ISSN: 2755-0214

Journal of Economics & Management Research

Review Article

Open Access

Real-Time Data Replication for Postgres using WAL Event Capture and Its usage in E-Commerce Inventory Management

Gautham Ram Rajendiran

USA

ABSTRACT
Real-time data replication is a core need for distributed systems, considering that data-centric applications are developing rapidly and that high availability, scalability, and consistency need to be ensured. This paper proposes a robust, scalable architecture based on PostgreSQL's WAL mechanism to capture and replicate changes across different regions. The architecture uses AWS Lambda for event-driven processing and Kinesis Firehose to streamline the delivery of data to multiple targets, including Redshift and Snowflake. We delve into the detailed technical aspects and considerations needed to build such a system, challenges faced, and benefits from this approach when it comes to real-time data replication.

***Corresponding author**
Gautham Ram Rajendiran, USA.

Received: May 02, 2022; **Accepted:** May 09, 2022; **Published:** May 16, 2022

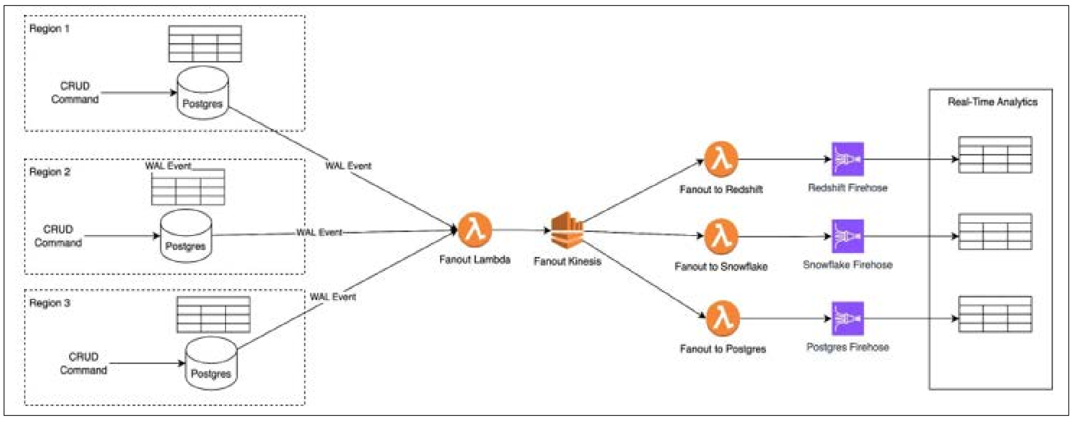
Keywords: Postgres, Write-Ahead Log, WAL Streaming, Data Replication, AWS Lambda, Kinesis, Redshift, Snowflake, Multi-Region, Cloud Computing

Introduction
Real-time data replication has become crucial for organizations in today's data-driven world to keep data synchronized across different regions and different varieties of platforms [1]. Many enterprises have difficulties in ensuring consistency, low latency, and availability of data across their operational environments. PostgreSQL Write-Ahead Log (WAL) provides a mechanism for capturing changes at the level of a transaction for enabling this with low impact on modifications without intrusive queries on the database itself [2,3].

The architecture presented utilizes AWS services, such as Lambda and Kinesis Firehose for processing and delivering these WAL events to multiple destinations, which is perfect for handling complex data replication scenarios [4,5]. By implementing this architecture, organizations will realize real-time replication and analytics, thus allowing a unified view of data across geographies.

Problem Scope
Most traditional data replication techniques are suffering from high latency, restricted scalability, and operational overheads. They usually require complex configurations for handling data collisions and schema changes in multi-environments. This proposed architecture leverages WAL events and AWS-managed services to help remove these limitations by providing an event-driven and serverless approach for data replication.

Implementation
Presented below is a high-level architecture diagram of how the system is implemented. It uses a plugin that can be installed in postgres in order to listen to WAL events, and converts these events into a JSON [3]. The Fanout Lambda periodically checks the WAL event log for new events using the LSN, and if new events are present, it processes these events and sends it out to the Fanout Kinesis endpoint to be processed by downstream consumers [6].



