University of Waterloo

Faculty of Engineering  
Department of Electrical and Computer Engineering

Analysis of Solutions for Writing Large Data Set into the Database Atomically

Lifion by ADP

New York, New York, United States

Prepared by  
Zhidong Zhang  
20619543  
z498zhan@edu.uwaterloo.ca  
3A Computer Engineering  
  
8 January 2019

692 Butternut Ave

Waterloo, Ontario, Canada

N2V 2L1

January 8, 2019

Vincent Gaudet, Chair  
Electrical and Computer Engineering  
University of Waterloo  
Waterloo, Ontario  
N2L 3G1

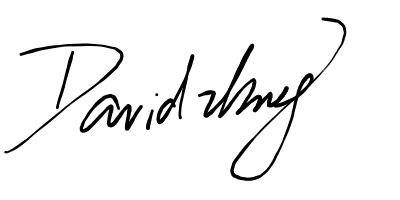
Dear Sir,

This report, entitled “Analysis of Solutions for Writing Large Data Set into the Database Atomically”, was prepared for my 3A Work Report. This report is in fulfillment of the course WKRPT 301. The purpose of this report is to evaluate different solutions for writing large data set into the database atomically and find out the best solution.

Lifion by ADP builds ADP’s next-generation human capital management system. This system is called the Lifion App Ecosystem which is a network of web applications designed to empower companies to meet their unique HCM needs.

The Lifion by ADP persistence group, in which I was employed, is managed by Adam Zeldis. The group is primarily involved with developing Lifion persistence services which allow users to create entity (tables), model (tables relationship) and shape (query to retrieve data). Therefore, we are responsible for processing the data, storing the data and retrieving the data.

I would like to thank Hozefa Moiyyadi for helping with understanding Lifion persistence services and the challenges we are facing in writing large data set. I hereby confirm that I have received no further help other than what is mentioned above in writing this report. I also confirm this report has not been previously submitted for academic credit at this or any other academic institution.

Sincerely,

Zhidong Zhang  
ID 20619543

# Contributions

During my co-op work placement, I was employed at Lifion by ADP in the position of a software developer. The persistence team I worked with was relatively small. It consisted of nine full-time employees and one intern. The persistence team is managed by Adam Zeldis, the engineering manager, and the persistence product is managed by Clifton Guzman, the product owner. I worked closely with everyone on the team. I was supervised by our two technical leads, Hozefa and Justin, and paired with a buddy, Mark. They helped me understanding the product, assigned me tickets and reviewed my code. The larger group this team belongs to, Lifion Development Platform group, was composed of hundreds of software engineers working on various projects. Throughout the work placement, I was working on both front end and back end of Lifion persistence services and modules.

This development team had many goals. Primarily, this team was responsible for developing persistence services, including front-end and back-end applications and several helper modules. Additionally, the group maintained the code base and its functionalities by bug fixes, continuous integration and deployment, and test development. This team also helped other teams who were using persistence services to understand better by answering their questions and solving their issues. Other goals included feature discussion, architecture design, production deployment, and server monitoring. Apart from that, the project manager was responsible for setting up meetings for sprint planning and communicating with other teams. When I joined the persistence team in September 2018, all persistence services were fully functional and deployed into production. From September to December, the persistence team focused on service enhancement, maintenance and new technology adoption. More specifically, batch read, batch write, client database sharding and GDPR implementation were the most important initiatives of the persistence team during the four months.

My tasks consisted of feature implementation, performance improvement, test development and bug fixing for the Lifion persistence service. I created API documentation for persistence services and assisted with secret adoption in the API server. Instead of using hardcoded database username and password, we started to use secrets to fetch the username and password. Additionally, I worked on feature development and bug fixes on our drag-and-drop UI designer. I also developed and optimized persistence services and many helper modules. Writing unit tests and integration tests are parts of my job as well. Since every intern is treated as a full-time employee in Lifion, I worked on lots of meaningful and challenging projects, participated in many high-level architecture design discussions, worked collaboratively with other teams and performed code views for other developers.

Most importantly, I was spending most of the time on designing and implementing the ability for persistence services to be able to write a large set of data into the database. After understanding the project scope and the requirements, I started researching and designing solutions for writing large data set. After several meetings and discussions with our technical leads and product owner, we had a final solution. Base on the solution, the product owner and I started planning the project. We estimated the complexity of the project and broke it down into small tasks and sub-tasks. After the planning is done, I followed the steps carefully and started to implement the service called persistence batch write service. During the implementation, I found challenges and issues I did not think about during the initial design. After investigations and researches, I successfully resolved the issue and finished the project on time. The result was supersizing. In the benchmarking testing, it proved that the persistence batch write service can successfully write one million records within sixty seconds, which far exceeded our expectations and needs.

