obsoletos. En la Tabla 6.1 se registran los campos clave —número de versión, nombre de script, descripción de la acción, fecha y SHA del commit— de modo que se pueda reconstruir en todo momento qué cambios introdujo cada migración. Para ello se utiliza una nomenclatura similar a Flyway (por ejemplo, V0_..., V1_...), facilitando la correlación con las “versiones” de esquema, al tiempo que preservamos la trazabilidad del modelo Orion. A diferencia de Flyway, que gestiona secuencias de scripts SQL numerados y aplica cada uno automáticamente al despliegue, aquí mantenemos un historial explícito en Markdown donde, además, se documenta la lógica de negocio asociada a cada versión y su procedencia exacta en el repositorio. De este modo ganamos en transparencia: no solo sabemos qué migración se ejecutó, sino también por qué y cuándo se revisó.

Una vez explicados ambos flujos de trabajo, puede surgir la duda de como un desarrollador puede hacer uso de la herramienta de manera efectiva. Para solventar esas dudas se muestra en la Figura 6.2 una vista de conjunto que une la perspectiva del desarrollador con el engranaje interno de la plataforma. En este diagrama, hay dos matices que no se han destacado hasta ahora y que marcan la diferencia en nuestro proceso.

Para realizar un análisis de código efectivo por parte del desarrollador se debe realizar en dos fases distintas: primero, el commit del script `.orion` que dispara la generación de consultas CodeQL; y, a continuación, un commit vacío que recoge la decisión del desarrollador. En esta segunda fase, se utiliza la opción `–allow-empty` de GitHub Actions para crear commits sin cambios en el contenido. Esta directiva es necesaria porque

a la hora de hacer un commit, Github obliga a reflejar algún cambio en el estado del repositorio. El mensaje del commit contendrá el mensaje adecuado (orion(accept): nombre_script, descripcion | orion(cancel): nombre_script). De este modo, basta con leer la cabecera del commit para extraer los parámetros necesarios —nombre de script, descripción, acción— y actualizar la tabla de versiones o eliminar el archivo correspondiente. Con este procedimiento, mantenemos un rastro indiscutible de quién decidió qué, y cuándo, reforzando la responsabilidad y la trazabilidad de cada cambio de esquema.

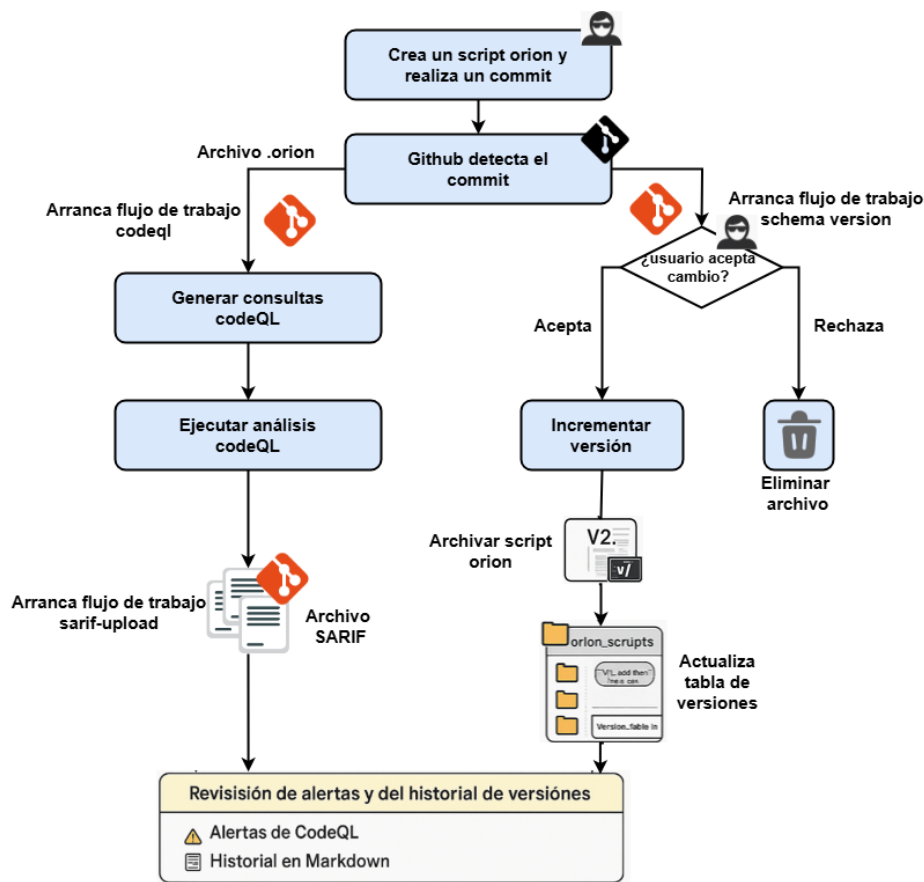


Figura 6.2: Visión integral del workflow automatizado en Orion, combinando análisis CodeQL y control de versiones de scripts.

En segundo lugar, el diagrama incorpora también el flujo de alertas SARIF, totalmente independiente, que se activa cada vez que se genera un informe .sarif. Sin esperar a la fusión de pull requests ni a etapas manuales adicionales, el workflow auxiliar detecta esos archivos, los envía al panel de Code Scanning de GitHub y convierte de inmediato los resultados en alertas visuales. Esto asegura una retroalimentación instantánea, permitiendo al equipo reaccionar en tiempo real.

Todo el código utilizado y mencionado a lo largo de los distintos capítulos de este Trabajo Fin de Grado se encuentra publicado en el repositorio público de GitHub: <https://github.com/Zerep0/TFG>. Aquí solo se ofrecen una visión genérica e ilustrativa de los comandos. La organización de carpetas, así como las instrucciones para clonar, compilar o ejecutar los ejemplos, están detalladas en el archivo **README.md** situado en la raíz de dicho repositorio.

7. Validación

7.1. Caso de estudio: Umugram

En este capítulo se valida el enfoque propuesto utilizando como caso de estudio *Umugram*, una aplicación tipo red social desarrollada con Java, Spring Boot y MySQL. El modelo de clases del dominio de *Umugram* puede apreciarse en la Figura 7.1. En el centro del diseño se encuentra la entidad **User**, que incluye información básica como el nombre de usuario, correo electrónico, contraseña y fecha de nacimiento, así como relaciones que permiten identificar a los usuarios que sigue o que le siguen. Además, cada usuario dispone de un perfil personal donde se almacena su avatar, biografía y un enlace a su sitio web.

Para compartir contenido, los usuarios pueden crear diferentes tipos de publicaciones: texto, fotografía, vídeo o retransmisiones en directo, todas ellas registradas con información sobre su autor y la fecha de creación. Los comentarios permiten interactuar bajo cada publicación, incluyendo respuestas encadenadas mediante una referencia al comentario original. En este caso de uso, se ha decidido persistir la información relativa a usuarios, publicaciones y comentarios.

Para la implementación de la aplicación se optó por generar el esqueleto del proyecto a través de **Spring Initializr**¹, un servicio web que permite crear en pocos clics un proyecto Spring Boot preconfigurado. Basta con seleccionar la versión del framework, el sistema de compilación (Maven o Gradle), los identificadores del proyecto (grupo y artefacto), y las dependencias necesarias como Spring Data JPA. El resultado es un archivo ZIP con la estructura de carpetas, el fichero `pom.xml` y una clase principal anotada con `@SpringBootApplication`, lista para compilar y ejecutar.

Código 7.1: Repositorio con extensión `JpaRepository` en *Umugram*

```
1 public interface UserRepository extends JpaRepository<User, Long> {  
2     @Query("SELECT f FROM User u JOIN u.followers f  
3         WHERE u.id = :userId")  
4     List<User> findFollowersById(@Param("userId") Long userId);  
5 }
```

Para la capa de persistencia, aprovechamos el soporte de Spring Data JPA mediante interfaces que extienden `JpaRepository`, lo cual nos permite abstraernos casi por completo del código de acceso a datos y definir con facilidad operaciones CRUD e in-

¹<https://start.spring.io/>

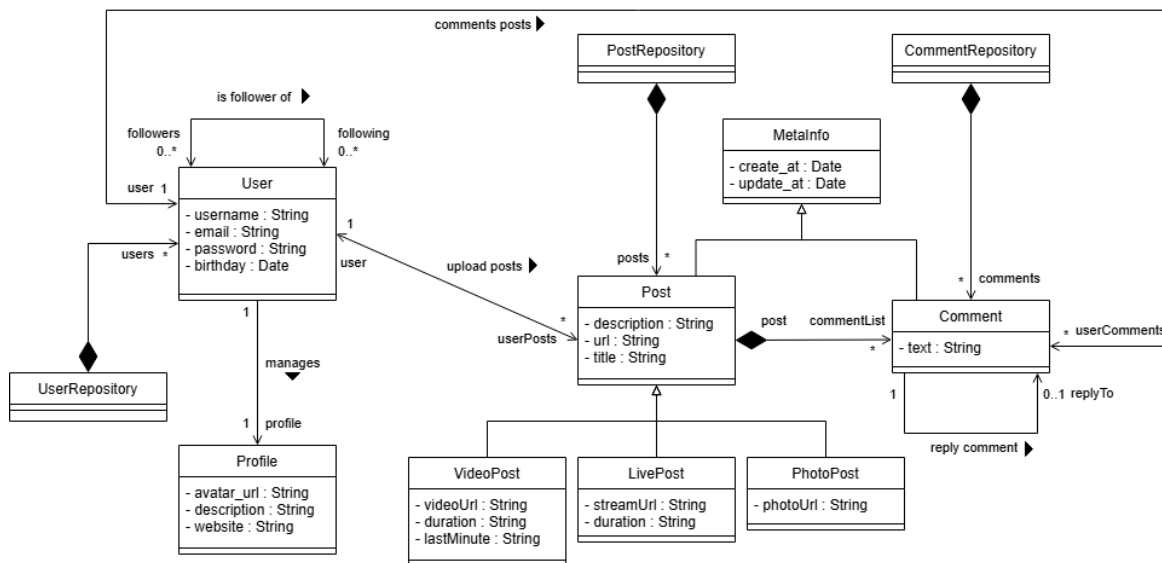


Figura 7.1: Diagrama de clases UML del caso de estudio Umugram

cluso consultas JPQL personalizadas usando `@Query`. En el fragmento de código 7.1 se muestra un ejemplo del repositorio de los usuarios con una consulta personalizada para buscar seguidores por su identificador.

Sin embargo, muchas organizaciones aún mantienen proyectos legados que no emplean frameworks como Spring debido al coste y riesgo que implicaría una migración completa, por lo que es habitual encontrarse con código que interactúa directamente con la especificación JPA. Para validar estos escenarios y asegurarnos de que nuestro conjunto de consultas de análisis estático en CodeQL detecta correctamente las llamadas programáticas al API de JPA sin depender de Spring, incluimos un repositorio adicional que utiliza directamente el `EntityManager`. Este componente, definido por la especificación JPA, actúa como punto de entrada genérico para crear y ejecutar consultas —tanto JPQL como SQL nativas— sobre cualquier implementación de JPA. En el ejemplo de la Figura 7.2 se extrae el contexto de persistencia de JPA para realizar una consulta en el repositorio de publicaciones siguiendo la nomenclatura tradicional del API.

