COMPUTERIZED AUTOMATIC SYSTEMS

DATABASE MIGRATION IN CODE-FREE FORMAT

Ulyana Dzelendzyak, PhD, As.-Prof., Danylo Nikulshyn, MS Student,
Andriy Pavelchak, PhD, As.-Prof.,
Lviv Polytechnic National University, Ukraine; e-mail: uliana.y.dzelendziak@lpnu.ua
Leonid Moroz, DSc, Prof.,
Cracow University of Technology, Poland

https://doi.org/

Abstract. Database migration is a critical process for businesses that need to maintain their data integrity and functionality as they transition to new technologies or systems [7]. The paper has researched the concept of database migration in a code-free format, including the benefits and challenges of this approach. Code-free migration tools provide an interface that allows users to drag and drop data and automate the migration process. This eliminates the need for manual coding and speeds up the migration process. Code-free solutions also provide users with visual representations of the data, making it easier to understand and manage. However, these tools require careful consideration of data quality and security to ensure that the migration process is successful.

The article provides different types of code-free migration tools, including data mapping software and drag-and-drop interfaces, and how they can be used to facilitate the migration process [3]. Best practices are provided for successful code-free database migration, including testing and validation.

Key words: Database migration, code-free format, representations of the data, web application, cloud technologies, Lambda function, cloud platform.

1. Introduction

As technology continues to evolve at a rapid pace, businesses are finding it increasingly important to maintain the integrity and functionality of their databases during migration to new systems or technologies. Traditional database migration methods can be complex and require technical expertise, which can pose a significant challenge for businesses that lack specialized IT departments.

Furthermore, traditional migration methods can be time-consuming and prone to errors, which can lead to data quality issues and potential security vulnerabilities. These challenges have made it difficult for businesses to migrate their databases efficiently and effectively, which can harm their operations and competitive advantage.

Code-free solutions have emerged as potential solutions to these challenges. The rise of these solutions has provided a simpler and more intuitive approach to database migration. Code-free migration tools provide an interface that allows users to automate the migration process and manage data more effectively. This eliminates the need for manual coding, which can save time and reduce the risk of errors.

2. Drawbacks

Finally, we highlight best practices for successful code-free database migration, including testing and validation. We conclude with some considerations for future research in this area, as the use of code-free solutions for database migration continues to grow in popularity.

3. Goal

The goal of the current article is the research benefits and challenges of codeless database migration solutions and guide best practices to ensure successful database migrations while maintaining data quality and security.

4. Migration Mechanism of Database

The main mechanism of any migrator is the recognition of data that have been migrating from one place to another. If we are considering relational databases, when everything is simpler, we can collect data about table schemas in the specified database with the help of SQL query [6].

For non-relational databases like MongoDB, DynamoDB, or Firestore we cannot act in the same way [1, 4]. Assuming we have a table with multiple columns, the schema can be derived as follows:

1. For the first column, we take the first element and evaluate its type using the parser method.
2. For the same column, we take the second element and evaluate its type using the parser method.
3. If the type of the second element matches the type of the first element, then we assume that this is the data type of the column.
4. If the types do not match, we continue to evaluate the types of subsequent elements in the column until we find the largest number of matches, which will be our type as a result.
5. We repeat steps 1-4 for all columns in the table.

Using this method, we can derive a schema for the table that reflects the data types of each column based on the data present in the table.

Loading data into a new database after deriving schemas for the tables can indeed be a straightforward process, especially if the new database is compatible with the data types identified in the schema. However, there are still some important considerations to keep in mind to ensure a successful data migration.

Firstly, it is important to ensure that the new database has been properly set up and configured to receive the data. This may involve creating new tables, setting up appropriate indexes and constraints, and configuring any necessary database parameters.

Secondly, it is important to carefully map the data from the old database to the new database, taking into account any differences in table structure, data types, and other factors that may affect the mapping process. This may involve writing scripts or using specialized data migration tools to ensure that the data is loaded correctly.