The relationship between this report and my job is that the report is related to the solution that I worked on for writing a large amount of data into the database atomically. Since writing large data set is challenging and the solution varies a lot for different services and requirements, it requires a significant amount of research and analysis. I exercised my independent research ability while seeking a viable solution. I also improved my ability to design solutions for complex engineering problems. Besides, I developed my analytical skills from determining the requirements of the project and finding out the best solution that fits the needs. During the implementation, I practiced my programming skill, learned to write clean and understandable code and deepened my knowledge in technologies like JavaScript, Node.js, MySQL and DynamoDB. Moreover, I practiced my presentation skill when presenting my solutions and doing the product demo to the team. When I was preparing for the meetings, I learned to organize and visualize solutions in flow charts and state diagrams.

In the broader scheme of things, the persistence batch write service that I developed will continue to help persistence services to write large data set into the database atomically. The persistence batch write service architecture I used can be used for reference when writing large data set becomes an issue. This will allow my team or other teams to improve the batch write service architecture and its implementation. Additionally, the work I have done on the test script development will not only keep a good test coverage for the application but also reduce the number of bugs when adding new features.

# Summary

The main purpose of the report is to evaluate three different solutions and determine the best solution for writing large data set into the database atomically. The three solutions are file-based approach, batch write with staging query and batch write with staging table.

The scope of the report is for writing large data set into the database atomically in persistence services. But the solution overviewed in the report can also be used for writing large data set in other back-end applications. This report is intended for readers with basic knowledge of web applications and databases.

The major points covered in this report are as followed. The first section introduces the background information about web services and technologies, and briefly describes the project. Section 2 outlines the requirements and criteria of the solution for writing large data set into the database atomically. The third section describes the architecture of three possible solutions in detail. The fourth section analyzes the solutions based on the criteria, uses a decision matrix for quantitative analysis of the solutions, and determines the best solution based on the decision matrix.

The major conclusion in this report is that batch write with staging query is the best solution. Compared to the file-based approach, batch write with staging query performs significantly better on low latency, being able to fail fast and easy to implement. Compared to batch write with staging table, batch write with staging query have the advantage of being able to process batches in parallel, which means batch write with staging query has a lower latency. Even though all three solutions do well on primary requirements, batch write with staging query is the best solution for this project.

The major recommendation in this report is for Lifion by ADP to continuous using batch write with staging query to write large data set into the database atomically in persistence back-end services. Besides, adding retry strategies for operations like saving and executing queries is also recommended. Another recommendation is that using MySQL as the database. Although there are several SQL databases other than MySQL, MySQL is widely-used due to excellent scalability and more comprehensive transaction supports.

Table of Contents

[Contributions iii](#_Toc535185090)

[Summary vi](#_Toc535185091)

[List of Figures viii](#_Toc535185092)

[List of Tables ix](#_Toc535185093)

[1 Introduction 1](#_Toc535185094)

[1.1 Background 1](#_Toc535185095)

[1.2 Project Scope 3](#_Toc535185096)

[2 Requirements and Criteria 3](#_Toc535185097)

[3 Possible Solutions 5](#_Toc535185098)

[3.1 File-based Approach 5](#_Toc535185099)

[3.2 Batch Write with Staging Query 6](#_Toc535185100)

[3.3 Batch Write with Staging Table 8](#_Toc535185101)

[4 Engineering Analysis 9](#_Toc535185102)

[5 Conclusions 12](#_Toc535185103)

[6 Recommendations 13](#_Toc535185104)

[Glossary 14](#_Toc535185105)

[References 15](#_Toc535185106)

# List of Figures

[Figure 1. Example of an HTTP POST Request. 1](#_Toc489537873)

[Figure 2. Multi-instance Backend Service. 2](#_Toc489537873)

[Figure 3. File-based Approach Architecture. 5](#_Toc489537873)

[Figure 4. Batch Write API Endpoint Specification. 7](#_Toc489537873)

[Figure 5. Batch Write with Staging Query Architecture. 7](#_Toc489537873)

[Figure 6. Batch Write with Staging Table Architecture. 8](#_Toc489537873)

# List of Tables

[Table 1. Weighting Factor. 4](#_Toc489537874)

[Table 2. Criteria Weighting Scale. 4](#_Toc489537875)

[Table 3. Weighted Decision Matrix. 11](#_Toc489537875)

1. Introduction

This section contains the introduction of important concepts that will be mentioned in this report, including the background knowledge and the project scope.

Background

This section contains the background knowledge of HTTP POST request, database transaction, atomic operation and multi-instance back-end service that are closely related to the project.

HTTP POST Request

The HTTP POST request is used to send data to a receiving server to create or update a resource and designed to communicate between clients and servers. It usually sends a small amount of data which is called a request payload [1].

Figure 1 shows an example of how an HTTP POST request works. A client sends an HTTP POST request with a small payload which is a person data object. The server receives the request, processes the payload data and returns a success or failure response to the client.