Código 7.2: Repositorio con métodos del `EntityManager` en *Umugram*

```

1 public class CustomPostRepositoryImpl implements CustomPostRepository {
2     @PersistenceContext private EntityManager entityManager;
3
4     @Override
5     public List<Post> cFindPostsWithLongCaptions(int minLength) {
6         Query q =
7             entityManager.createQuery("SELECT p FROM Post p
8                                     WHERE LENGTH(p.caption) > :minLength");
9     }
10 }

```

```
9   q.setParameter("minLength", minLength);  
10  return q.getResultList();  
11  }  
12 }
```

7.2. Resultados obtenidos en Umugram

7.2.1. Generación en MySQL de Umugram con Orion

Para crear la base de datos inicial de Umugram, se ha utilizado el propio lenguaje Orion, como si partieramos de un esquema vacío y luego añadiéramos todas las entidades y sus relaciones. De esta forma el motor de Orion nos genera los scripts de creación de la base de datos en MySQL. Una vez creada, se definirán varias operaciones de modificación del esquema, para comprobar el correcto funcionamiento de la herramienta. En este contexto se distinguen dos modalidades de uso de Orion:

- **Modo creación** (SCRIPT MODE), en el que el script define tanto la construcción de todas las entidades cómo la creación de los atributos y relaciones que lo componen.
- **Modo evolución**, donde el script orion se aplica sobre un script Athena ya existente generado durante el modo creación o procedente de una evolución anterior.

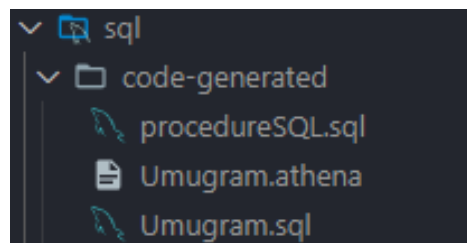


Figura 7.2: Scripts generados tras hacer uso del generador

Tal como ilustra la figura 7.2, al ejecutar un script en cualquiera de estos modos, el motor de Orion produce tres salidas complementarias:

1. El scripts de migración propiamente dicho, escritos en el lenguaje destino (en este escenario, MySQL), listo para aplicarse sobre la base de datos.
2. El modelo de esquema resultante expresado en Athena, que actúa como un manifiesto de la versión actual del diagrama lógico. Dependiendo de la procedencia del esquema Athena (creación previa o modificación del esquema ya existente) el número de la versión diferirá.

3. Los procedimientos almacenados auxiliares utilizados durante la ejecución de las operaciones de migración.

Con el fin de minimizar la curva de aprendizaje, las instrucciones de Orion han sido diseñadas siguiendo una sintaxis descriptiva muy similar a la de MySQL, aunque más simplificada. De este modo, al escribir un `ADD ENTITY User ...` o un `RENAME ATTR Post::caption TO title` el desarrollador reconoce inmediatamente la familiaridad de la forma y el propósito de cada comando.

Es importante volver a mencionar que, internamente, Orion maneja en memoria un esquema Athena completo (existente o no) antes de ejecutar cualquier operación de evolución. Por tanto, si deseamos modificar únicamente una parte del esquema —por ejemplo, renombrar una entidad o eliminar un atributo— tendremos que asegurarnos de que dicha entidad y sus atributos ya estén presentes en ese esquema inicial. En la práctica, eso significa que, en un script de Orion, primero debemos importar el esquema Athena de la base de datos actual o incluir operaciones de Orion que simulen su creación. Si se intentara ejecutar un comando de modificación contra un elemento no declarado previamente, el generador detectaría que no existe en el modelo Athena cargado y la transformación fallaría. En esta situación, la solución sería utilizar un esquema Athena con la información de las entidades asociadas a manipular dentro del script orion mediante la línea de importación `USING` (ejem: `USING Umugram:1`) donde el número final indica qué versión actual es la que se requiere del esquema.

Código 7.3: Script Orion para añadir Post

```
1 ADD ENTITY Post: {  
2   +post_id: String,  
3   caption: String,  
4   description: String,  
5   created_at: Timestamp,  
6   updated_at: Timestamp  
7 }  
8 ADD REF Post::user_id : String& TO User
```

Si queremos añadir una nueva entidad, la operación de creación no permite establecer referencias, por tanto se debe realizar por separado. En el fragmentos de código 7.3 y 7.4 se muestra las instrucciones necesarias para crear la entidad Post y la salida generada en MySQL. Al ejecutar este script con el motor *modelo-a-texto* de Orion se produce, entre otras salidas, el siguiente SQL para MySQL:

Código 7.4: SQL generado para MySQL de la entidad Post

```
1 CREATE TABLE IF NOT EXISTS POST  
2 (  
3   caption VARCHAR(100),  
4   created_at TIMESTAMP,  
5   description VARCHAR(100),
```

```

6  post_id VARCHAR(100),
7  updated_at TIMESTAMP,
8  CONSTRAINT post_pk PRIMARY KEY (post_id)
9);
10 ALTER TABLE POST ADD COLUMN user_id VARCHAR(100);
11 ALTER TABLE POST ADD CONSTRAINT post_user_id_fk
12 FOREIGN KEY (user_id) REFERENCES USER(user_id);

```

Internamente, tras procesar la operación, la instantánea en memoria de la entidad Post dentro del esquema Athena queda como se comenta en el código 7.5

Código 7.5: Fragmento en memoria de Post en el esquema .athena

```

1 root entity Post {
2   +post_id: String,
3   caption: String,
4   description: String,
5   created_at: Timestamp,
6   updated_at: Timestamp,
7   user_id: ref<UserUmugram as String>&
8 }

```

En el Anexo C se incorpora el fichero completo de pruebas, que incluye tanto el script Orion íntegro como la salida SQL y la instantánea del esquema Athena para este ejemplo. En él se realizan como ejemplo todas las operaciones ofrecidas por parte de la taxonomía de Orion comentadas en la tabla 5.1 en el Capítulo 5.

Partiendo del esquema actual que hemos creado para la base de datos, se plantea un escenario en el que deseamos conservar un histórico completo de cada publicación (Post) para poder consultar versiones anteriores, auditar cambios o, en su caso, deshacer modificaciones. Partiendo del esquema actual que hemos creado para la base de datos, se plantea un escenario en el que deseamos conservar un histórico completo de cada publicación (Post) para poder consultar versiones anteriores, auditar cambios o, en su caso, deshacer modificaciones.

Código 7.6: Operación promote del atributo revision en Post

```

1 // ORION CODE
2 ADD ATTR Post::revision : int
3 PROMOTE ATTR Post::revision
4
5 // SQL GENERATED
6 ALTER TABLE POST ADD COLUMN revision INTEGER;
7 CREATE TABLE IF NOT EXISTS POST_NEW
8 (
9   caption VARCHAR(100),
10  created_at TIMESTAMP,
11  description VARCHAR(100),

```

```
12 post_id VARCHAR(100),
13 revision VARCHAR(100),
14 updated_at TIMESTAMP,
15 user_id VARCHAR(100),
16 CONSTRAINT user_id_fk FOREIGN KEY(user_id) REFERENCES USER(user_id),
17 CONSTRAINT post_new_pk PRIMARY KEY (post_id, revision));
18 INSERT INTO POST_NEW SELECT * FROM POST;
19 CALL update_fk_schema("POST", "post_id", "revision");
20 CALL update_fk_data("POST", "post_id", "revision");
21 DROP TABLE POST;
22 ALTER TABLE POST_NEW RENAME TO POST;
```

Un inconveniente que cabe destacar es que este flujo puede fallar si ya existiera una tabla con el mismo nombre que la tabla temporal que se va a crear, si algún valor no cumpliera las nuevas restricciones NOT NULL o si hubiera inconsistencias en los datos heredados.

Por ello, aunque todas las operaciones de Orion han sido cuidadosamente diseñadas y probadas en múltiples situaciones, se recomienda que quien ejecute las migraciones tenga conocimientos sólidos de SQL y del modelo relacional destino. De este modo podrá anticipar y resolver conflictos en caso de que la situación pudiera darse.

7.2.2. Integración continua en Umugram

El entorno completo de Umugram, con todas las tecnologías descritas (Java, Spring Boot, Spring Data JPA y MySQL), se ha publicado en el repositorio `Zerep0/TFG` <https://github.com/Zerep0/TFG>, donde también se incluyen los distintos pipelines de integración continua, la herramienta M2T de orion a codeQL y la herramienta M2T de orion a MySQL. A partir de esta base, ejecutaremos dos operaciones de Orion sobre el esquema actual y, para cada una de ellas, mostraremos los resultados obtenidos mediante el análisis estático de CodeQL: primero simularemos la opción de “Cancelar” (no aplicar los cambios) y, después, la de “Aceptar” (generar y aplicar el script de migración), de forma que pueda verificarse cómo queda registrado el historial de versiones en el fichero ‘athena’ y en el control de migraciones.

Para ilustrar el flujo de “Cancelar”, partimos de la decisión de retirar completamente la funcionalidad de publicaciones en *Umugram*, quedando solo la capa de mensajería. Siguiendo las buenas prácticas de versionado de migraciones—donde cada script comienza con un prefijo de versión y un nombre claro que describe su propósito— el desarrollador nombra el archivo de operaciones de Orion de forma explícita, por ejemplo `remove_all_post_entities.orion`. De esta manera, en caso de ser aceptado se le añadirá al principio el prefijo con la versión obteniendo así un nombre similar al obtenido en la herramienta Flyway. Este sería el contenido del script:

```
1 Umugram operations
```

```
2  SCRIPT MODE
3  DELETE ENTITY Post
4  DELETE ENTITY VideoPost
5  DELETE ENTITY PhotoPost
6  DELETE ENTITY LivePost
```

Una vez se ha creado el archivo. Desde la ruta raíz se añade como un nuevo cambio y se debe realizar un commit indicando en la cabecera del mensaje explícitamente solo el nombre del script orion con el que fue nombrado:

```
1 git add remove_all_post_entities.orion
2 git commit -m "remove_all_post_entities.orion"
3 git push origin main
```

En la Figura 7.3 se ilustra el proceso que se dispara tras el `push`: por un lado, el flujo de versionado de esquema (que no llega a ejecutarse al no coincidir el nombre del script con la convención establecida), y por otro, el análisis de CodeQL, cuyos resultados aparecen en la sección de code-scanning mostrando las partes del código potencialmente afectadas.

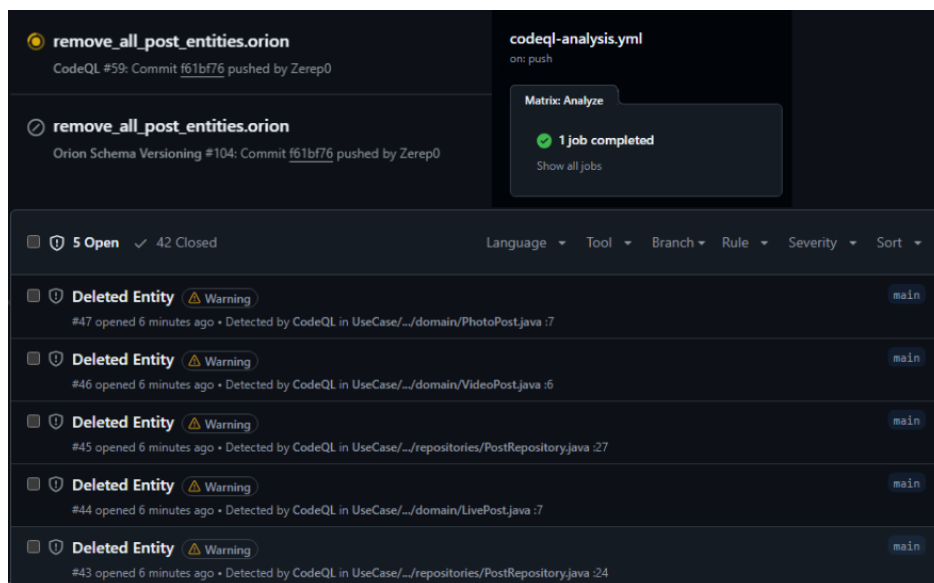


Figura 7.3: Workflow codeql tras recibir el script orion

Las consultas internamente, tal y como observamos en la imagen 7.4 tienen una ruta que especifica la ubicación exacta de la línea que se debe corregir. Esto se debe a que en la elaboración de la consulta se hace uso del campo `location` que extrae la ruta local del fichero en donde se encuentre ubicado.