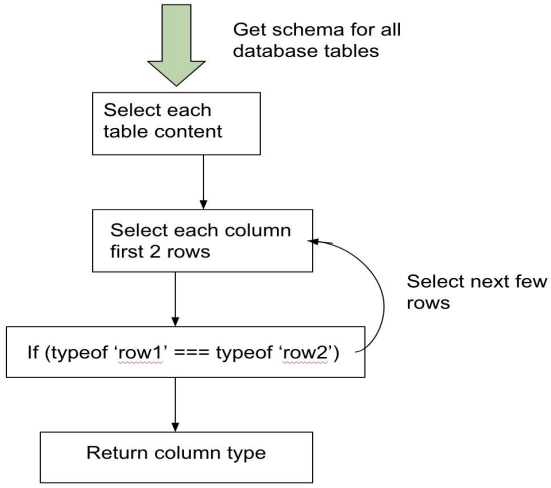

Fig. 1 A simplified graphic representation of the database migration algorithm

```
pgClient.query('SELECT table_name FROM information_schema.tables WHERE table_schema = \'' + pgSchema + '\'', (err, result) => {
    if (err) {
        console.log(err);
    } else {
        for (let i = 0; i < result.rows.length; i++) {
            const table_name = result.rows[i].table_name;

            pgClient.query('SELECT * FROM public."' + table_name + '"', (err, result) => {
                const limit = 100;
                let skip = 0;
                let count = 0;
                const total = result.rows.length;
                while (count < total) {
                    pgClient.query('SELECT * FROM public."' + table_name + '" LIMIT ' + limit + ' OFFSET ' + skip, (err, result) => {
                        if (err) {
                            console.log(err);
                        } else {
                            mongoClient.db(mongoDbName).collection(table_name + '').insertMany(result.rows, (err, result) => {
                                if (err) {
                                    console.log(err);
                                } else {

                                }
                            });
                        }
                    });
                    skip += limit;
                    count += limit;
                }
            });
        }
    }
});
```

Fig. 2 Code example for database migration from PostgreSQL to MongoDB

```
        const int32Max = 2147483647;
        const largestDocumentKeys = Object.keys(largestDocument[0]);
        const largestDocumentValues = largestDocumentKeys.map(key => largestDocument[0][key]);
        const largestDocumentTypes = largestDocumentValues.map(value => {
            if (isString(value)) { return 'TEXT'; }
            else if (isDate(value)) { return 'DATE'; }
            else if (isNumber(value) && Number(value) > int32Max) { return 'BIGINT'; }
            else if (isNumber(value) && Number(value) <= int32Max) { return 'INTEGER'; }
            else if (isArray(value) && !isArrayEmpty(value) && isArrayOfNumbers(value)) { return 'INTEGER[]'; }
            else if (isArray(value) && !isArrayEmpty(value) && isArrayOfStrings(value)) { return 'TEXT[]'; }
            else if (isArray(value) && isArrayEmpty(value)) { return 'TEXT[]'; }
            else if (isBoolean(value)) { return 'BOOLEAN'; }
            else return 'TEXT';
        });

        const largestDocumentTypesWithKeys = largestDocumentKeys.map((key, index) => `${key} ${largestDocumentTypes[index]}`);
        await pgClient.query(`CREATE TABLE ${table} (${largestDocumentTypesWithKeys.join(',')})`);
```

Fig. 3 Example code to determine the type of a column in a database table

In conclusion, it is important to test the data migration thoroughly to ensure that the data has been loaded correctly and that necessary constraints and indexes have been applied. This may involve performing data validation checks, comparing the data in the old and new databases, and testing the performance of the new database with the migrated data.

While loading data into a new database after deriving schemas can be a relatively simple process, it is still important to approach the migration with care and attention to detail to ensure a successful outcome.

For instance, when it comes to inserting data into MongoDB, it's important to ensure that you have the correct data types before sending it to the database. This is because MongoDB is a document-oriented database, which means that data is stored as documents with their unique structure.

To ensure that your data is properly inserted into MongoDB, you can think of the process as being similar to a table scraper. Just like how a scraper would collect data from a table, you'll need to collect the correct data types for each field in your document.