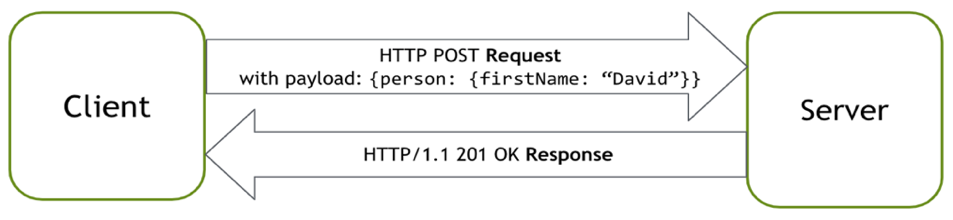


Figure 1: Example of an HTTP POST Request

There are two limitations on HTTP POST requests caused by the server implementation. One is the payload size limit. Since HTTP requests are not meant to send large files, the server usually sets the payload size less than 10 MB. If a user sends a payload greater than 10 MB, the server will close the connection immediately and throw a “Request Entity Too Large” error to the client. The other limitation is the request timeout limit. Since a server should not keep unused connections alive for too long, there is a request timeout limit which is usually set to 60s. To keep the HTTP connection alive between client and server, the client must periodically inform the server that it is still there by sending information to that server. If the server does not receive any information from the client through the same HTTP connection within a specified time, the server will drop the HTTP connection and stop wasting resources. For example, the server has a 60s request timeout limit. A client is trying to send a large payload using an HTTP POST request to the server. If the server does not finish processing the payload data within 60 seconds, the client will receive a “Request Timeout” error. Therefore, due to the server implementation and HTTP request characteristics, using an HTTP POST request to send large data or perform a long-running task is costly and error-prone.

Database Transaction and Atomic Operation

A database transaction is a series of read or write operations which are either all completed successfully or undone from the database. In relational databases, database transactions must maintain atomicity, consistency, isolation and durability [2]. More specifically, atomicity ensures that all operations within a transaction must succeed completely; otherwise, all changes that have been made must be undone completely. Consistency ensures that, when a transaction successfully commits, its data changes must persist correctly based on database constraints. Isolation provides that transactions are independent, which means transactions never share their data. Durability ensures that transaction data changes must persist in any system failure [3]. Therefore, using a database transaction can ensure data accuracy, completeness and integrity.

An atomic operation is one that either all happens, or nothing happens, which means it cannot be partial. It is obvious that database transaction is an atomic operation and is widely used when data accuracy is essential.

Multi-instance backend service

Multi-instance backend service consists of multiple backend server instances and a load balancer. More specifically, backend application is deployed multiple times concurrently, which creates multiple instances of the backend API server. A load balancer distributes the application traffic to each instance, and each instance can serve the incoming request, as shown in Figure 2.

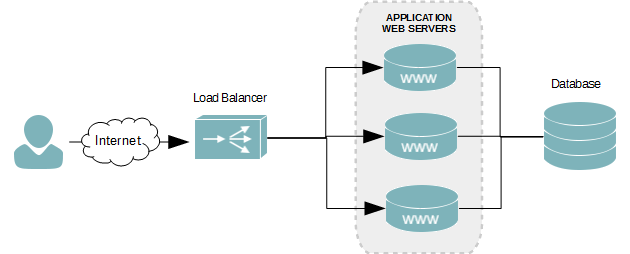


Figure 2. Multi-instance Backend Service [4]

Multi-instance backend service provides more scalability, availability and concurrency than single-instance backend service. If one instance goes down, the backend service will continue to serve the client. Since there are multiple instances, multiple requests from the client can be served concurrently.

Project Scope

Lifion by ADP builds HR applications and a metadata-driven development platform for building HR tools. Using Lifion development platform, developers can quickly and easily create, deploy and manage HR applications. In the platform, there are persistence services which handle all read and write operations to the database. There is another service called ETL which needs to use persistence services to write millions of records with accuracy. However, persistence services while performing writes with a significant amount of data was facing issues like payload too large, timeouts and socket hang up. Sending large payloads or files across the network to multiple servers is both costly and error-prone. This issue has been solved temporarily by increasing the payload limit. However, timeout error still occurred when payload size reached its limit. Therefore, to write a large amount of data into the database atomically in persistence services, a new solution is needed.

This report will analyze three possible solutions for writing large data set into the database atomically and use a decision matrix to determine the best solution.

1. Requirements and Criteria

There are three major requirements of the solution for writing a large amount of data into database atomically in persistence services:

1. The solutions must be able to write large data set without timing out or error out.
2. Atomicity must be guaranteed. To avoid user getting incomplete data, the entire write operation should act like a transaction which is all or none.
3. Write operations order must be maintained. For writing data into the database, there are three different operations: create, update and delete. Therefore, inside the large payload, insert, update and delete operations are specified in each data object as a property. It will result in a bad state if an update or delete for a record happens before the insert. For example, in the large payload, the first data object is a person record with an insert operation, and the second object is the same person record with a delete operation. After server finishes processing the payload, the server must first insert the person record to the database, and then delete the person record. The other way around would be deleting a non-existing person record, which results in an error. Since write operation types are defined in the original large payload by the client, the original large payload has the correct write operations order. Therefore, no matter how the large payload gets split into chunks, the order of the actual writes to the database needs to be the same as the order of the data and the operation types defined inside the original payload.

In addition to the three main requirements listed above, there are other less critical requirements should be outlined:

1. Latency should be minimal. Even though writing a large amount of data is a long-running job, the operation still needs to be relatively fast.
2. The system needs to be able to fail fast. The client needs to get notified by errors or exceptions as soon as possible to stop the current write operation or start retrying. By failing fast, it can not only save time and resources but also makes the system more robust.
3. Easy to implement. The solution should be straightforward and easy to implement for both client (ETL) and server (Persistence). In software design, simplicity is essential to make software easier to implement, maintain and enhance.