WAL To JSON Plugin

The first step in the architecture is capturing the Postgres data changes using a WAL to JSON plugin [7]. This plugin listens to the Postgres WAL and serializes each event into a JSON format. The JSON payload includes key metadata such as:

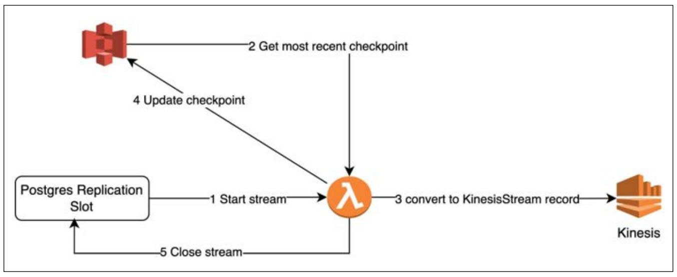
1. **Table Name:** The name of the table that has been modified.
2. **Schema Information:** Details about the schema to which the table belongs.
3. **CRUD Operation:** Specifies whether the operation is an Insert, Update, or Delete.
4. **Old vs. New Values:** For update operations, the plugin captures both the old and new values of the changed rows.

A sample Change Data Capture event

```
{
        "kind": "insert",
        "schema": "public",
        "table": "table1_with_pk",
        "columnnames": ["a", "b", "c"],
        "columntypes": ["integer", "character varying(30)",
"timestamp without time zone"],
        "columnvalues": [1, "Backup and Restore", "2018-03-27
11:58:28.988414"]
}
```

The JSON payload is enriched with additional metadata, such as the timestamp of the event and the region where the change occurred. This metadata is crucial for downstream processing and helps maintain traceability of data changes.

Postgres Stream to Fanout Lambda


2 Get most recent checkpoint
4 Update checkpoint
Postgres Replication Slot
1 Start stream
3 convert to KinesisStream record
5 Close stream
Kinesis

Step 1: Read Replication Slot Stream

This example uses the PostgreSQL JDBC driver to create a logical replication slot stream. A replication slot keeps track of WAL (Write-Ahead Log) position for the streaming client [8]. It is a necessity to prevent data loss when the database gets altered.

Create Replication Slot

The create Replication Slot Stream method initiates a logical replication stream using the given slot options - include-timestamp, and status interval configuration. This slot is used to stream WAL events continuously to the Lambda function.

PG Replication Stream

This object initiates the logical replication stream from the given replication slot [9]. It continuously reads the WAL events and passes them on to the next step for processing.

```
private PGReplicationStream createReplicationSlotStream(Con
nection databaseConnection,
                                ReplicationSlotDataSource
replicationSlotDataSource)
      throws SQLException {
    final PGConnection pgConnection = databaseConnection.
unwrap(PGConnection.class);
    return pgConnection.getReplicationAPI()
        .replicationStream()
        .logical()
                .withSlotName(replicationSlotDataSource.
getReplicationSlot())
        .withSlotOption("include-timestamp", true)
        .withSlotOption("include-types", false)
        .withStatusInterval(2, TimeUnit.SECONDS)
        .start();
  }
```

Step 2: Get Checkpoint and Process Checkpoint Records

After the stream is established, the system will read the WAL events and process these in batches.

Checkpoint Handling

The function handle Stream Batch captures a batch of records from the replication stream and finds out what is the last processed Log Sequence Number. To prevent data loss from occurring due to failures, it will keep track of LSNs that have been processed.

Record Extraction

The program loops through the batch of PG Stream Data, accumulating all the records to prepare them for processing. It keeps track of the last non-empty LSN for accurately identifying the most recent changes.

Logging and Writing

Each record is logged while the record Writer writes accumulated records to the intermediate store, or sends them directly to a downstream service such as AWS Lambda or Kinesis.

```
private void handleStreamBatch(List<PGStreamData> dataList) {
    if (dataList.isEmpty()) {
      return;
    }
   LogSequenceNumber lastLSN = dataList.get(dataList.size()
- 1).getSequenceNumber();
    LogSequenceNumber lastNonEmptyLSN = null;
    for (int i = dataList.size() - 1; i >= 0; i--) {
      PGStreamData streamData = dataList.get(i);
      if (!streamData.getRecordList().isEmpty()) {
      lastNonEmptyLSN = streamData.getSequenceNumber();
        break;
      }
    }
   log.info("get last LSN {}, last non-empty LSN {}", lastLSN,
lastNonEmptyLSN);
    List<Record> recordList = new ArrayList<>();
    for (PGStreamData data : dataList) {
      recordList.addAll(data.getRecordList());
    }
    if (!recordList.isEmpty()) {
      log.info("get record size {}", recordList.size());
      recordWriter.write(recordList);
    }
    this.onSuccess.accept(lastLSN, lastNonEmptyLSN);
  }
```

Step 3: Send Processed Records to Kinesis

Once the records are processed, they need to be forwarded to a Kinesis stream for further handling. The Lambda function that receives the records transforms the data into a format suitable for Kinesis and pushes it downstream.