Una de las consultas detectadas surge del código generado 7.7 para localizar consultas



Figura 7.4: Alertas mostradas en la sección code-scanning de Github

con la anotación `@Query`.

Código 7.7: Consulta CodeQL para localizar una consulta con anotación `@Query`

```

1 from Class entity, Location usageLoc, string message
2 where
3   entity.hasName("PhotoPost") and
4   isEntity(entity) and
5   exists(Annotation nq, Annotation q |
6     (isQuery(q) and
7       usesOldEntity(q.getValue("value"), entity) and
8       usageLoc = q.getTarget().getLocation() and
9       message = "Query uses entity '" + entity.getName() + "' which is ←
          ↪ marked for deletion."))
10 select usageLoc, message

```

Llegados a este punto, el desarrollador reconsidera la eliminación de las entidades de publicación y opta por mantenerlas. Para indicar al pipeline que se cancela la migración, basta con realizar un *commit* vacío que dé por concluida la operación de Orion, especificando su nombre tal como fue definido en el script original:

```

1 git commit --allow-empty
2           -m "orion(cancel): remove_all_post_entities.orion"

```

A continuación, consideremos un segundo escenario en el que la aplicación decide prescindir únicamente del servicio de streaming y, por tanto, eliminar la entidad `LivePost`. Siguiendo el mismo flujo de trabajo anterior —generación de un script descriptivo (esta vez, nombrado como `delete_live_post.orion`), commit con su nombre y push para disparar el análisis— el desarrollador decide aceptar los cambios:


```

1 git commit --allow-empty
2 -m "orion(accept): delete_live_post.orion, Se ha borrado la clase ↔
   ↔ LivePost"

```

De este modo, el sistema registra automáticamente cada nueva versión del script conforme a la convención de nombres de Flyway: el archivo se renombra a `V0_delete_live_post.orion` y se almacena en la carpeta de migraciones. La figura 7.5 muestra el historial después de ejecutar varios pasos de migración: cada entrada se añade de forma cronológica con su identificador de versión y una descripción clara de la operación realizada. La primera entrada corresponde al ejemplo inicial y, posteriormente, se incorporaron dos migraciones adicionales.

Version	Fichero .orion	Descripción	Fecha	Commit SHA
0	V0_delete_live_post.orion	Se ha borrado la clase LivePost	2025-05-27 20:14 UTC	07da301
1	V1_rename_post_userpost.orion	Se ha renombrado la clase Post a UserPost	2025-05-28 09:35 UTC	3c1a8b9
2	V2_add_storypost.orion	Se ha añadido la clase StoryPost	2025-05-29 11:42 UTC	c9e4f22

Figura 7.5: Tabla de versiones vista desde Github tras aceptar diferentes scripts

Anteriormente se había indicado que, en el proceso de *code-scanning* de GitHub Actions, cada vez que se analizaba un script de Orion se generaba un conjunto de hallazgos o “alerts” en la sección de resultados estáticos. Teniendo ese aspecto en cuenta, la primera ejecución mostrará todas las inconsistencias detectadas por CodeQL a partir de las operaciones definidas en el script, pero en análisis posteriores, no volverán a aparecer, ya que el motor de escaneo las considera antiguas. Para forzar su re-evaluación es necesario que el desarrollador haya realizado cambios reales en el código, de modo que el sistema las trate como nuevas. Una vez que aparece una alerta, el usuario tiene tres opciones sobre cada hallazgo:

- **Ignorar**, dejando la alerta pendiente para futuras revisiones.
- **Descartar como prueba** (i.e., marcarla como “false positive” o caso de prueba), para que no vuelva a mostrarse en la historia de alertas.
- **Marcar como reparado**, lo cual coincide con la resolución definitiva del problema y hace que la alerta desaparezca de los resultados activos.

En este sentido, el flujo de validación de cambios se mantiene limpio y únicamente se notifican nuevas incidencias tras modificaciones efectivas en el código fuente. Una vez corregido el código y aplicado el script de migración, resulta conveniente volver a ejecutar el flujo de análisis para verificar si siguen apareciendo nuevas alertas en

la sección de *code-scanning*. Basta con volver a hacer `git push` tras subir el mismo script Orion y revisar los resultados en GitHub. Si finalmente comprobamos que no hay nuevas notificaciones, es posible anular su efecto realizando un commit vacío con la sintaxis habitual de cancelación o borrar el fichero manualmente.

8. Conclusiones y líneas futuras

8.1. Conclusiones Generales

Este proyecto ha completado los resultados de la tesis de Alberto Hernández [1] añadiendo soporte relacional al lenguaje Orion de modo que se ha demostrado cómo el metamodelo unificado **U-Schema**, junto con este lenguaje, permite abstraer por completo la evolución de esquemas, de modo que el mismo conjunto de operaciones —añadir, eliminar, renombrar, mover, promover, entre otras— funciona indistintamente sobre bases de datos relacionales y NoSQL.

Por otra parte, el caso de uso de Umugram ha permitido validar que se ha diseñado e implementado una estrategia de integración continua orientada a la adaptación del código. Gracias a OrionFlow, las consultas obsoletas o las inconsistencias derivadas de un script de evolución se detectan de forma inmediata, señalando con precisión las líneas de código afectadas. Además, se incorpora un mecanismo de trazabilidad mediante una tabla de control que registra el historial de versiones, siguiendo un enfoque similar al utilizado por herramientas como Flyway o Liquibase.

8.2. Limitaciones del enfoque y trabajo futuro

8.2.1. Limitaciones de la transformación con Orion

Aunque Orion ofrece un conjunto completo de operaciones para la evolución de esquemas, no todas son triviales de aplicar en escenarios reales. Operaciones como **Promote** —que convierte un atributo en clave primaria— o **Split** —que transforma tablas en subtablas con columnas extraídas— implican lógicas complejas, como el recorrido de todas las tablas relacionadas para actualizar sus claves foráneas. En entornos con esquemas heterogéneos, datos inconsistentes o ausencia de índices adecuados, estas transformaciones pueden fallar o generar resultados inesperados. Por ello, y como se destacó en el capítulo anterior, aunque la herramienta está diseñada para usuarios, se recomienda su uso a desarrolladores con experiencia previa en modelado y migración de bases de datos, capaces de anticipar y resolver posibles conflictos de integridad.

Orion funciona junto con Athena y requiere de un esquema previo para realizar modificaciones suponiendo una tarea extra para el desarrollador. Para atenuar esta curva de aprendizaje, se ha empleado un enfoque de few-shot prompting sobre ChatGPT descrito en un artículo conjunto con el grupo de investigación ModelUM [28]. Median-

te unas pocas decenas de ejemplos representativos y la propia gramática de Athena (grammar prompting), se entrenó un modelo de lenguaje genérico para que reconozca y genere automáticamente el fichero `.athena` correspondiente a partir de descripciones en lenguaje natural o de fragmentos de bases de datos relacionales o NoSQL. De este modo, el desarrollador solo necesita formular prompts sencillos y delega en el LLM la inferencia y validación del esquema Athena, pudiendo centrarse exclusivamente en definir las operaciones de evolución sin verse obligado a dominar todos los detalles sintácticos de Athena.

8.2.2. Trabajo futuro sobre la herramienta predictora

En futuras iteraciones de la herramienta, se prevé abordar estas carencias y ampliar su cobertura de la siguiente manera:

1. **Tabla de versionado enriquecida:** Actualmente sólo soportamos migraciones con prefijo `V` (al estilo Liquibase/Flyway). Se contempla incorporar los indicadores `U` (undo) y `R` (redo) para facilitar migraciones bidireccionales y escenarios de rollback precisos.
 2. **Sugerencias de actualización automática:** Actualmente el sistema se limita a señalar los elementos del código que podrían verse afectados por una evolución del esquema. Sin embargo, en algunas operaciones como los renombramientos, sería posible ir más allá y proponer una actualización directa del código, sugiriendo por ejemplo el nuevo nombre a utilizar.
 3. **Integración de DataFlow y TaintTracking en CodeQL:** DataFlow permite seguir el recorrido de datos entre métodos y consultas, mientras que TaintTracking identifica flujos de datos inseguros. Su inclusión ayudaría a detectar “residuos” o usos inseguros de parámetros en consultas JPQL/SQL que escapan al análisis sintáctico. Aunque ambos enfoques fueron evaluados, los tiempos de análisis se incrementaron significativamente, por lo que quedaron descartados en esta versión. Será necesario realizar estudios comparativos de rendimiento en bases de código a gran escala para determinar si su uso resulta rentable en entornos industriales.
 4. **Compatibilidad con anotaciones legacy:** La versión actual detecta anotaciones Jakarta EE (`jakarta.persistence.*`). Ampliaríamos el reconocimiento a `javax.persistence.*`.
-

Bibliografía

- [1] Alberto Hernández Chillón. *A Multi-Model Environment for NoSQL and Relational Systems*. PhD thesis, Faculty of Informatics. University of Murcia, Spain, 2022.
- [2] Pramod Sadalage and Martin Fowler. *Evolutionary Database Design*, 2016. URL <https://martinfowler.com/articles/evodb.html>.
- [3] Github. Github Official Website. <https://github.com/>, 2025. Accessed June 2025.
- [4] Gitlab. Gitlab Official Website. <https://about.gitlab.com/es/>, 2025. Accessed June 2025.
- [5] Flyway. Flyway Official Website. <https://flywaydb.org/>, 2025. Accessed June 2025.
- [6] Liquibase. Liquibase Official Website. <https://www.liquibase.org/>, 2025. Accessed June 2025.
- [7] Pramodkumar J. Sadalage Martin Fowler. *NoSQL Distilled. A Brief Guide to the Emerging World of Polyglot Persistence*. Pearson, 1 edition, 2009.
- [8] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 2012.
- [9] Carlos J. Fernández Candel, Diego Sevilla Ruiz, and Jesús J. García-Molina. A unified metamodel for nosql and relational databases. *Information Systems*, 104: 101898, 2022. doi: <https://doi.org/10.1016/j.is.2021.101898>.
- [10] Alberto Hernández Chillón, Diego Sevilla Ruiz, and Jesús García Molina. Athena: A database-independent schema definition language. In Iris Reinhartz-Berger and Shazia Sadiq, editors, *Advances in Conceptual Modeling - ER 2021 Workshops CoMoNoS, EmpER, CMLS, St. John's, NL, Canada, October 18-21, 2021, Proceedings*, volume 13012 of *Lecture Notes in Computer Science*, pages 33–42. Springer, 2021. URL https://doi.org/10.1007/978-3-030-88358-4_4.
- [11] Alberto Hernández Chillón, Diego Sevilla Ruiz, and Jesús García Molina. Deimos: A model-based nosql data generation language. In Georg Grossmann and Sudha