In total, there are six requirements for the project. These requirements can be summarized into six criteria: data size, atomicity, write operations order, low latency, fail fast and easy to implement. Based on the importance of each requirement and the weighting factor in Table 1, a weight is given to each criterion in Table 2.

Table 1: Weighting Factor

|  |  |
| --- | --- |
| Weight | Meaning |
| 5 | Very high importance |
| 4 | High importance |
| 3 | Medium importance |
| 2 | Low importance |
| 1 | Very low importance |
| 0 | Not important |

Table 2: Criteria Weighting Scale

|  |  |
| --- | --- |
| Criterion | weighting |
| Data Size | 5 |
| Atomicity | 5 |
| Write Operations Order | 5 |
| Low Latency | 3 |
| Fail Fast | 3 |
| Easy to Implement | 3 |

1. Possible Solutions

After introducing the project and analyzing the requirements, there are three viable solutions for writing large data set into the database atomically. The first solution is file-based approach, the second solution is batch write with staging query, and the third solution is batch write with staging table.

File-based Approach

The first solution is the file-based approach. Instead of using HTTP POST requests, the file-based approach uses file streaming to send large payloads to the persistence write service.

Figure 3 illustrates how the file-based approach works. First of all, the client stores the large payload into a file and store it in Amazon S3 which a simple storage service provided by Amazon. After that, the client sends the file information to persistence write service using AWS SQS which is a message queue. Upon receiving the file information message, persistence writes service loads and processes the payload file in S3 using a stream and write the payload data to the database in a single database transaction. Finally, persistence write service sends a success or error message to the client through SQS.

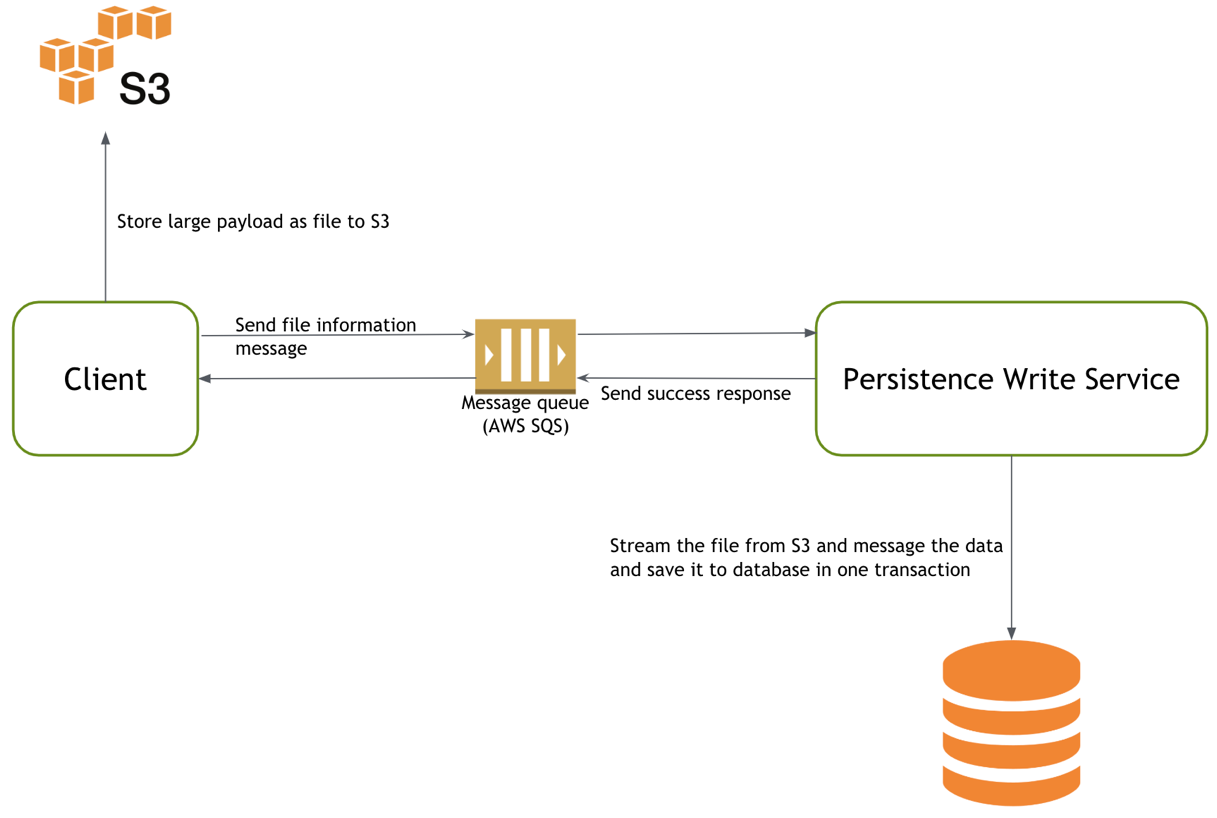


Figure 3: File-based Approach Architecture

However, there are several challenges in this file-based approach. Both client and persistence write service need to incorporate publisher and subscriber for communication, which adds complexity to the solution. Since the client and the server use a message queue to communicate, the client is disconnected throughout the writing process. Although it is not a time-bound approach, if there is an expectation to finish the job, getting noticed about it as soon as possible becomes a challenge.