Transformation to Kinesis Record
The Lambda function typically converts each record into a JSON or binary format before sending it to Kinesis. Additional metadata may be added during this transformation, such as the source region, event timestamp, and other relevant attributes.

```
  @Retryable(maxAttempts = 3, delay = 1000, backoffCoefficient = 1.5,
      recoverableThrowables = {AmazonKinesisFirehoseException.class})
    private FirehoseRedshiftDefinition getFirehoseRedshiftDefinition(String firehoseName) {
        FirehoseRedshiftDefinition definition = firehoseCache.get(firehoseName, FIREHOSE_CACHE_TIME);
        if (definition != null) {
            return definition;
        }
        DescribeDeliveryStreamRequest request = new DescribeDeliveryStreamRequest();
        request.setDeliveryStreamName(firehoseName);
        DescribeDeliveryStreamResult result = firehoseClient.describeDeliveryStream(request);
        if (result == null) {
            return null;
        }
        definition = FirehoseRedshiftDefinition.from(result.getDeliveryStreamDescription());
      log.info("get firehose data {} for {}", definition, firehoseName);
        firehoseCache.put(firehoseName, definition);
        return definition;
    }

    @Override
    @WithMetrics
    public void write(@Count(name = "BatchWriteToFirehose.InputSize", path = "size()") final List<Record> theRecords) {

        if (theRecords == null || theRecords.isEmpty()) {
            log.info("empty record, skip");
            return;
        }
        final List<Record> records = ImmutableList.copyOf(theRecords);
      final Map<String, List<Record>> tableNameByRecordsMap =
            records.stream().collect(Collectors.groupingBy(Record::getTableName));
        final Map<String, List<Record>> supportedTableNameByRecordsMap =
            tableNameByRecordsMap.entrySet().stream()
                .filter(entry -> isTableSupported(entry.getKey()))
                .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));
        supportedTableNameByRecordsMap.forEach(this::putRecordsByTable);
    }
```

Step 4: Update most Recently Processed LSN to S3

After successfully sending the records to Kinesis, it's crucial to update the checkpoint information. This ensures that in the event of a failure or restart, the system knows from which LSN to resume processing.

Checkpoint Update
The LSN of the last successfully processed batch is stored in an S3 bucket. This checkpoint serves as a marker for where the replication stream should resume in case of any interruption.

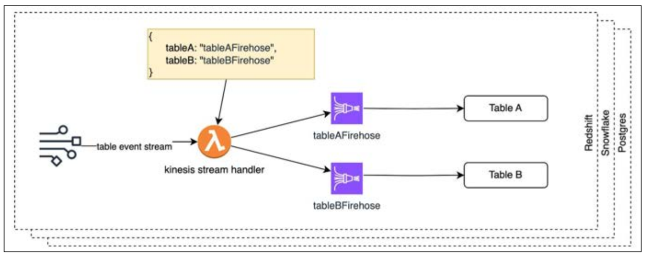
Step 5: Gracefully Handle Close

To avoid data loss or inconsistencies, the replication stream must be closed gracefully. The Lambda function should ensure that all pending records are flushed, and the checkpoint is updated before the stream is closed.

```
    @Override
    @Retryable(maxAttempts = DEFAULT_RETRY_ATTEMPTS,
        delay = DEFAULT_RETRY_DELAY_INTERNAL_IN_MILLIS,
        backoffCoefficient = DEFAULT_RETRY_BACKOFF_COEFFICIENT,
        recoverableThrowables = {RetryableException.class})
    public void close() {
        this.streamStarted = false;
        try {
            log.info("Closing replication slot stream");
            if (this.stream != null) {
                this.stream.close();
            }
            log.info("Closing db connection");
            if (this.dbConnection != null) {
                this.dbConnection.close();
            }
        } catch (SQLException e) {
            log.warn("Failed to close RecordReader", e);
        throw new RetryableException("Failed to close RecordReader", e);
        }
    }
```

Resilience and Idempotency
1. **Error Handling and Retries:** Provide comprehensive error handling that captures failed records for retries. To that effect, the system is robust by using retries from AWS Lambda and implementing retry logic within the handle Batch function.
2. **Event Ordering:** The replication slots in Postgres are designed to maintain a strict ordering of events. This architecture leverages that feature to process all the changes in the order they happened, so that across regions/target destinations, consistency is maintained.
3. **Idempotency:** This Lambda function should be idempotent upon processing every batch of WAL events. Each record should contain an identifier based on LSN and timestamp, which means it cannot be duplicated in the downstream systems.