-
- Ram, editors, *Advances in Conceptual Modeling - ER 2020 Workshops CMAI, CMLS, CMOMM4FAIR, CoMoNoS, EmpER, Vienna, Austria, November 3-6, 2020, Proceedings*, volume 12584 of *Lecture Notes in Computer Science*, pages 151–161. Springer, 2020. doi: 10.1007/978-3-030-65847-2_14.
- [12] Alberto Hernández Chillón, Meike Klettke, Diego Sevilla Ruiz, and Jesús García Molina. A generic schema evolution approach for nosql and relational databases. *IEEE Trans. Knowl. Data Eng.*, 36(7):2774–2789, 2024. doi: 10.1109/TKDE.2024.3362273.
- [13] Craig Walls. *Spring Boot in Action*. Manning Publications, 1 edition, December 2015. ISBN 9781617292545. Foreword by Andrew Glover; printed in black & white.
- [14] Sun Microsystems. Página principal de JPA. <http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html>. Último acceso: 28-06-2025.
- [15] Cătălin Tudose. *Java Persistence with Spring Data and Hibernate*. Manning Publications, expanded edition, February 2023. ISBN 978-1617299186. Forewords by Dmitry Aleksandrov and Mohamed Taman; printed in black & white.
- [16] SonarQube. SonarQube Official Website. <https://www.sonarsource.com/products/sonarqube/>, 2025. Accessed June 2025.
- [17] Alberto Hernández Chillón, Jesús García Molina, José Ramón Hoyos, and María-José Ortín-Ibáñez. Propagating schema changes to code: An approach based on a unified data model. In George Fletcher and Verena Kantere, editors, *Proceedings of the Workshops of the EDBT/ICDT 2023 Joint Conference, Ioannina, Greece, March, 28, 2023*, volume 3379 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2023. URL https://ceur-ws.org/Vol-3379/CoMoNoS_2023_id251_Alberto_Hernandez_Chillon.pdf.
- [18] CodeQL. Github. codeql, 2025. URL <https://github.com/github/codeql>. Accessed June 2025.
- [19] Priscila Heller. *Automating Workflows with GitHub Actions: Automate software development workflows and seamlessly deploy your applications using GitHub Actions*. Packt Publishing, 1 edition, November 2021. ISBN 978-1800569034.
- [20] MySQL. MySQL Official Website. <https://www.mysql.com/>, 2025. Accessed June 2025.
- [21] Carlos Javier Fernández Candel. *A Unified Data Metamodel for Relational and NoSQL databases: Schema Extraction and Query*. PhD thesis, Faculty of Informatics, University of Murcia, Murcia, Spain, 2022.
-

-
- [22] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2009.
- [23] Carlo Curino, Hyun Jin Moon, Alin Deutsch, and Carlo Zaniolo. Automating the database schema evolution process. In *The VLDB Journal*, volume 22, pages 73–98, 2013.
- [24] Jean-Marc Hick and Jean-Luc Hainaut. Strategy for database application evolution: The DB-MAIN approach. In *International Conference on Conceptual Modeling*, pages 291–306. Springer, 2003.
- [25] Jérôme Fink, Maxime Gobert, and Anthony Cleve. Adapting Queries to Database Schema Changes in Hybrid Polystores. In *20th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2020, Adelaide, Australia, September 28 - October 2, 2020*, pages 127–131. IEEE, 2020. doi: 10.1109/SCAM51674.2020.00019. URL <https://doi.org/10.1109/SCAM51674.2020.00019>.
- [26] Uta Störl and Meike Klettke. Darwin: A Data Platform for Schema Evolution Management and Data Migration. In *DataPlat 2022: 1st International Workshop on Data Platform Design, Management and Optimization*, 2022.
- [27] Xtext Main Webpage, 2012. URL <https://eclipse.dev/Xtext/>. Accessed: May 2025.
- [28] Antonio Pérez-Serrano, Jesús Joaquín García Molina, María-José Ortín-Ibañez, and José Ramón Hoyos Barceló. Una exploración sobre cómo dsl existentes pueden beneficiarse de llm. JISBD2025, sep 2023.
- [29] Alberto Hernández Chillón, Diego Sevilla Ruiz, and Jesús García Molina. Athena: A database-independent schema definition language. In Iris Reinhartz-Berger and Shazia Sadiq, editors, *Advances in Conceptual Modeling - ER 2021 Workshops CoMoNoS, EmpER, CMLS, St. John's, NL, Canada, October 18-21, 2021, Proceedings*, volume 13012 of *Lecture Notes in Computer Science*, pages 33–42. Springer, 2021. URL https://doi.org/10.1007/978-3-030-88358-4_4.
- [30] ISO/IEC. Information Technology – Database Languages SQL – Part 1: Framework (SQL/Framework). Technical Report ISO/IEC 9075-1:2023, International Organization for Standardization, 2023.
- [31] Carlos Javier Fernández Candel, Diego Sevilla Ruiz, and Jesús Joaquín García Molina. A unified metamodel for nosql and relational databases. *Information Systems*, 104:101898, 2022. ISSN 0306-4379. doi: 10.1016/j.is.2021.101898.
-

- [32] Meike Klettke, Uta Störl, Manuel Shenavai, and Stefanie Scherzinger. NoSQL Schema Evolution and Big Data Migration at Scale. In *IEEE International Conference on Big Data*. IEEE Computer Society, 2016.
-

A. Anexo I

A.1. Flyway

Flyway permite gestionar la evolución de esquemas de bases de datos mediante scripts SQL versionados. Cada script se ejecuta una única vez y queda registrado en una tabla de control llamada `flyway_schema_history`¹. A continuación se muestra un ejemplo de estructura típica en un proyecto:

Código A.1: Script de creación de tabla con Flyway

```
1 -- V1__create_users_table.sql
2 CREATE TABLE users (
3     id SERIAL PRIMARY KEY,
4     username VARCHAR(50) NOT NULL,
5     email VARCHAR(100),
6     created_at TIMESTAMP DEFAULT NOW()
7 );
```

Cada script debe colocarse en una carpeta especial ('db/migration') y seguir la convención de nombres `V<versión>__<descripción>.sql`. Flyway ejecutará los scripts en orden ascendente de versión.

Los aspectos más relevantes son:

- **Nombre y esquema.** Por defecto la tabla se llama `flyway_schema_history` y se crea en el `schema` por defecto. Ambos se pueden cambiar con las propiedades `table` y `defaultSchema`.²
- **Auditoría completa.** Cada fila es la huella de una migración: quién, cuándo y con qué código se alteró la BD. Jamás se edita a mano; si se corrompe se usa `flyway repair`.
- **Integridad mediante *checksum*.** En cada `flyway validate` se comprueba que el checksum almacenado coincide con el del fichero en disco; de no ser así se marca como *Failed/Missing* y se bloquea el despliegue hasta repararlo.

¹<https://www.red-gate.com/hub/product-learning/flyway/exploring-the-flyway-schema-history-table>

²<https://documentation.red-gate.com/fd/flyway-schema-history-table-273973417.html>

- **Múltiples estados.** La combinación de `installed_rank` y `success` permite a Flyway derivar estados como *Pending*, *Success*, *Failed*, *Ignored*, *Out of Order*, etc., que se ven en `flyway info`.
- **Bloqueo de concurrencia.** Antes de escribir en la tabla, Flyway coloca un *lock* interno (no es otra tabla) para evitar ejecuciones paralelas que pudieran desordenar `installed_rank`.

A continuación se simula el contenido de la tabla de control que mantiene Flyway mostrando los cambios más relevantes que registra:

<code>installed_rank</code>	<code>ver.</code>	<code>desc.</code>	<code>type</code>	<code>script</code>	<code>installed_by</code>	<code>installed_on</code>	<code>execution_time</code>
1	1	Initial Build	SQL	V1__add_Build.sql	A. García	2025-06-10 11:32	506
2	2	Pubs Original Data	SQL	V2__Pubs_Original.sql	A. García	2025-06-10 11:35	1581

Tabla A.1: Ejemplo parcial de la tabla `flyway_schema_history`

Flyway no genera código automáticamente ni analiza el impacto en el código fuente, por lo que su uso es más adecuado en entornos controlados donde se gestionan los cambios manualmente.

A.2. Liquibase

Liquibase gestiona la evolución del esquema mediante archivos *changelog* en formato XML, YAML o JSON. Cada bloque de cambios, denominado `<changeSet>`, se identifica por los atributos `id` y `author`, lo que permite controlar la ejecución y, en caso necesario, la reversión de los scripts.

El código A.2 muestra un *changelog* típico que:

1. Establece una **precondición** para que la migración la ejecute el usuario `liquibase`.
2. **Crea** la tabla `person` con las columnas `id`, `firstname`, `lastname` y `state`.
3. **Añade** la columna `username` a la tabla `person`.
4. **Genera** la tabla de referencia `state` (lookup table) a partir de los valores existentes en la columna `state`.

Código A.2: Ejemplo de *changelog* Liquibase

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <databaseChangeLog
3     xmlns="http://www.liquibase.org/xml/ns/dbchangelog"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

```

```

5      xmlns:ext="http://www.liquibase.org/xml/ns/dbchangelog-ext"
6      xmlns:pro="http://www.liquibase.org/xml/ns/pro"
7      xsi:schemaLocation="
8          http://www.liquibase.org/xml/ns/dbchangelog
9          http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-latest↵
10         ↵ .xsd
11         http://www.liquibase.org/xml/ns/dbchangelog-ext
12         http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-ext.↵
13         ↵ xsd
14         http://www.liquibase.org/xml/ns/pro
15         http://www.liquibase.org/xml/ns/pro/liquibase-pro-latest.xsd">
16
17 <!-- 1. Precondición: la migración debe ejecutarla el usuario '↵
18 ↵ liquibase' -->
19 <preConditions>
20     <runningAs username="liquibase"/>
21 </preConditions>
22
23 <!-- 2. Creación de la tabla 'person' -->
24 <changeSet id="1" author="nvoxland">
25     <createTable tableName="person">
26         <column name="id" type="int" autoIncrement="true">
27             <constraints primaryKey="true" nullable="false"/>
28         </column>
29         <column name="firstname" type="varchar(50)"/>
30         <column name="lastname" type="varchar(50)">
31             <constraints nullable="false"/>
32         </column>
33         <column name="state" type="char(2)"/>
34     </createTable>
35 </changeSet>
36
37 <!-- 3. Adición de la columna 'username' -->
38 <changeSet id="2" author="nvoxland">
39     <addColumn tableName="person">
40         <column name="username" type="varchar(8)"/>
41     </addColumn>
42 </changeSet>
43
44 <!-- 4. Creación de la tabla de referencia 'state' -->
45 <changeSet id="3" author="nvoxland">
46     <addLookupTable
47         existingTableName="person" existingColumnName="state"
48         newTableName="state" newColumnName="id"
49         newColumnDataType="char(2)"/>

```

```

47 </changeSet>
48
49 </databaseChangeLog>

```

Liquibase mantiene el historial de las migraciones en la tabla **DATABASECHANGELOG** (DBCL). Si la tabla no existe, Liquibase la crea automáticamente antes de aplicar el primer *changeset*. Los aspectos más relevantes son:

- **Nombre:** por defecto **DATABASECHANGELOG**³, modificable con `--database-changelog-table-name`.
- **Identificación de filas:** combinación de los campos **ID**, **AUTHOR** y **FILENAME**. No hay clave primaria para evitar límites de longitud en algunos SGBD, pero la combinación de los tres campos es única.
- **Propósito principal:** registrar la ejecución de cada *changeset*, su orden, su fecha y su resultado (**EXECUTED**, **FAILED**, **RERAN...**), además de metadatos como checksum, etiquetas, contexto, versión de Liquibase, etc.
- **Histórico completo:** desde Liquibase Pro 4.27.0 se recomienda la tabla **DATABASECHANGELOGHISTORY**⁴, que preserva cambios múltiples sobre un mismo *changeset*.

Una simulación simplificada de la estructura y algunos registros de **DATABASECHANGELOG** se muestra en la A.2. Para brevedad se omiten columnas menos usadas y se incluyen valores de ejemplo.

ID	AUTHOR	FILENAME	ORD. EXEC.	FECHA EXEC.	EXEC TYPE	MD5SUM
1	nvoxland	db/changelog.xml	1	2025-05-29 10:11	EXECUTED	9f7a2...
2	nvoxland	db/changelog.xml	2	2025-05-29 10:12	EXECUTED	4b15e...
3	nvoxland	db/changelog.xml	3	2025-05-29 10:13	EXECUTED	a7c44...

Tabla A.2: Simulación de la tabla **DATABASECHANGELOG**

La ventaja de Liquibase frente a Flyway es su soporte para múltiples formatos, funciones como rollback y generación de diffs automáticos entre esquemas. Sin embargo, tampoco detecta el impacto de los cambios en el código fuente, a diferencia de la solución planteada en este trabajo.

³<https://docs.liquibase.com/concepts/tracking-tables/databasechangelog-table.html>

⁴<https://docs.liquibase.com/concepts/tracking-tables/databasechangeloghistory-table.html>

B. Anexo II

B.1. Plantillas de texto en Xtend (*String Templates*)

Las *string templates*¹ de Xtend son el elemento clave que simplifica la generación de código en la transformación M2T. Se declaran entre comillas simples triples `''' ... '''` y permiten incrustar expresiones Xtend dentro del texto mediante los delimitadores `«` y `»`. De este modo se evita la concatenación manual con `+` típica de Java, ganando legibilidad y reduciendo errores de escape.

Sintaxis básica

Código B.1: Ejemplo mínimo de plantilla