Once you have all of the correct data types, you can then send the data to the database. This is important because MongoDB is quite strict about data types, and if you send data with the wrong data type, it can cause issues in the future when you try to read the data.

By ensuring that you have the correct data types before inserting data into MongoDB, you can be sure that your data will be read correctly in the future, without any issues related to data types. This can help to prevent problems down the line and ensure that your data is stored and retrieved as expected. Below is the code of the method that allows you to overwrite data from a table of this database into a newly created table of another database.

```
const keys = Object.keys(row);
const values = keys.map(key => row[key]);

for (const key of keys) {
    if (columns.indexOf(key) < 0) {
        const result = await pgClient.query(`SELECT data_type FROM informa-
tion_schema.columns WHERE table_name = '${table}' AND column_name = '${key}'`);
         if (result.rows.length === 0) {
             const value = row[key];
             if (isString(value)) { await pgClient.query(`ALTER TABLE ${table} ADD
COLUMN ${key} TEXT`); }
             else if (isDate(value)) { await pgClient.query(`ALTER TABLE ${table} ADD
COLUMN ${key} DATE`); }
             else if (isNumber(value) && Number(value) > int32Max) { await pgCli-
ent.query(`ALTER TABLE ${table} ADD COLUMN ${key} BIGINT`); }
             else if (isNumber(value) && Number(value) <= int32Max) { await pgCli-
ent.query(`ALTER TABLE ${table} ADD COLUMN ${key} INTEGER`); }
             else if (isArray(value) && !isArrayEmpty(value) && isArrayOfNum-
bers(value)) { await pgClient.query(`ALTER TABLE ${table} ADD COLUMN ${key}
INTEGER[]`); }
             else if (isArray(value) && !isArrayEmpty(value) && isArrayOf-
Strings(value)) { await pgClient.query(`ALTER TABLE ${table} ADD COLUMN ${key}
TEXT[]`); }
             else if (isArray(value) && isArrayEmpty(value)) { await pgCli-
ent.query(`ALTER TABLE ${table} ADD COLUMN ${key} TEXT[]`); }
             else if (isBoolean(value)) { await pgClient.query(`ALTER TABLE ${table}
ADD COLUMN ${key} BOOLEAN`); }
      else { await pgClient.query(`ALTER TABLE ${table} ADD COLUMN ${key} TEXT`); }
      columns.push(key);
     }
  }
}
const query = `INSERT INTO ${table} (${keys.join(',')}) VALUES (${values.map((value,
index) => `$${index + 1}`).join(',')})`;
```

If you have followed the correct procedures when inserting data into MongoDB, you would end up with a table structure that is both consistent and accurate. This is because MongoDB is designed to store data flexibly and dynamically, which means that it can handle a wide variety of data types and structures.

By taking the time to define your data structure beforehand, you can ensure that your data is organized in a way that makes sense, and then it will be easier in the future to access and manipulate. This can help in developing the process smoother and in improving the performance and scalability of the application.