Fanout Lambda to Kinesis
The diagram demonstrates how the Fanout Lambda function processes events and sends them to different Kinesis Firehose delivery streams, ultimately delivering data to tables in Redshift or Postgres [10].



Event Source to Lambda: The architecture is initiated with an event stream, or in other words, input that Lambda reads from, comprising database change events. Each event is enriched with metadata and labeled in relation to the target tables.

Table Mapping
The following mapping of tableA and tableB corresponds to the Firehose delivery streams, namely tableAFirehose and tableBFirehose. This needs to be configured prior to enabling of the stream so that correct routing can be ensured.

Firehose Delivery
The output of the Firehose streams is configured to deliver the data into Redshift, Postgres or any of the streams supported by Firehose [5]. Each table at the destination will pre-exist, and ingestion should have been already pre-configured via Firehose.

Event Routing
The routing logic in the Lambda function inspects the metadata of each event to determine which Firehose stream is the target. For example, if the source table of the event is tableA, then it routes the event to tableAFirehose.

Further transformations can occur, like event payload conversion to CSV or JSON before sending to Firehose.

Requirements Pre-Configuration
The tables to be replicated need to be created at the target database systems. Table A and Table B therefore need to be present in the destination before turning a stream.

Firehose Mappings
All tables should have a corresponding mapping created for the Firehose delivery stream to avoid misrouting events.
Technical Considerations

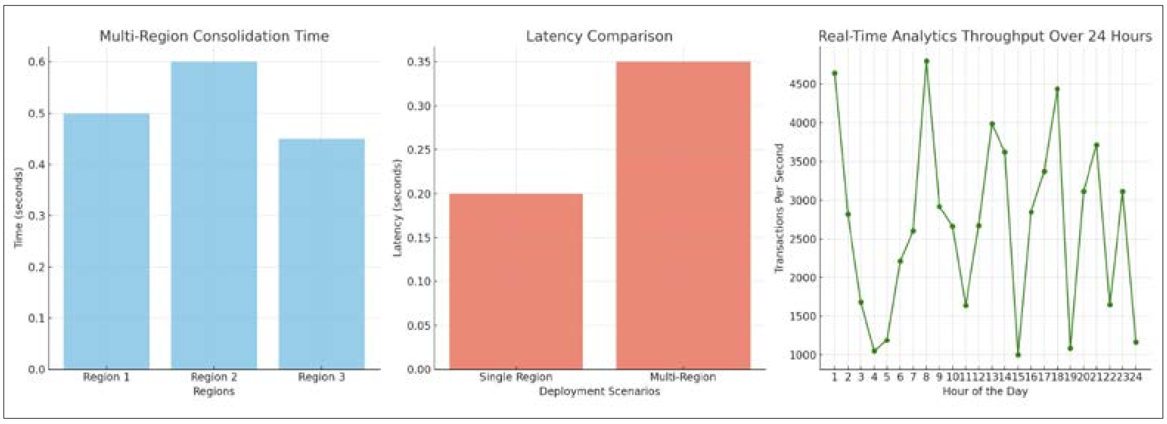
Schema Evolution
Implement a schema registry and automated update mechanism for Firehose in case there are frequent table schema changes [11].

Idempotency and Retry Logic
The events could be processed multiple times, so at both places-Lambda and Firehose, idempotency keys or deduplication logic need to be configured to avoid duplicate records [12].

Result
The proposed real-time data replication architecture demonstrates significant improvements in multi-region data consolidation, real-time analytics performance, and latency reduction.

E-Commerce Inventory Management
This is the real-time data replication architecture from PostgreSQL, which leverages WAL events for better inventory management for eCommerce platforms. It enables the architecture to stream every change coming in the database in real time into various downstream systems and keep the inventory data across multi-regions consistent and updated. This is particularly critical in eCommerce, where the availability of accurate inventory data directly impacts order fulfillment and customer satisfaction. It allows for seamless synchronization of the stock level between primary and regional warehouses to make sure that the global view is always correct. For every new order placed or item returned, the changes are captured as WAL events and immediately propagated to downstream databases such as Redshift and Snowflake. This is how the platform can do real-time inventory analytics and predict stock shortages before they even happen. By spreading these events over the regions, the system can scale with low latency for updates of its inventory, hence an efficient and global scalable solution for operations in eCommerce. This won't just reduce overselling or stockouts but also supports advanced features: dynamic reordering and auto-restocking based on real-time demand patterns.