```
1 val tableName = "PLAYER"
2 val sql = '''
3     CREATE TABLE «tableName»_LOG (
4         id INT AUTO_INCREMENT PRIMARY KEY,
5         old_state JSON,
6         ts TIMESTAMP DEFAULT CURRENT_TIMESTAMP
7     );
8 '''
```

- El texto se mantiene tal cual; las variables o expresiones entre `« ... »` se evalúan y el resultado se inserta en la cadena final.
- Las plantillas son *multilínea*, por lo que los saltos de línea y tabulaciones del propio archivo Xtend se preservan en la salida.
- Al compilarse a Java, Xtend optimiza la plantilla en una única `StringBuilder`, evitando penalizaciones de rendimiento.

Expresiones de control

Los bloques `«FOR ... ENDFOR»`, `«IF ... ENDIF»` y `«SWITCH ... ENDSWITCH»` permiten introducir lógica condicional o iterativa sin salir de la plantilla.

¹https://eclipse.dev/Xtext/xtend/documentation/203_xtend_expressions.html#templates

Código B.2: Iteración con FOR para tipos colección

```

1 '''
2 «FOR feat : features SEPARATOR "\n"»
3 -- tabla auxiliar para la colección «feat.name»
4 CREATE TABLE «entity.name.toUpperCase»_«feat.name.toUpperCase» (
5     owner_id INT REFERENCES «entity.name»(id),
6     item «feat.dbType»,
7     PRIMARY KEY (owner_id, item)
8 );
9 «ENDFOR»
10 '''

```

Además, Xtend recorta la sangría común de la plantilla para que el texto resultante no herede los espacios del código fuente. Esto permite mantener las plantillas correctamente indentadas dentro del método sin alterar la salida SQL final. Los listados B.3 y B.4 muestran plantillas más complejas empleadas en la práctica para renombrar tablas y crear procedimientos almacenados, respectivamente.

B.2. Ejemplo de operacion dispatch

A continuación se documenta la función `generateBasicOp` escrita en Xtend. Su misión es traducir una operación del modelo Orion `EntityRenameOp` en sentencias SQL de `RENAME TABLE` que afectan tanto a la tabla “raíz” de la entidad como a las tablas de colección que dependen de ella.

Código B.3: Generación de SQL a partir de `EntityRenameOp`

```

1 private def dispatch generateBasicOp(EntityRenameOp op) '''
2     «val schema =
3         aHandler.getSchemaTypeDecl(schemaUpdater.schema, op.spec.ref)»
4     «val features = aHandler
5         .getFeaturesInSchemaType(schema)
6         .filter[f | f instanceof SimpleFeature
7             && isCollection((f as SimpleFeature).type)]
8         .map[f | f as SimpleFeature]»
9
10    «FOR feat : features SEPARATOR "\n"»
11        RENAME TABLE «op.spec.ref.toUpperCase + "_" +
12            feat.name.toUpperCase»
13        TO «op.spec.name.toUpperCase + "_" +
14            feat.name.toUpperCase»;
15    «ENDFOR»
16
17    RENAME TABLE «op.spec.ref.toUpperCase»

```

```

18 TO «op.spec.name.toUpperCase»;
19 ' ' '

```

Explicación paso a paso

1. Obtención del esquema de la entidad

`aHandler.getSchemaTypeDecl(...)` localiza en el *metamodelo Athena* el tipo concreto de la entidad cuyo nombre antiguo se almacena en `op.spec.ref`.

2. Detección de colecciones hijas

Mediante `getFeaturesInSchemaType` se listan los *features* de la entidad. Después se filtran sólo aquellos que son de tipo `SimpleFeature`, y representan colecciones (`isCollection`). Una colección es detectada como válida si se trata de un agregado o de los tipos especiales proporcionados por Athena: Set, Map y Tuple. El resultado se castea de nuevo a `SimpleFeature` para disponer de sus propiedades.

3. Bucle FOR interno

Por cada colección se genera una sentencia `RENAME TABLE` que cambia el nombre de la tabla *ENTIDAD_OLD_FEATURE* por *ENTIDAD_NEW_FEATURE*.

```
RENAME TABLE PLAYER_SCORES → USERPLAYER_SCORES;
```

4. Renombrado de la tabla principal

Fuera del bucle se emite el `RENAME TABLE` final para la propia entidad:

```
RENAME TABLE PLAYER → USERPLAYER;
```

Con este patrón, cualquier renombrado de entidad queda reflejado de forma consistente tanto en la tabla raíz como en todas sus tablas de colección.

B.3. Procedimientos auxiliares

Aquí se adjunta el código de los procedimientos auxiliares generados al realizar una ejecución de la transformación M2T de Orion a MySQL.

Código B.4: Procedimientos generados por la transformación M2T Orion-MySQL

```

1 -- Procedures to help in the sql operations
2
3 -- Procedure for dropping weak entities, used in EntityDeleteOp ↵
  ↵ operations
4
5 DELIMITER $$
6

```

```

7 CREATE PROCEDURE dropWeakEntities(IN parent_table VARCHAR(255))
8 BEGIN
9   -- (1) Create temporary table if it does not exist
10  CREATE TEMPORARY TABLE IF NOT EXISTS tmp_fks (cmd VARCHAR(1024));
11
12  -- (2) Insert statements to drop weak entities
13  INSERT INTO tmp_fks (cmd)
14    SELECT DISTINCT CONCAT('DROP TABLE IF EXISTS ', TABLE_NAME, ';')
15    FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE
16    WHERE REFERENCED_TABLE_NAME = parent_table
17       AND TABLE_SCHEMA = DATABASE()
18       AND TABLE_NAME IN (
19         SELECT TABLE_NAME
20         FROM INFORMATION_SCHEMA.TABLE_CONSTRAINTS
21         WHERE CONSTRAINT_TYPE = 'PRIMARY KEY'
22         AND TABLE_SCHEMA = DATABASE()
23       );
24
25  -- (3) Call the helper procedure to execute the commands and clean up ↩
26  ↩ the temporary table
27  CALL executeCommand();
28 END$$
29
30 DELIMITER ;
31
32 -- Procedure for dropping all foreign keys declared in a given table
33 DELIMITER $$
34
35 CREATE PROCEDURE dropFKsInTable(IN tabla_objetivo VARCHAR(255))
36 BEGIN
37   -- (1) Create temporary table if it does not exist
38   CREATE TEMPORARY TABLE IF NOT EXISTS tmp_fks (cmd VARCHAR(1024));
39
40   -- (2) Clear any existing commands in the temporary table
41   TRUNCATE TABLE tmp_fks;
42
43   -- (3) Insert commands to drop all foreign keys on the target table
44   INSERT INTO tmp_fks (cmd)
45     SELECT DISTINCT
46       CONCAT('ALTER TABLE ', TABLE_NAME, ' DROP FOREIGN KEY ', ↩
47       ↩ CONSTRAINT_NAME, ';')
48     FROM INFORMATION_SCHEMA.TABLE_CONSTRAINTS
49     WHERE CONSTRAINT_TYPE = 'FOREIGN KEY'
50        AND TABLE_SCHEMA = DATABASE()

```



```

50     AND TABLE_NAME = tabla_objetivo;
51
52 -- (3) Execute the generated commands and clean up
53 CALL executeCommand();
54 END$$
55
56 DELIMITER ;
57
58
59 -- Procedure for dropping foreign keys from a table, used in ↵
    ↵ EntityDeleteOp operations
60
61 DELIMITER $$
62
63 CREATE PROCEDURE dropFKsFromTable(IN parent_table VARCHAR(255))
64 BEGIN
65 -- (1) Create temporary table if it does not exist
66 CREATE TEMPORARY TABLE IF NOT EXISTS tmp_fks (cmd VARCHAR(1024));
67
68 -- (2) Clear any existing commands in the temporary table
69 TRUNCATE TABLE tmp_fks;
70
71 -- (3) Insert statements to drop foreign keys
72 INSERT INTO tmp_fks (cmd)
73     SELECT DISTINCT CONCAT('ALTER TABLE ', TABLE_NAME, ' DROP FOREIGN ↵
        ↵ KEY ', CONSTRAINT_NAME, ';')
74     FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE
75     WHERE REFERENCED_TABLE_NAME = parent_table
76         AND TABLE_SCHEMA = DATABASE();
77
78 -- (3) Call the helper procedure to execute the commands and clean up ↵
        ↵ the temporary table
79 CALL executeCommand();
80
81 END$$
82
83 DELIMITER ;
84
85
86 -- Helper procedure to execute dynamic SQL commands from the temporary ↵
        ↵ table
87
88 DELIMITER $$
89
90 CREATE PROCEDURE executeCommand()

```

```

91 BEGIN
92   DECLARE done INT DEFAULT 0;
93   DECLARE v_cmd VARCHAR(1024);
94   DECLARE cur CURSOR FOR SELECT cmd FROM tmp_fks;
95   DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
96
97   -- (1) Open the cursor and iterate over commands
98   OPEN cur;
99   read_loop: LOOP
100     FETCH cur INTO v_cmd;
101     IF done THEN
102       LEAVE read_loop;
103     END IF;
104
105     SET @sql_exec = v_cmd;
106     PREPARE st FROM @sql_exec;
107     EXECUTE st;
108     DEALLOCATE PREPARE st;
109   END LOOP;
110   CLOSE cur;
111
112   -- (2) Drop the temporary table after use
113   DROP TEMPORARY TABLE IF EXISTS tmp_fks;
114   SET done = 0;
115
116 END$$
117
118 DELIMITER ;
119
120
121 -- Procedure for dropping constraints for a column, used in ↩
122   ↪ FeatureDeleteOp operations
123
124
125 DELIMITER $$
126
127 CREATE PROCEDURE dropConstraintsForColumn(
128   IN p_table_name VARCHAR(255),
129   IN p_column_name VARCHAR(255)
130 )
131 BEGIN
132   DECLARE done INT DEFAULT 0;
133   DECLARE v_constraint_name VARCHAR(255);
134   DECLARE v_constraint_type VARCHAR(64);
135
136   -- (1) Cursor selects ALL CONSTRAINTS names for the given table and ↩

```

```

    ↪ column
135 DECLARE cur CURSOR FOR
136     SELECT kcu.CONSTRAINT_NAME, tc.CONSTRAINT_TYPE
137     FROM INFORMATION_SCHEMA.TABLE_CONSTRAINTS tc
138     JOIN INFORMATION_SCHEMA.KEY_COLUMN_USAGE kcu
139         ON tc.CONSTRAINT_NAME = kcu.CONSTRAINT_NAME
140         AND tc.TABLE_NAME = kcu.TABLE_NAME
141         AND tc.CONSTRAINT_SCHEMA = kcu.CONSTRAINT_SCHEMA
142     WHERE kcu.TABLE_NAME = p_table_name
143         AND kcu.COLUMN_NAME = p_column_name
144         AND kcu.TABLE_SCHEMA = DATABASE();
145
146 DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
147
148 -- (2) Open cursor and loop through each found constraint
149 OPEN cur;
150 read_loop: LOOP
151     FETCH cur INTO v_constraint_name, v_constraint_type;
152     IF done THEN
153         LEAVE read_loop;
154     END IF;
155
156     SET @sql = CONCAT('ALTER TABLE ', p_table_name, ' DROP CONSTRAINT ', ↪
    ↪ v_constraint_name, ';');
157     PREPARE stmt FROM @sql;
158     EXECUTE stmt;
159     DEALLOCATE PREPARE stmt;
160 END LOOP;
161 CLOSE cur;
162 END$$
163
164 DELIMITER ;
165
166
167 -- Procedure for dropping check constraints for a column, used in ↪
    ↪ FeatureRenameOp operations
168
169 DELIMITER $$
170
171 CREATE PROCEDURE dropCheckConstraintsForColumn(
172     IN p_table_name VARCHAR(255),
173     IN p_column_name VARCHAR(255)
174 )
175 BEGIN
176     DECLARE done INT DEFAULT 0;

```