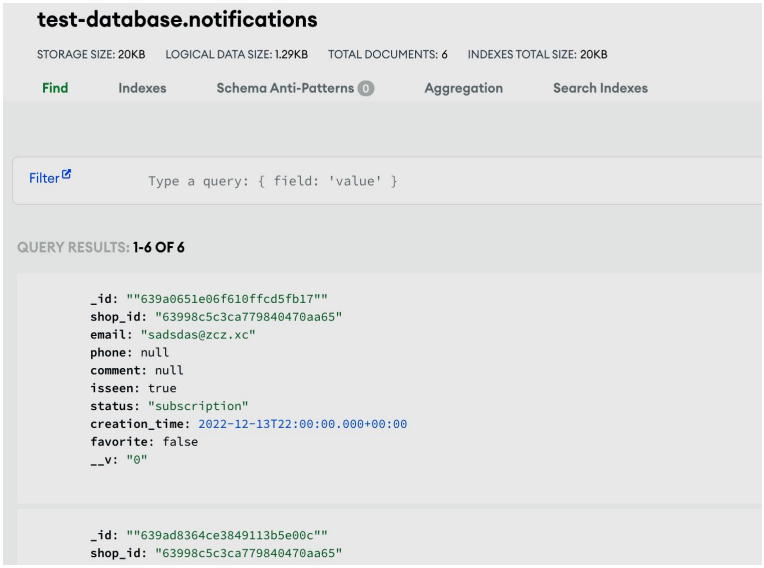

Fig. 4 Example of migrated PostgreSQL database to MongoDB

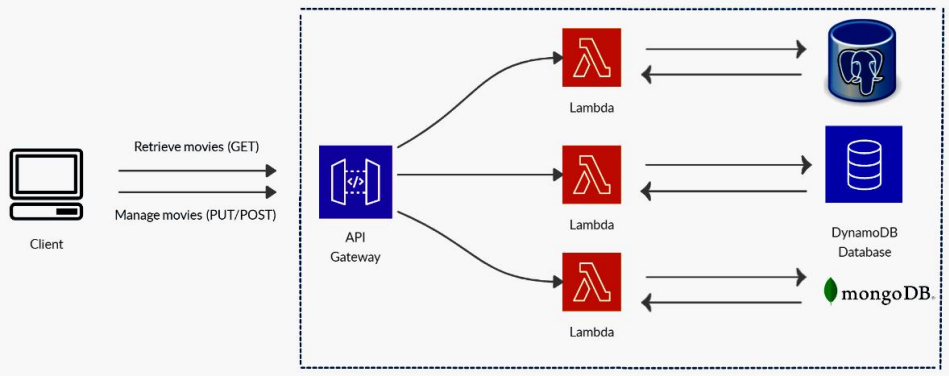

Fig. 5 A graphical representation of the architecture of a cloud solution on Amazon Web Services

5. Big Data Gap

There is an additional technical challenge associated with data migration, which is the sheer quantity of data that needs to be moved. For large databases, migrating the data can take hours or even days of continuous computing power. To address this challenge, it is often necessary to implement a cloud architecture that is optimized for data migration, with the necessary computational resources and infrastructure to ensure a smooth and efficient migration process.

The combination of AWS Gateway, AWS Lambda, and a relational database management system (RDBMS) provides a powerful and scalable architecture for building web applications and services. AWS Gateway acts as a front-end for the application, providing an interface for users to interact with the system. It allows users to send requests to the system via HTTP or other protocols, which are then forwarded to AWS Lambda for processing (fig. 4).

AWS Lambda is a serverless computing service provided by AWS that allows users to run code without having to manage servers. It can be used to implement business logic, data processing, and other functions required by the application. Lambda functions are triggered by events, such as requests from AWS Gateway, and can be written in a variety of programming languages, including Python, Node.js, and Java.

The flow of data in this architecture is as follows: when a user sends a request to AWS Gateway, the request is forwarded to AWS Lambda, which executes the appropriate function to process the request. The Lambda function can then read or write data to the RDBMS, depending on the requirements of the application. Once the data has been processed, the Lambda function returns a response to AWS Gateway, which in turn sends the response back to the user.

In summary, this architecture provides a scalable and flexible solution for building web applications and services, with the ability to handle large volumes of traffic and process data quickly and efficiently [8]. The cost of using AWS Lambda for database migration depends on various factors such as the size of the database, the amount of data being migrated, the duration of the migration process, and the number of Lambda function invocations required to complete the migration [2].

Assuming a 2GB database migration, the cost would depend on the time taken to complete the migration process. Lambda charges are based on the number of invocations,

duration of the function execution, and memory usage. As an example, let's assume that the migration process takes 2 hours to complete and requires a Lambda function with 512MB of memory.

The estimated cost of the Lambda function is determined by the following factors:

- execution Time: 2 hours or 7200 seconds;
- memory: 512 MB;
- number of Function invocations: 1.

The cost of running this lambda function is calculated as follows:

- compute time:

7200 seconds * 512 MB = 3,686,400 MB-seconds;

- compute cost:

$0.00001667 per GB-second * 3,686,400 MB-seconds / 1,000,000 = $0.0614;

- request cost:

$0.20 per 1 million requests * 1 = $0.0002.