To conclude, the first solution is the file-based approach. The solution consists of loading and processing the payload file into the persistence write service using a stream, writing the payload data to the database in a single database transaction and using a message queue to communicate between the client and the persistence write service.

Batch Write with Staging Query

The second possible solution is batch write with staging query. The large payload data is split into chunks, and all the data chunks are sent to the persistence batch write service as batches in parallel. For each batch, persistence batch write service processes the received data chunk, generates the SQL query that saves the data and captures the SQL query into the database without executing it. After all batches are processed successfully, persistence batch write service executes all the captured SQL queries in a single database transaction.

Figure 4 shows the RESTful API endpoint definition of the persistence batch write service, and Figure 5 describes how the batch write with staging query works. The entire batch write operation is considered a job and identified using a unique job id. There are three stages in a job: begin, batch and commit. In the beginning stage, the client splits the large payload into chunks less than 4 MB and marks each payload chunks with a batch index to keep track of the order of payload chunks inside the original large payload. Then, the client generates a unique job id and sends a begin signal using the begin API endpoint to the persistence batch write service. After receiving the begin signal, the persistence batch write service creates a staging query table for storing captured SQL queries. After that, it is the batching stage. The client sends those payload chunks with its job id and batch indexes using the batch API endpoint in parallel. Once receiving those payload chunks, persistence batch write service processes the data and generates the SQL query to store the data. Instead of executing the query, persistence batch write service saves the query as a string inside the staging query table along with the job id and its batch index. After all batch requests are processed successfully, it is the committing stage. The client sends the commit signal using the commit API endpoint with the same job id. To guarantee the atomicity and the correct write operations order, batch write service runs a stored procedure which retrieves all the queries belongs to the job from the staging query table and execute all the queries based on the batch indexes order in a single database transaction.