```

177 DECLARE v_check_name VARCHAR(255);
178
179 -- (1) Cursor selects ALL CHECK constraint names for the given table ↵
    ↵ and column
180 DECLARE cur CURSOR FOR
181     SELECT tc.CONSTRAINT_NAME
182     FROM INFORMATION_SCHEMA.TABLE_CONSTRAINTS tc
183     JOIN INFORMATION_SCHEMA.CHECK_CONSTRAINTS cc ON cc.CONSTRAINT_NAME = ↵
    ↵ tc.CONSTRAINT_NAME
184     JOIN INFORMATION_SCHEMA.COLUMNS c ON c.TABLE_NAME = tc.TABLE_NAME ↵
    ↵ AND c.TABLE_SCHEMA = tc.CONSTRAINT_SCHEMA
185     WHERE tc.CONSTRAINT_TYPE = 'CHECK'
186           AND tc.TABLE_NAME = p_table_name
187           AND c.COLUMN_NAME = p_column_name
188           AND tc.CONSTRAINT_SCHEMA = DATABASE();
189
190 DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
191
192 -- (2) Open cursor and loop through each found constraint
193 OPEN cur;
194 read_loop: LOOP
195     FETCH cur INTO v_check_name;
196     IF done THEN
197         LEAVE read_loop;
198     END IF;
199
200     -- Build and execute the DROP CHECK statement for this constraint
201     SET @sql = CONCAT('ALTER TABLE ', p_table_name, ' DROP CHECK ', ↵
    ↵ v_check_name, ';');
202     PREPARE stmt FROM @sql;
203     EXECUTE stmt;
204     DEALLOCATE PREPARE stmt;
205 END LOOP;
206 CLOSE cur;
207 END$$
208
209 DELIMITER ;
210
211
212 -- Procedure for updating foreign key schema, used in ↵
    ↵ AttributePromoteOp operations
213
214 DELIMITER $$
215
216 CREATE PROCEDURE update_fk_schema(

```

```

217 IN p_promoted_table VARCHAR(64),
218 IN p_old_pk VARCHAR(255),
219 IN p_add_pk VARCHAR(255)
220 )
221 BEGIN
222     -- Declare all variables at the beginning
223     DECLARE done INT DEFAULT FALSE;
224     DECLARE v_table_name VARCHAR(64);
225     DECLARE v_constraint_name VARCHAR(64);
226     DECLARE v_fk_columns VARCHAR(255);
227     DECLARE v_num INT;
228     DECLARE v_index INT DEFAULT 1;
229     DECLARE v_add_col VARCHAR(64);
230     DECLARE col_type VARCHAR(255);
231
232     -- Cursor to group the FKs that reference p_promoted_table
233     DECLARE cur CURSOR FOR
234         SELECT TABLE_NAME, CONSTRAINT_NAME, GROUP_CONCAT(COLUMN_NAME ↵
                ↵ ORDER BY ORDINAL_POSITION) as fk_cols
235         FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE
236         WHERE REFERENCED_TABLE_NAME = p_promoted_table
237         GROUP BY TABLE_NAME, CONSTRAINT_NAME;
238
239     DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;
240
241     OPEN cur;
242     fk_loop: LOOP
243         FETCH cur INTO v_table_name, v_constraint_name, v_fk_columns;
244         IF done THEN
245             LEAVE fk_loop;
246         END IF;
247
248         -- Calculate the number of additional columns to add
249         SET v_num = (LENGTH(p_add_pk) - LENGTH(REPLACE(p_add_pk, ',', '' ↵
                ↵ )) + 1);
250         SET v_index = 1;
251
252         WHILE v_index <= v_num DO
253             SET v_add_col = TRIM(SUBSTRING_INDEX(SUBSTRING_INDEX(p_add_pk ↵
                ↵ , ',', v_index), ',', -1));
254
255             -- If the additional column is not found in the current FK, ↵
                ↵ add it to the table
256             IF FIND_IN_SET(v_add_col, v_fk_columns) = 0 THEN
257

```

```

258      -- (1) Build SELECT ... INTO @col_type to get the column ↵
259      ↵ type from the referenced table
260      SET @colTypeSql = CONCAT(
261          "SELECT COLUMN_TYPE INTO @col_type ",
262          "FROM INFORMATION_SCHEMA.COLUMNS ",
263          "WHERE TABLE_SCHEMA = DATABASE() ",
264          " AND TABLE_NAME = '", p_promoted_table, "' ",
265          " AND COLUMN_NAME = '", v_add_col, "' ",
266          "LIMIT 1"
267      );
268
269      -- (2) Execute using PREPARE/EXECUTE
270      PREPARE stmt FROM @colTypeSql;
271      EXECUTE stmt;
272      DEALLOCATE PREPARE stmt;
273
274      -- (3) Assign the value from the user variable @col_type ↵
275      ↵ to the local variable col_type
276      SET col_type = @col_type;
277
278      -- (4) Use col_type in the ALTER TABLE statement
279      SET @sql = CONCAT('ALTER TABLE ', v_table_name,
280          ' ADD COLUMN ', v_add_col, ' ', col_type);
281      PREPARE stmt FROM @sql;
282      EXECUTE stmt;
283      DEALLOCATE PREPARE stmt;
284      END IF;
285
286      SET v_index = v_index + 1;
287      END WHILE;
288
289      -- Drop the old FK.
290      SET @sql = CONCAT('ALTER TABLE ', v_table_name, ' DROP FOREIGN ↵
291      ↵ KEY ', v_constraint_name);
292      PREPARE stmt FROM @sql;
293      EXECUTE stmt;
294      DEALLOCATE PREPARE stmt;
295
296      -- Create the new FK that uses both the original and additional ↵
297      ↵ columns.
298      SET @sql = CONCAT(
299          'ALTER TABLE ', v_table_name,
300          ' ADD CONSTRAINT fk_', v_table_name, '_', p_promoted_table, ' ↵
301          ↵ _new FOREIGN KEY (' ,
302          p_old_pk, ' , ', p_add_pk,

```

```

298         ') REFERENCES ', p_promoted_table, '_NEW (' ,
299         p_old_pk, ', ', p_add_pk, ') '
300     );
301     PREPARE stmt FROM @sql;
302     EXECUTE stmt;
303     DEALLOCATE PREPARE stmt;
304 END LOOP;
305 CLOSE cur;
306 END$$
307
308 DELIMITER ;
309
310
311 -- Procedure for updating foreign key data, used in AttributePromoteOp ←
312     ↪ operations
313
314 DELIMITER $$
315
316 CREATE PROCEDURE update_fk_data(
317     IN p_promoted_table VARCHAR(64), -- E.g.: 'PLAYER1'
318     IN p_old_pk VARCHAR(255), -- E.g.: 'player_id'
319     IN p_add_pk VARCHAR(255) -- E.g.: 'score,level'
320 )
321 BEGIN
322     DECLARE done INT DEFAULT FALSE;
323     DECLARE v_table_name VARCHAR(64);
324     DECLARE v_constraint_name VARCHAR(64);
325     DECLARE v_fk_columns VARCHAR(255);
326
327     DECLARE v_join_condition VARCHAR(1000) DEFAULT '';
328     DECLARE v_set_clause VARCHAR(1000) DEFAULT '';
329
330     DECLARE v_num INT;
331     DECLARE v_index INT DEFAULT 1;
332     DECLARE v_old_col VARCHAR(64);
333     DECLARE v_add_col VARCHAR(64);
334
335     -- Cursor: finds all the FKs referencing the new table
336     DECLARE cur CURSOR FOR
337         SELECT TABLE_NAME, CONSTRAINT_NAME, GROUP_CONCAT(COLUMN_NAME ←
338             ↪ ORDER BY ORDINAL_POSITION) AS fk_cols
339         FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE
340         WHERE REFERENCED_TABLE_NAME = CONCAT(p_promoted_table, '_NEW')
341         GROUP BY TABLE_NAME, CONSTRAINT_NAME;

```

```

341 DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;
342
343 OPEN cur;
344 fk_loop: LOOP
345     FETCH cur INTO v_table_name, v_constraint_name, v_fk_columns;
346     IF done THEN
347         LEAVE fk_loop;
348     END IF;
349
350     -- 1) Build the JOIN condition from the old columns
351     SET v_join_condition = '';
352     SET v_index = 1;
353     SET v_num = (LENGTH(p_old_pk) - LENGTH(REPLACE(p_old_pk, ',', ''↵
↵ )) + 1);
354
355     WHILE v_index <= v_num DO
356         SET v_old_col = TRIM(SUBSTRING_INDEX(SUBSTRING_INDEX(p_old_pk↵
↵ , ',', v_index), ',', -1));
357         IF v_index = 1 THEN
358             SET v_join_condition = CONCAT('r.', v_old_col, ' = p.', ↵
↵ v_old_col);
359         ELSE
360             SET v_join_condition = CONCAT(v_join_condition, ' AND r.'↵
↵ , v_old_col, ' = p.', v_old_col);
361         END IF;
362         SET v_index = v_index + 1;
363     END WHILE;
364
365     -- 2) Build the SET clause with the new columns to copy
366     SET v_set_clause = '';
367     SET v_index = 1;
368     SET v_num = (LENGTH(p_add_pk) - LENGTH(REPLACE(p_add_pk, ',', ''↵
↵ )) + 1);
369
370     WHILE v_index <= v_num DO
371         SET v_add_col = TRIM(SUBSTRING_INDEX(SUBSTRING_INDEX(p_add_pk↵
↵ , ',', v_index), ',', -1));
372         IF v_index = 1 THEN
373             SET v_set_clause = CONCAT('r.', v_add_col, ' = p.', ↵
↵ v_add_col);
374         ELSE
375             SET v_set_clause = CONCAT(v_set_clause, ', r.', v_add_col↵
↵ , ' = p.', v_add_col);
376         END IF;
377         SET v_index = v_index + 1;

```



```

378     END WHILE;
379
380     -- 3) Execute the UPDATE with JOIN, copying the data from the ↵
381     ↵ _NEW table
382     SET @sql = CONCAT(
383         'UPDATE ', v_table_name, ' r ',
384         'JOIN ', p_promoted_table, '_NEW p ON ', v_join_condition, ' ↵
385         ↵ ',
386         'SET ', v_set_clause
387     );
388
389     PREPARE stmt FROM @sql;
390     EXECUTE stmt;
391     DEALLOCATE PREPARE stmt;
392
393     END LOOP;
394
395     CLOSE cur;
396
397     END$$
398
399     DELIMITER ;
400
401
402 -- Procedure for casting columns with foreign keys, used in ↵
403     ↵ ReferenceCastOp operations
404
405     DELIMITER $$
406
407     CREATE PROCEDURE castColumnFKs(
408         IN p_table_name VARCHAR(255),
409         IN p_column_name VARCHAR(255),
410         IN p_new_type VARCHAR(255)
411     )
412     BEGIN
413         DECLARE done INT DEFAULT 0;
414         DECLARE ref_table VARCHAR(255);
415         DECLARE ref_column VARCHAR(255);
416
417         -- Cursor to find columns that reference p_table_name.p_column_name
418         DECLARE ref_cur CURSOR FOR
419             SELECT TABLE_NAME, COLUMN_NAME
420             FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE
421             WHERE TABLE_SCHEMA = DATABASE()
422                 AND REFERENCED_TABLE_NAME = p_table_name
423                 AND REFERENCED_COLUMN_NAME = p_column_name;

```