Performance Metrics



Multi-Region Consolidation
The chart entitled Multi-Region Consolidation Time represents the time consumed while consolidating data from various regions. Each region represents a time in seconds, and test results are consistent across regions. Region 1 and Region 3 were able to perform slightly faster at 0.5 and 0.45 seconds, respectively, while Region 2 had a slightly higher consolidation time of 0.6 seconds. This would therefore imply that architecture has consistent performance within regions for low latency and timely availability of data to downstream systems.

Real-Time Analytics of Big Data
The graph on the right, Real-Time Analytics Throughput Over 24 Hours, shows the throughput performance of the system operating at high transactions per second: The system processes between 1,000 and 4,500 transactions every second during this 24-hour period. Peaks in throughput, such as for hours 2, 7, 14, and 18, indicate that sudden spikes in the data ingestion are handled with great efficiency by the system, thus being fit for scenarios needing real-time analytics and monitoring of big sets of data.

Very Low Latency
For instance, the Latency Comparison chart shows the difference between a single-region and multi-region deployment scenario. The average latency for a single region is about 0.2 sec, which goes up to 0.35 seconds when using multiple regions. The low latency in multi-region deployment speaks to efficiency in the architecture to propagate changes across multiple regions without introducing significant delays.

These results confirm that the real-time data replication system has met its objectives: it provides multi-region data consolidation, supports real-time analytics of large-scale data, and keeps latency very low-even in complex multi-region deployments. Next steps could be to further reduce latency by optimizing the processing times of Lambda functions and also increasing throughput by utilizing more parallelism in delivery streams in Kinesis Firehose.

Conclusion
In this paper, we proposed a highly available and scalable architecture for real-time replication using PostgreSQL Write-Ahead Log events and AWS services. The architecture will make use of logical replication slots in PostgreSQL, seamless integration with AWS Lambda, and Kinesis Firehose, which in turn will enable real-time data streaming across multiple regions into target data stores like Redshift and Postgres. It solves many of the cardinal challenges related to traditional data replication approaches characterized by high latency, lack of flexibility, and operational complexity. The study was a result of implementing this architecture in a high traffic e-commerce system and was used for real-time analysis of inventory data. Key results shown in this work bring to light the architecture's ability to perform multi-region data consolidation with consistent performance, allowing for high throughput as part of real-time analytics over large datasets. Furthermore, the system maintains very low latency and is thus appropriate for mission-critical applications that demand timely and reliable data replication.

References

1. Hamdi S, Ben Salem M, Bouazizi E, Bouaziz R (2013) Management of the real-time derived data in a Distributed Real-Time DBMS using the semi-total replication data. ACS International Conference on Computer Systems and Applications (AICCSA) Ifrane Morocco 2013: 1-4
2. (2024) PostgreSQL: The World's Most Advanced Open Source Relational Database. Available: https://www.postgresql.org/.
3. (2024) Write-Ahead Logging (WAL) Introduction. Available: https://www.postgresql.org/docs/current/wal-intro.html.
4. (2024) AWS Lambda. Available: https://aws.amazon.com/lambda/.
5. (2024) Amazon Kinesis Data Firehose. Available: https://docs.aws.amazon.com/firehose/latest/dev/what-is-this-service.html.
6. (2024) PostgreSQL Datatype PG LSN. Available: https://www.postgresql.org/docs/current/datatype-pg-lsn.html.
7. (2024) WAL2JSON. Available: https://github.com/eulerto/wal2json.
8. (2024) PostgreSQL JDBC Driver. Available: https://jdbc.postgresql.org/.
9. (2024) Logical Replication in PostgreSQL Available: https://www.postgresql.org/docs/current/logical-replication.html.
10. (2024) Amazon Redshift. Available: https://aws.amazon.com/redshift/.
11. (2024) Confluent Schema Registry. Available: https://docs.confluent.io/platform/current/schema-registry/index.html.
12. Gilbert J and Price E (2021) Software Architecture Patterns for Serverless Systems: Architecting for Innovation with Events, Autonomous Services, and Micro Frontends. Packt Publishing Ltd.