Therefore, the estimated cost for migrating a 2GB database using AWS Lambda is accessed as $0.0616. Please note that this is just an estimation since the actual cost can vary depending on some factors.

6. Alternatives

Several alternatives for database migration can be used depending on needs:

1. Backup and Restore: You can create a backup of your database and restore it on a new server. This method allows you to preserve the structure of the database and data, but it can be time and resource-consuming.

2. Database Replication: This method allows you to create a copy of your database on a new server and synchronize data between them. This method can be faster and less resource-intensive, but it requires additional configuration and maintenance.

3. ETL Process: This method allows you to collect data from different sources, transform it, and load it into a new database. This method can be useful if you want to change the format or structure of data in your database.

4. Cloud-based Solutions: You can use cloud databases, such as Amazon RDS or Microsoft Azure SQL, to easily migrate your database to a new server. This method can be fast and convenient, but it can be more expensive than other methods.

5. Docker Containers: Docker containers allow you to package your database and all its dependencies into one container that can be easily moved to a new server. This method can be convenient if you want to ensure the standardization of your servers.

Usually, specialized tools such as SQL scripts or ETL (Extract, Transform, Load) tools are applied for database migration. These tools typically provide a deeper level of control and customization, which can be important when migrating complex databases [5].

However, a code-free process can be useful for migrating simple databases or automating routine tasks during database migration. For example, you can create a simple interface for managing database migration using a code-free process.

7. Conclusions

In conclusion, migrating databases from one system to another can be a challenging process, particularly when dealing with large amounts of data. However, by leveraging the power of cloud computing technologies, such as AWS Lambda, data scientists and developers can implement efficient and scalable architectures for data migration and processing.

AWS Lambda, in particular, offers a serverless computing model that eliminates the need to manage servers and infrastructure, allowing data scientists to focus on writing and executing code [9]. High scalability and parallel processing capabilities make it an ideal choice for processing large datasets, while its integration with other AWS services, such as AWS Gateway and RDBMS, provides a complete solution for building web applications and services.

Given the advancements in cloud computing technology, data migration, and processing have become increasingly efficient and accessible. Finally, data scientists and developers become capable of handling even the most complex datasets with greater ease and achieving more profound insights.

8. Gratitude

The authors thank the Team of the Department of Computerized automation systems for their support.

9. Mutual claims of authors

The authors have no claims against each other.

References

[1] Ain El Hayat S., Bahaj M. Modeling and transformation from temporal object relational database into MongoDB: Rules. Advances in Science, Technology and Engineering Systems. 2020. № 5 (4). C. 618–625. DOI: https://doi.org/10.25046/aj050473.

[2] Andreas Wittig, Michael Wittig, “Amazon Web Services in Action”, Second Edition, Manning Publications Co., 2018.

[3] CRUD REST API with Node.js, Express, and PostgreSQL, 2022. [Online]. Available: https://blog.logrocket.com/crud-rest-api-node-js-express-postgresql/

[4] Ji L. F., Azmi N. F. M. The development of a new data migration model for NOSQL databases with different schemas in the environment management system. Journal of Environmental Treatment Techniques, № 8 (2). p. 787 – 793, 2020. http://www.jett.dormaj.com/docs/Volume8/Issue%202/The%20Development%20of.pdf

[5] Michael J Hernandez, “Database Design for Mere Mortals: 25th Anniversary Edition”, Addison-Wesley Professional, 2020.

[6] PostgreSQL, 2023. [Online]. Available: https://www.postgresql.org/docs/

[7] Preston Z., Practical Guide to Large Database Migration. CRC Press, USA, 2021, https://www.routledge.com/ Practical-Guide-to-Large-Database-Migration/Zhang/p/book

[8] Real-Time AWS Cloud Migration Monitoring, 2023. [Online]. Available: https://www.striim.com/blog/real-time-aws-cloud-migration-monitoring/

[9] Steve M. Burnett, “Amazon Web Services For Beginners”, US, chapt.1, pp. 10-16, 2021. https://www.studypool.com/documents/23651179/aws-for-beginners