```
420 -- Handler for when the cursor reaches the end
421 DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;
422
423 -- 1) Disable FK checks for this session
424 SET @sql = 'SET FOREIGN_KEY_CHECKS = 0';
425 PREPARE stmt FROM @sql;
426 EXECUTE stmt;
427 DEALLOCATE PREPARE stmt;
428
429 -- 2) Cast the original column
430 SET @sql = CONCAT(
431     'ALTER TABLE `', p_table_name,
432     '` MODIFY COLUMN `', p_column_name,
433     '` ', p_new_type, ';'
434 );
435 PREPARE stmt FROM @sql;
436 EXECUTE stmt;
437 DEALLOCATE PREPARE stmt;
438
439 -- 3) Cast the columns that reference it
440 OPEN ref_cur;
441 read_loop: LOOP
442     FETCH ref_cur INTO ref_table, ref_column;
443     IF done THEN
444         LEAVE read_loop;
445     END IF;
446
447     -- Cast the referencing column to the same type
448     SET @sql = CONCAT(
449         'ALTER TABLE `', ref_table,
450         '` MODIFY COLUMN `', ref_column,
451         '` ', p_new_type, ';'
452     );
453     PREPARE stmt FROM @sql;
454     EXECUTE stmt;
455     DEALLOCATE PREPARE stmt;
456 END LOOP;
457 CLOSE ref_cur;
458
459 -- Reset 'done' in case it is used later
460 SET done = 0;
461
462 -- 4) Re-enable FOREIGN_KEY_CHECKS
463 SET @sql = 'SET FOREIGN_KEY_CHECKS = 1';
464 PREPARE stmt FROM @sql;
```

```
465 EXECUTE stmt;  
466 DEALLOCATE PREPARE stmt;  
467 END$$  
468  
469 DELIMITER ;
```

C. Anexo III

Aquí se muestran tanto los ficheros `.orion` usados en las pruebas como los resultados (*scripts* MySQL) generados por la herramienta M2T Orion-MySQL.

C.1. Scripts orion

El objetivo de estos *scripts* es hacer uso de la mayor parte de las operaciones que actualmente soporta el prototipo: creación, borrado, renombrado, casteo de atributos, *promote* de campos, etc. Además, cada caso incluye sentencias `INSERT` opcionales para verificar la correcta inserción y posterior extracción de datos.

Código C.1: Script 1 – Operaciones *Umugram*

```
1 Umugram operations
2
3 SCRIPT MODE
4
5 // This script is used to perform operations on Umugram
6
7 // CREATE OPERATIONS
8
9 // CREATE A USER
10 ADD ENTITY User: {
11     +user_id: String,
12     username: String,
13     !email: String /^.+@.+\.com$/,
14     password: String,
15     created_at: Timestamp,
16     updated_at: Timestamp
17 }
18
19 /*
20 -- Data for use in extract
21 INSERT INTO USER (created_at, email, password, updated_at, user_id, ↵
    ↵ username) VALUES
22 ('2024-01-15 08:23:45', 'alice.smith@example.com', 'P@ssw0rd1', ↵
    ↵ '2024-03-01 12:00:00', 'u001', 'alice_smith'),
23 ('2024-02-10 14:05:12', 'bob.johnson@example.com', 'Secr3tKey!', ↵
```

```

    ↪ '2024-03-02 09:30:15', 'u002', 'bobj'),
24 ('2024-03-05 19:47:30', 'carla98@example.com', 'MyPa$$123', '2024-03-05 ↪
    ↪ 20:00:00', 'u003', 'carla98'),
25 ('2024-03-20 11:12:00', 'daniel.k@example.com', 'Qwerty!234', ↪
    ↪ '2024-03-21 08:45:00', 'u004', 'danielk'),
26 ('2024-04-01 07:00:00', 'eva.green@example.com', 'Green$Eva56', ↪
    ↪ '2024-04-02 16:20:00', 'u005', 'evagreen');

27
28 */
29
30 // EXTRACT METADATA OF USER
31 EXTRACT ENTITY User INTO User_Metadata(user_id, created_at, updated_at)
32
33
34 // CREATE A PROFILE
35 ADD AGGR User::profile: {
36     nameProfile: String,
37     avatar_url: String,
38     description: String,
39     website: String
40 }&
41
42 // CREATE RELATIONSHIP FOLLOWERS, NOT VALID WITHOUT AT LEAST ONE FIELD
43 ADD RELATIONSHIP Followers
44
45 // CREATE TABLE POST
46 ADD ENTITY Post: {
47     +post_id: String,
48     caption: String,
49     description: String,
50     created_at: Timestamp,
51     updated_at: Timestamp
52 }
53
54 // CREATE TABLE VIDEOPOST
55 ADD ENTITY VideoPost: {
56     +videopost_id: String,
57     videoUrl: String,
58     duration: double
59 }
60
61 // CREATE TABLE PHOTOPOST
62 ADD ENTITY PhotoPost: {
63     +photopost_id: String,
64     photoUrl: String

```

```
65 }
66
67 // CREATE TABLE LIVEPOST
68 ADD ENTITY LivePost: {
69     +realpost_id: String,
70     +duration : Timestamp
71 }
72
73 // CREATE TABLE COMMENT
74 ADD ENTITY Comment: {
75     +comment_id: String,
76     text: String,
77     created_at: Timestamp,
78     updated_at: Timestamp
79 }
80
81 // ADD REFERENCES
82 ADD REF Post::user_id : String& TO User
83 ADD REF VideoPost::post_id : String& TO Post
84 ADD REF PhotoPost::post_id : String& TO Post
85 ADD REF Comment::user_id : String& TO User
86 ADD REF Comment::replyTo : String& TO Comment
87 ADD REF Comment::post_id : String& TO Post
88
89 // ADD RELATIONSHIP REFERENCES
90 ADD REF Followers::follower_id : String& TO User
91 ADD REF Followers::following_id : String& TO User
92
93 // CAST ATTR DURATION
94 CAST ATTR VideoPost::duration TO int
95
96 // RENAME ATTR
97 RENAME VideoPost::duration TO seconds
98
99 // ADD ATTRS
100 ADD ATTR User::birthday : Timestamp
101 ADD ATTR VideoPost::lastMinute: Timestamp
102 ADD ATTR Comment::hashtag : List<String>
103 ADD ATTR Post::revision : int
104
105 // PROMOTE ATTR
106 PROMOTE ATTR Post::revision
107
108 // RENAME ENTITY
109 RENAME ENTITY User TO UserUmugram
```

```

110
111 // DELETE ENTITY
112 DELETE ENTITY LivePost

```

Código C.2: Script 2 – Operaciones avanzadas *Umugram*

```

1 Umugram operations
2
3 SCRIPT MODE
4
5 /*
6 INSERT INTO USER (created_at, email, password, updated_at, user_id, ↵
    ↵ username) VALUES
7 ('2024-01-15 08:23:45', 'alice.smith@example.com', 'P@ssw0rd1', ↵
    ↵ '2024-03-01 12:00:00', 'u001', 'alice_smith'),
8 ('2024-02-10 14:05:12', 'bob.johnson@example.com', 'Secr3tKey!', ↵
    ↵ '2024-03-02 09:30:15', 'u002', 'bobj'),
9 ('2024-03-05 19:47:30', 'carla98@example.com', 'MyPa$$123', '2024-03-05 ↵
    ↵ 20:00:00', 'u003', 'carla98'),
10 ('2024-03-20 11:12:00', 'daniel.k@example.com', 'Qwerty!234', ↵
    ↵ '2024-03-21 08:45:00', 'u004', 'danielk'),
11 ('2024-04-01 07:00:00', 'eva.green@example.com', 'Green$Eva56', ↵
    ↵ '2024-04-02 16:20:00', 'u005', 'evagreen');
12 */
13
14 // CREATE A USER
15 ADD ENTITY User: {
16     +user_id: String,
17     username: String,
18     !email: String /^.+@.+\.com$/,
19     password: String,
20     created_at: Timestamp,
21     updated_at: Timestamp
22 }
23
24 // EXTRACT METADATA OF USER
25 SPLIT ENTITY User INTO
26     User_Metadata(user_id, created_at, updated_at),
27     User_Account(user_id, username, email, password)
28
29 // COPY USERNAME TO USER_METADATA
30 COPY User_Account::username TO User_Metadata::username
31 WHERE user_id = user_id

```


C.2. Código MySQL generado

Además de los procedimientos auxiliares, la transformación genera el script SQL completo que materializa todas las operaciones definidas en cada fichero `.orion`. Estos son los resultados de las ejecuciones usando como entrada los scripts mencionados en la sección anterior:

??:

Código C.3: Script SQL generado para el script 1 C.1

```

1 CREATE DATABASE IF NOT EXISTS umugram;
2
3 USE umugram;
4
5 /* ADD ENTITY User: {
6   +user_id: String, username: String,
7   !email: String /^.+@.+\.\com$/, password: String, created_at: Timestamp, updated_at: Timestamp↵
8     ↵ }
9 */
10 CREATE TABLE IF NOT EXISTS USER
11 (
12   created_at TIMESTAMP,
13   email VARCHAR(100) CHECK (email REGEXP '^.+@.+\.\com$'),
14   password VARCHAR(100),
15   updated_at TIMESTAMP,
16   user_id VARCHAR(100),
17   username VARCHAR(100),
18   CONSTRAINT user_email_ak UNIQUE(email),
19   CONSTRAINT user_pk PRIMARY KEY (user_id)
20 );
21
22 /* EXTRACT ENTITY User INTO User_Metadata ( user_id, created_at, updated_at )
23
24 */
25 -- (1) Crear nueva tabla con los atributos extraídos
26 CREATE TABLE IF NOT EXISTS USER_METADATA (
27   user_id VARCHAR(100),
28   created_at TIMESTAMP,
29   updated_at TIMESTAMP,
30   CONSTRAINT user_metadata_pk PRIMARY KEY (user_id)
31 );
32
33 -- (2) Copiar datos a la nueva tabla
34 INSERT INTO USER_METADATA (user_id, created_at, updated_at)
35 SELECT user_id, created_at, updated_at FROM USER;
36
37 /* ADD AGGR User::profile: { nameProfile: String, avatar_url: String, description: String, ↵
38     ↵ website: String
39 }&
40 */
41 CREATE TABLE IF NOT EXISTS USER_PROFILE
42 (
43   user_id VARCHAR(100),
44   nameprofile VARCHAR(100),
45   avatar_url VARCHAR(100),
46   description VARCHAR(100),
47   website VARCHAR(100),
48   CONSTRAINT profile_pk PRIMARY KEY(user_id),
49   CONSTRAINT user_profile_fk FOREIGN KEY(user_id) REFERENCES USER(user_id)
50 );
51
52 /* ADD RELATIONSHIP Followers

```