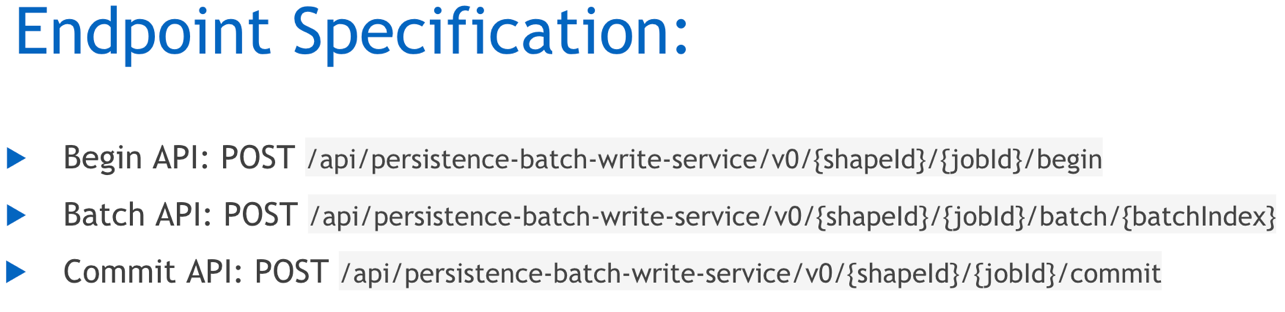
**

Figure 4: Batch Write API Endpoint Specification

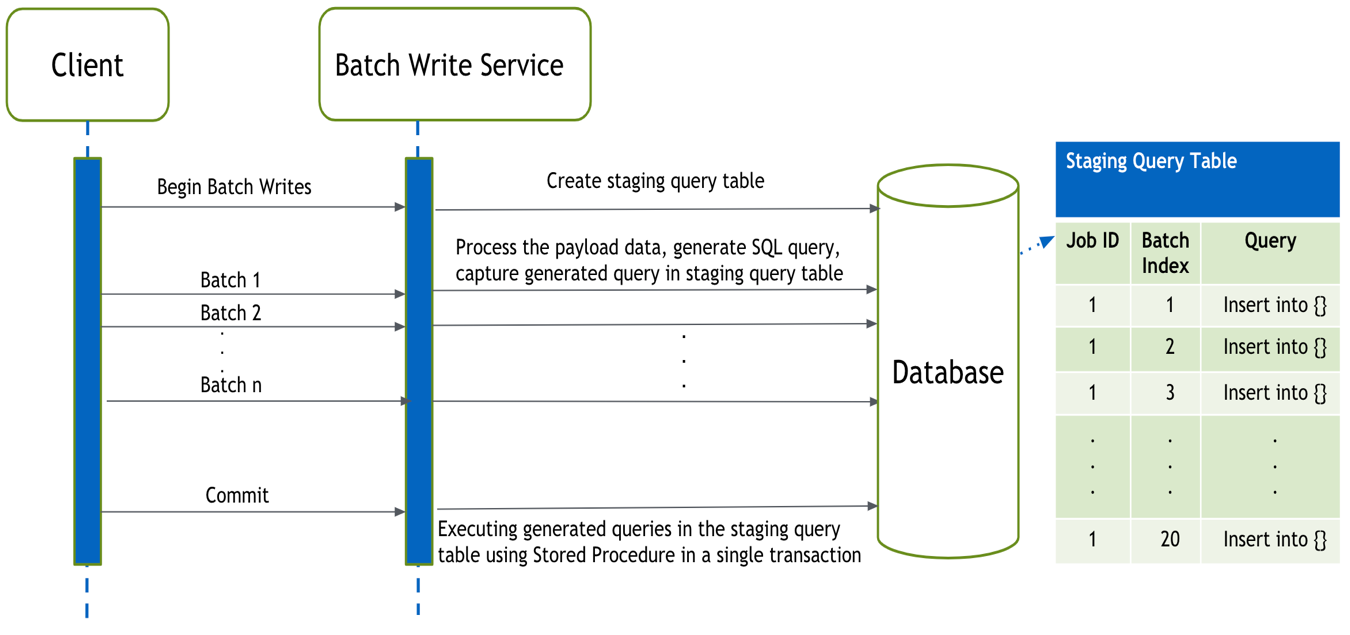


Figure 5: Batch Write with Staging Query Architecture

By marking each batch and each query with a batch index, it not only ensures the correct operations order but also achieves parallel batching requests. Therefore, this design takes advantage of the multiple instances and makes the batching process lightweight and fast. Moreover, the entire batch write operation is synchronized on the client side so that it can fail fast when a certain batch request has issues with the data or services.

To conclude, the second solution is batch write with staging query. The solution consists of splitting the large data set into small chunks, sending data chunks to persistence batch write service as batches in parallel, capturing the SQL query for each batch without executing the query, and executing all the captured queries based on the batch indexes order in one database transaction after the last batch arrives.

Batch Write with Staging Table

The third solution is batch write with staging table. The large payload is split into small chunks, and all the data chunks are sent to the persistence batch write service as batches sequentially. For each batch, persistence batch write service processes the received data chunk and stores the data into the staging table. After all batches are processed successfully, persistence batch write service moves data from the staging table to the actual table.

Figure 6 describes how the batch write with staging table works. Similar to the batch write with staging query, the entire batch write operation is considered as a job and identified using a unique job id. There are three stages in a job: begin, batch and commit. In the beginning stage, the client splits the large data set into chunks with size less than 4 MB. After that, the client sends a start request and gets back with a unique job id which is used throughout the entire batch write process. After that, it is the batching stage. The client sends those payload chunks using the batch endpoint to the batch write service. To maintain write operations order, the client needs to send those payload chunks in sequence. Upon receiving those payload chunks, the batch write service processes the data and saves the data into the staging tables which are created in the runtime when the actual tables are created. After all batch requests are processed successfully, it is the committing stage. The client sends the commit signal using the commit API endpoint with the same job id. Then, the batch write service uses DML statements to move data from staging tables to actual tables in a single database transaction.