```

53
54 */
55 -- YOU NEED ADD AT LEAST ONE REFERENCE TO GENERATE THE TABLE
56
57 /* ADD ENTITY Post: {
58   +post_id: String, caption: String, description: String, created_at: Timestamp, updated_at: ↵
59   ↵ Timestamp }
60 */
61 CREATE TABLE IF NOT EXISTS POST
62 (
63   caption VARCHAR(100),
64   created_at TIMESTAMP,
65   description VARCHAR(100),
66   post_id VARCHAR(100),
67   updated_at TIMESTAMP,
68   CONSTRAINT post_pk PRIMARY KEY (post_id)
69 );
70
71 /* ADD ENTITY VideoPost: {
72   +videopost_id: String, videoUrl: String, duration: double
73 }
74 */
75 CREATE TABLE IF NOT EXISTS VIDEOPOST
76 (
77   duration DOUBLE,
78   videourl VARCHAR(100),
79   videopost_id VARCHAR(100),
80   CONSTRAINT videopost_pk PRIMARY KEY (videopost_id)
81 );
82
83 /* ADD ENTITY PhotoPost: {
84   +photopost_id: String, photoUrl: String
85 }
86 */
87 CREATE TABLE IF NOT EXISTS PHOTOPOST
88 (
89   photourl VARCHAR(100),
90   photopost_id VARCHAR(100),
91   CONSTRAINT photopost_pk PRIMARY KEY (photopost_id)
92 );
93
94 /* ADD ENTITY LivePost: {
95   +realpost_id: String,
96   +duration : Timestamp }
97 */
98 CREATE TABLE IF NOT EXISTS LIVEPOST
99 (
100   duration TIMESTAMP,
101   realpost_id VARCHAR(100),
102   CONSTRAINT livepost_pk PRIMARY KEY (realpost_id, duration)
103 );
104
105 /* ADD ENTITY Comment: {
106   +comment_id: String, text: String, created_at: Timestamp, updated_at: Timestamp }
107 */
108 CREATE TABLE IF NOT EXISTS COMMENT
109 (
110   comment_id VARCHAR(100),
111   created_at TIMESTAMP,
112   text VARCHAR(100),
113   updated_at TIMESTAMP,
114   CONSTRAINT comment_pk PRIMARY KEY (comment_id)
115 );
116
117 /* ADD REF Post::user_id : String& TO User
118 */

```

```
119-- (1) Multiplicity ? o &: Add reference column at actual table
120ALTER TABLE POST
121ADD COLUMN user_id VARCHAR(100);
122
123-- (2) Add foreign key
124ALTER TABLE POST
125ADD CONSTRAINT post_user_id_fk FOREIGN KEY (user_id)
126REFERENCES USER(user_id);
127
128
129/* ADD REF VideoPost::post_id : String& TO Post
130 */
131-- (1) Multiplicity ? o &: Add reference column at actual table
132ALTER TABLE VIDEOPOST
133ADD COLUMN post_id VARCHAR(100);
134
135-- (2) Add foreign key
136ALTER TABLE VIDEOPOST
137ADD CONSTRAINT videopost_post_id_fk FOREIGN KEY (post_id)
138REFERENCES POST(post_id);
139
140
141/* ADD REF PhotoPost::post_id : String& TO Post
142 */
143-- (1) Multiplicity ? o &: Add reference column at actual table
144ALTER TABLE PHOTOPOST
145ADD COLUMN post_id VARCHAR(100);
146
147-- (2) Add foreign key
148ALTER TABLE PHOTOPOST
149ADD CONSTRAINT photopost_post_id_fk FOREIGN KEY (post_id)
150REFERENCES POST(post_id);
151
152
153/* ADD REF Comment::user_id : String& TO User
154 */
155-- (1) Multiplicity ? o &: Add reference column at actual table
156ALTER TABLE COMMENT
157ADD COLUMN user_id VARCHAR(100);
158
159-- (2) Add foreign key
160ALTER TABLE COMMENT
161ADD CONSTRAINT comment_user_id_fk FOREIGN KEY (user_id)
162REFERENCES USER(user_id);
163
164
165/* ADD REF Comment::replyTo : String& TO Comment
166 */
167-- (1) Multiplicity ? o &: Add reference column at actual table
168ALTER TABLE COMMENT
169ADD COLUMN replyto VARCHAR(100);
170
171-- (2) Add foreign key
172ALTER TABLE COMMENT
173ADD CONSTRAINT comment_replyto_fk FOREIGN KEY (replyto)
174REFERENCES COMMENT(comment_id);
175
176
177/* ADD REF Comment::post_id : String& TO Post
178 */
179-- (1) Multiplicity ? o &: Add reference column at actual table
180ALTER TABLE COMMENT
181ADD COLUMN post_id VARCHAR(100);
182
183-- (2) Add foreign key
184ALTER TABLE COMMENT
185ADD CONSTRAINT comment_post_id_fk FOREIGN KEY (post_id)
```

```

186 REFERENCES POST(post_id);
187
188
189 /* ADD REF Followers::follower_id : String& TO User
190 */
191 CREATE TABLE IF NOT EXISTS FOLLOWERS (
192     follower_id VARCHAR(100),
193     CONSTRAINT follower_id_pk PRIMARY KEY (follower_id),
194     CONSTRAINT followers_follower_id_fk FOREIGN KEY (follower_id) REFERENCES USER(user_id)
195 );
196
197 /* ADD REF Followers::following_id : String& TO User
198
199 */
200 -- (1) Multiplicity ? o &: Add reference column at actual table
201 ALTER TABLE FOLLOWERS
202 ADD COLUMN following_id VARCHAR(100);
203
204 -- (2) Add foreign key
205 ALTER TABLE FOLLOWERS
206 ADD CONSTRAINT followers_following_id_fk FOREIGN KEY (following_id)
207 REFERENCES USER(user_id);
208
209 CALL dropFKsInTable("FOLLOWERS");
210 ALTER TABLE FOLLOWERS DROP PRIMARY KEY;
211
212
213 ALTER TABLE FOLLOWERS ADD PRIMARY KEY (follower_id, following_id);
214
215 ALTER TABLE FOLLOWERS
216 ADD CONSTRAINT followers_following_id_fk FOREIGN KEY (following_id)
217 REFERENCES USER (user_id);
218
219 ALTER TABLE FOLLOWERS
220 ADD CONSTRAINT followers_follower_id_fk FOREIGN KEY (follower_id)
221 REFERENCES USER (user_id);
222
223 /* CAST ATTR VideoPost::duration TO int
224
225 */
226 ALTER TABLE VIDEOPOST MODIFY COLUMN duration INTEGER;
227
228 /* RENAME VideoPost::duration TO seconds
229
230 */
231 ALTER TABLE VIDEOPOST RENAME COLUMN duration TO seconds;
232
233 /* ADD ATTR User::birthday : Timestamp
234 */
235 ALTER TABLE USER ADD COLUMN birthday TIMESTAMP;
236
237 /* ADD ATTR VideoPost::lastMinute : Timestamp
238 */
239 ALTER TABLE VIDEOPOST ADD COLUMN lastminute TIMESTAMP;
240
241 /* ADD ATTR Comment::hashtag : List<String>
242 */
243 ALTER TABLE COMMENT ADD COLUMN hashtag JSON;
244
245 /* ADD ATTR Post::revision : int
246
247 */
248 ALTER TABLE POST ADD COLUMN revision INTEGER;
249
250 /* PROMOTE ATTR Post::revision
251
252 */

```

```

253 CREATE TABLE IF NOT EXISTS POST_NEW
254 (
255     caption VARCHAR(100),
256     created_at TIMESTAMP,
257     description VARCHAR(100),
258     post_id VARCHAR(100),
259     revision INTEGER,
260     updated_at TIMESTAMP,
261     user_id VARCHAR(100),
262     CONSTRAINT user_id_fk FOREIGN KEY(user_id) REFERENCES USER(user_id),
263     CONSTRAINT post_new_pk PRIMARY KEY (post_id, revision)
264 );
265 INSERT INTO POST_NEW SELECT * FROM POST;
266 CALL update_fk_schema("POST", "post_id", "revision");
267 CALL update_fk_data("POST", "post_id", "revision");
268 DROP TABLE POST;
269 ALTER TABLE POST_NEW RENAME TO POST;
270
271 /* RENAME ENTITY User TO UserUmugram
272
273 */
274 RENAME TABLE USER_PROFILE TO USERUMUGRAM_PROFILE;
275 RENAME TABLE USER TO USERUMUGRAM;
276
277 /* DELETE ENTITY LivePost
278 */
279 CALL dropWeakEntities("LIVEPOST");
280 CALL dropFKsFromTable("LIVEPOST");
281 DROP TABLE IF EXISTS LIVEPOST;
282
283 -- Delete helper procedures
284 DROP PROCEDURE IF EXISTS dropWeakEntities;
285 DROP PROCEDURE IF EXISTS dropFKsFromTable;
286 DROP PROCEDURE IF EXISTS dropFKsInTable;
287 DROP PROCEDURE IF EXISTS executeCommand;
288 DROP PROCEDURE IF EXISTS dropConstraintsForColumn;
289 DROP PROCEDURE IF EXISTS dropCheckConstraintsForColumn;
290 DROP PROCEDURE IF EXISTS update_fk_schema;
291 DROP PROCEDURE IF EXISTS update_fk_data;
292 DROP PROCEDURE IF EXISTS castColumnFKs;

```

Código C.4: Script SQL generado para el script 2 C.2

```

1 CREATE DATABASE IF NOT EXISTS umugram;
2
3 USE umugram;
4
5 /* ADD ENTITY User: {
6     +user_id: String, username: String,
7     !email: String /^.+@.+\.\com$/, password: String, created_at: Timestamp, updated_at: Timestamp ↵
8     ↵ }
9 */
10 CREATE TABLE IF NOT EXISTS USER
11 (
12     created_at TIMESTAMP,
13     email VARCHAR(100) CHECK (email REGEXP '^.+@.+\.\com$'),
14     password VARCHAR(100),
15     updated_at TIMESTAMP,
16     user_id VARCHAR(100),
17     username VARCHAR(100),
18     CONSTRAINT user_email_ak UNIQUE(email),
19     CONSTRAINT user_pk PRIMARY KEY (user_id)
20 );
21
22 /* SPLIT ENTITY User INTO User_Metadata ( user_id, created_at, updated_at ), User_Account ( ↵

```

```

    ↪ user_id, username, email, password )
23
24 */
25-- (1) Crear nueva tabla con los atributos extraídos
26CREATE TABLE IF NOT EXISTS USER_METADATA (
27    user_id VARCHAR(100),
28    created_at TIMESTAMP,
29    updated_at TIMESTAMP,
30    CONSTRAINT user_metadata_pk PRIMARY KEY (user_id)
31);
32
33-- (2) Copiar datos a la nueva tabla
34INSERT INTO USER_METADATA (user_id, created_at, updated_at)
35SELECT user_id, created_at, updated_at FROM USER;
36
37-- (1) Crear nueva tabla con los atributos extraídos
38CREATE TABLE IF NOT EXISTS USER_ACCOUNT (
39    user_id VARCHAR(100),
40    username VARCHAR(100),
41    email VARCHAR(100) CHECK (email REGEXP '^.+@.+\.\.com$'),
42    password VARCHAR(100),
43    CONSTRAINT user_account_pk PRIMARY KEY (user_id)
44);
45
46-- (2) Copiar datos a la nueva tabla
47INSERT INTO USER_ACCOUNT (user_id, username, email, password)
48SELECT user_id, username, email, password FROM USER;
49
50CALL dropWeakEntities("USER");
51CALL dropFKsFromTable("USER");
52DROP TABLE IF EXISTS USER;
53
54/* COPY User_Account::username TO User_Metadata::username WHERE user_id = user_id
55 */
56ALTER TABLE USER_METADATA ADD COLUMN username VARCHAR(100);
57
58UPDATE USER_METADATA tg
59JOIN USER_ACCOUNT sr ON tg.user_id = sr.user_id
60SET tg.username = sr.username;
61
62-- Delete helper procedures
63DROP PROCEDURE IF EXISTS dropWeakEntities;
64DROP PROCEDURE IF EXISTS dropFKsFromTable;
65DROP PROCEDURE IF EXISTS dropFKsInTable;
66DROP PROCEDURE IF EXISTS executeCommand;
67DROP PROCEDURE IF EXISTS dropConstraintsForColumn;
68DROP PROCEDURE IF EXISTS dropCheckConstraintsForColumn;
69DROP PROCEDURE IF EXISTS update_fk_schema;
70DROP PROCEDURE IF EXISTS update_fk_data;
71DROP PROCEDURE IF EXISTS castColumnFKs;

```