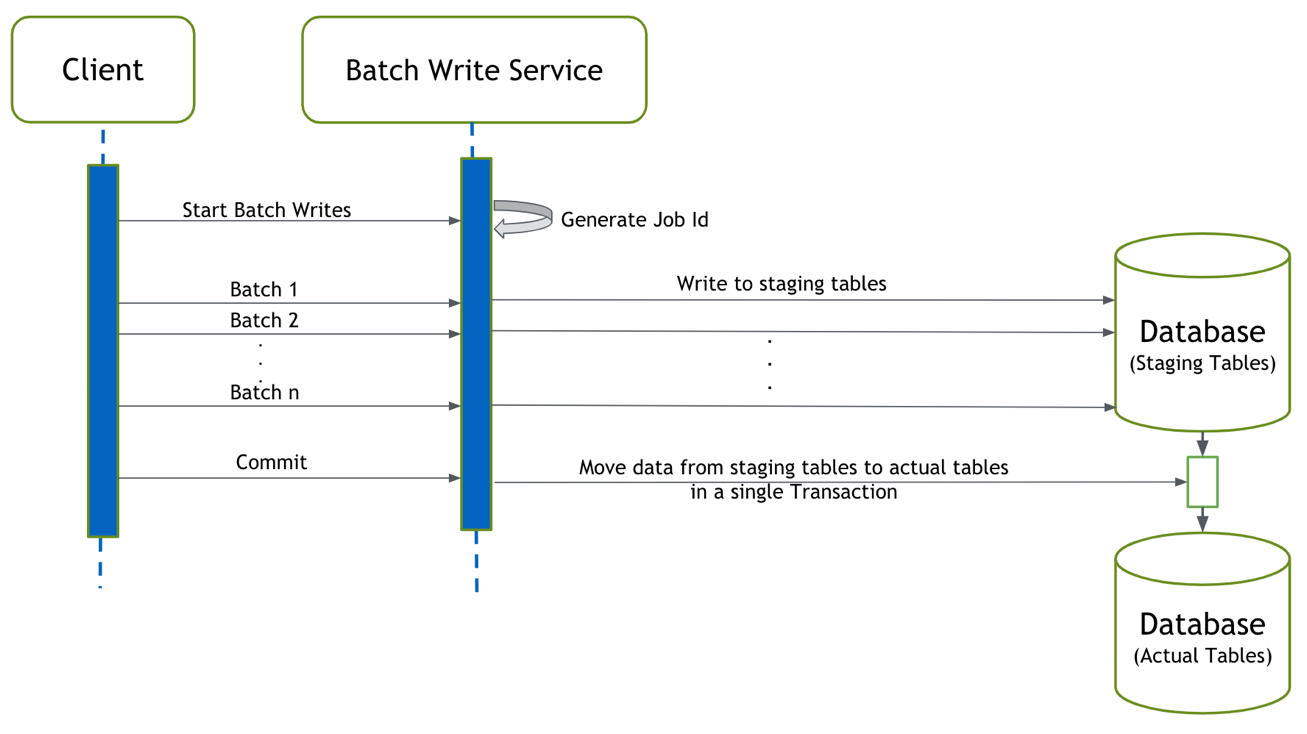


Figure 6: Batch Write with Staging Table Architecture

This solution is straightforward for inserts, but there are some challenges in updates and deletes since those operations types need to be stored to migrating the data from staging tables to actual tables. It is challenging to work on DML statements for migrating the data as well.

To conclude, the third solution is batch write with staging table. The solution consists of splitting the large data set into chunks, sending data chunks to persistence batch write service as batches sequentially, storing the data in the staging table for each batch and migrating all the data from staging tables to actual tables after the last batch arrives.

1. Engineering Analysis

Those three solutions will be analyzed based on the requirements and criteria listed in Section 2. During the analysis, each solution will be given a score from 1 to 10 according to how well it meets each criterion. A total score will be calculated using a decision metric. The solution that has the highest score will be the most feasible solution. From here, file-based approach will be called solution 1, batch write with staging query will be called solution 2 and batch write with staging table will be called solution 3.

**Criteria 1 – Data size:**

All three solutions support large data size. Solution 1 uses a message queue to communicate between the client and the server, and using file streaming to fetch the payload data, which resolves the timeout limit and payload limit issues. Therefore, solution 1 can write a large amount of data without timing out. In solution 2 and 3, by properly splitting and grouping the payload in chunks, throughput is no longer limited by payload size and timeout limit. Therefore, all three solutions are given a score of 10 on data size.

**Criteria 2 - Atomicity:**

All three solutions use database transaction to ensure the atomicity of the operation. Therefore, all three solutions are given a score of 10 on atomicity.

**Criteria 3 – Write Operations Order:**

In solution 1, the large payload file is streamed and process sequentially, which means the write operations order is preserved. In solution 2, marking the payload chunks using batch indexes and executing it based on the batch indexes order also preserve the write operations order. In solution 3, batch write requests are executed sequentially to maintain the payload chunks’ order and data correctness in the staging tables. All three solutions do well on keeping the write operations order. Therefore, all three solutions are given a score of 10.

**Criteria 4 – Low Latency:**

In solution 1, persistence write service needs to stream the file and processes it sequentially, which means it can only operate on one instance of the service for each job. Similarly, in solution 3, batch requests need to be sent and process sequentially to maintain the write operations order. Whereas, in solution 2, batch requests can be sent and processed in parallel and take advantage of multiple instances to achieve concurrency, which means its latency is lower than the other two solutions. Therefore, the score of solution 2 is 10, and the scores of solutions 1 and solution 3 are both 5.

**Criteria 5 – Fail Fast:**

In solution 1, by using a message queue to communicate between the client and the service, the client is disconnected throughout the writing process, which means if there is an exception to finish the job, it is hard to notify the client immediately. However, in solution 2 and 3, by using HTTP requests and keeping a live connection between the client and the server, if a certain batch request has issues with data or services, the client gets an error response immediately and fails fast. Therefore, regarding being able to fail fast, solution 1 is given a score of 6, and solution 2 and 3 are given a score of 10.

**Criteria 6 – Easy to** **Implement:**

In solution 1, incorporating publisher and subscriber for communication on both client and service side adds complexity. Therefore, solution 1 is given a score of 5. Solution 2 and solution 3 are both API (HTTP requests) approaches, which makes it easier to be adopted by the client. On the service side, solution 2 is easier to implement than solution 3, since there are challenges on handling update and delete operations when using the staging table. Therefore, solution 2 is given a score of 10, and solution 3 is given a score of 7.

Table 3 on the next page shows how much each solution scores in different criteria and calculates the overall score of each solution.

Table 3. Weighted Decision Matrix

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | Solutions | | | | | |
|  |  | Solution 1 | | Solution 2 | | Solution 3 | |
| Criteria | Weighting | Score | Weighted | Score | Weighted | Score | Weighted |
| Data Size | 5 | 10 | 50 | 10 | 50 | 10 | 50 |
| Atomicity | 5 | 10 | 50 | 10 | 50 | 10 | 50 |
| Write operations order | 5 | 10 | 50 | 10 | 50 | 10 | 50 |
| Low Latency | 3 | 5 | 15 | 10 | 30 | 5 | 15 |
| Fail Fast | 3 | 6 | 18 | 10 | 30 | 10 | 30 |
| Easy to Implement | 3 | 5 | 15 | 10 | 30 | 7 | 21 |
| Total | 24 | 46 | 198 | 60 | 240 | 52 | 216 |
| Final Score |  |  | 8.25 |  | 10 |  | 9 |

As shown in Table 3, solution 1 has a total score of 8.25, solution 2 has a total score of 10 and solution 3 has a total score of 9. It is obvious that solution 2 has the highest score; therefore, solution 2 which is batch write with staging query performs the best for writing large data set to the database atomically in persistence backend services.

1. Conclusions

Three solutions file-based approach, batch write with staging query and batch write with staging table are introduced and analyzed for writing a large amount of data into the database atomically in persistence services.

Regarding data size, atomicity and write operations order, all three solutions are comparable. All of them show similar abilities to write large data set, ensure the atomicity of the write operation and guarantee the correct write operations order.

In terms of be able to fail fast and easy to implement, batch write with staging query and batch write with staging table does significantly better than the file-based approach. Using HTTP requests instead of a message queue, batch write with staging query and batch write with staging table have not only the simpler design but also the ability to notify the client in time.

Concerning low latency, batch write with staging query performs the best among those three solutions. To ensure the correct write operations order, file-based approach and batch write with staging table has to process the large payload sequentially, whereas, batch write with staging query uses batch indexes to keep track of the write operations order and make parallel batch processing possible. Therefore, by saving batch processing time, batch write with staging query has the lowest latency.

According to the decision matrix in Table 3, batching write with staging query performs perfectly on all six criteria and has the highest overall score. Therefore, batch write with staging query is the best solution for writing large data set into the database atomically.

1. Recommendations

Based on the analysis and conclusions in this report, it is recommended that batch write with staging query should be implemented for writing large data set into the database atomically in persistence backend services.

It is also recommended to add retry strategies for operations like storing SQL queries and executing SQL queries to improve the stability of the application.

For the scalability and functionality of the batch write service, the recommended database is MySQL. It is easy to set up master-slave replication in MySQL to deliver scalability [5]. Besides, MySQL Thread Pool provides a scalable thread-handling model to improve the scalability even further [6]. Regarding features, MySQL supports lots of SQL functionalities, including stored procedures and transactions. The stored procedure makes database-intensive operations, like executing all the batch queries, more efficient. Transactions make a sequence of database operations atomic and improve data accuracy. Therefore, MySQL is a recommended database for this project.

# Glossary

**Amazon S3**: It is a short form of Amazon Simple Storage Service, which is an object storage service that offers industry-leading scalability, data availability, security, and performance.

**Amazon SQS**: It is a short form of Amazon Simple Queue Service, which is a fully managed distributed message queuing service.

**API**: An application programming interface

**Back End:** The server-side of a web application

**DML**: A data manipulation language which deals with data manipulation, and includes most common SQL statements such as SELECT, INSERT, UPDATE, DELETE etc. It is used to store, modify, retrieve, delete and update data in the database.

**Front End:** The client-side of a web application

**HTTP:** Hypertext Transfer Protocol

**Message Queue**: A form of asynchronous service-to-service communication used in serverless and microservices architectures.

**RESTful API**: An application programming interface that uses HTTP requests to GET, PUT, POST and DELETE data.

**Stored Procedures**: A set of [Structured Query Language (SQL)](https://searchsqlserver.techtarget.com/definition/SQL) statements with an assigned name, which is stored in a [relational database management system](https://searchdatamanagement.techtarget.com/definition/RDBMS-relational-database-management-system) as a group so that it can be reused and shared by multiple programs.

**SQL**: It stands for Structured Query Language and is used to communicate with a database.

# References

|  |  |
| --- | --- |
| [1] | D. Stenberg, Everything curl, Stockholm: GitBook, 2015. |
| [2] | "Transaction," [Online]. Available: https://www.techopedia.com/definition/16455/transaction. [Accessed 8 January 2019]. |
| [3] | "SQL Transactions," tutorialspoint, [Online]. Available: https://www.tutorialspoint.com/sql/sql-transactions.htm. [Accessed 8 January 2019]. |
| [4] | S. Rainville, "Load Balancing Web Servers with Windows Server 2012 R2," [Online]. Available: http://www.serverlab.ca/tutorials/windows/web-servers/load-balancing-web-servers-with-windows-server-2012-r2/. [Accessed 13 September 2017]. |
| [5] | "Using Replication for Scale-Out," Oracle, [Online]. Available: https://dev.mysql.com/doc/refman/5.7/en/replication-solutions-scaleout.html. [Accessed 8 January 2019]. |
| [6] | "MySQL Enterprise Scalability," Oracle, [Online]. Available: https://www.mysql.com/products/enterprise/scalability.html. [Accessed 8 January 2019]. |