Architectural challenges for building a low latency, scalable multi-tenant data warehouse

Mataprasad Agrawal
Solutions Architect,
Services CTO
Our analytics services team at Persistent implemented a multi-tenant cloud analytics add-on product for one of our customers. The initial approach was to extend an existing ETL implementation that moved operational data from a single-tenant OLTP database schema to a data warehouse (DW), making it amenable to multidimensional analysis. This Whitepaper talks about (i) the challenges stemming from this approach, in terms of both flexibility and performance, and (ii) the solution to overcome these challenges for thousands of tenants.

**Scenario**

Our customer has an OLTP application product with a web interface for end users from their own customers (tenants) to interact through short-lived transactions supporting a business process. This product evolved from an on-premises product, which had an add-on allowing reporting on operational data through analytics-type queries; an ETL module populated a DW with a dimensional model. With the move to the cloud, the analytics module was also to become cloud based and multi-tenant.

The ETL module, therefore, was extended to make it work for multiple clients in the cloud from the newly implemented OLTP cloud application. It became a synchronous process working along the following lines:

- A small change data capture (CDC) process (based on DB table I/U/D triggers) was built writing the changes from the operational multi-tenant database into CDC tables in real time.

- Managed deployment on cloud infrastructure (at a minimum, hardware, OS, network and storage)

- A traditional ETL process running periodically (say, every 15-mins), read the changes from the CDC tables and loaded the data into a stage area, applied transformations and then wrote them finally to the DW.

Figure 1 - Initial solution diagram

The same transformations were applied to all the tenants in a synchronous pipeline. At the same time, the solution co-locates several hundred tenants per DW schema. We believe this is the Achilles heel of the solution, as it makes it difficult to deal with the pace at which tenants want to adopt new versions of the application, or to scale to ingest large data volumes updated at the source (which now may happen due to multi-tenancy co-location) with low latency. For this reason, this solution broke in less than 4-months of time in production and the data team was back to the drawing board. The following challenges were revealed by our root-cause analysis:
1. Schema version management for tenants

The source application(s) add features over a period of time, which leads to structural changes in the OLTP database schema. Making all tenants move in lockstep to the next version of the source application was considered too inflexible. Tenants want the freedom to move to the newest and greatest version at their own convenience, so ETL has to be aware of version deployed for a tenant; in addition, tenant movement to a new version should be swift with no data loss or corruption. The two following cases of OLTP schema changes exist:

a. When their structural changes are small – we refer to this as a “non-breaking change” below, we map two or more OLTP schema versions to a single DW schema version.

b. When there is a major (“breaking”) change, this induces a change in the DW version, which must then co-exist with the previous DW versions.

2. Schema changes

a. Entity relationships are dynamic in nature and can vary greatly from simple to very complex (deeply nested or hierarchical). Modeling these relationships precisely in the DW schema can be a very time-consuming process. New entities can be added or moved, new attributes to existing entities can be added or deleted i.e. it can be breaking or non-breaking change and ETL must be tolerant to schema-changes.

b. In addition to supporting structured data with evolving schemas over time, the solution must also support unstructured and semi-structured sources such as comments, photos, e-mail messages and weblogs that will be added soon to the OLTP application.

3. Scaling to volume and velocity of changes

a. Building all this at scale (100+ TB, with 10000+ tenants) becomes an even bigger challenge that requires ETL worker jobs to run at a rate that matches the data update frequency of the source(s).

b. The system must be able to re-consume the changed data in the event of ETL failures.

4. Provide overall low latency

Data must be available downstream as quickly as possible, close to real-time (measured in seconds, as opposed to minutes or hours).

What are the possible solutions?

There are different dimensions in play here:

- Architecture – uniform vs. dual (e.g. Lambda)
- ETL - cloud ETL services vs. Traditional ETL tools on IaaS vs. custom ETL programs
- DW - Big data stack (Apache Hadoop) vs. MPP databases

There can be many solutions combining elements from these three dimensions. Most of them would require some sort of custom code to handle volume and velocity of the changes. Our solution implemented a uniform architecture based on, a custom ETL pipeline moving data to an MPP database (AWS Redshift), as per the stack illustrated below.
What was implemented?

Figure 2 - working solution diagram

The key of the solution is to decouple the previous synchronous process in different asynchronous streams that

  i. can apply different transformation logic or schema (dependent on the tenant version subscribed to) and
  ii. can work in parallel, to solve both the flexibility and the scale challenges mentioned above.

How key challenges were addressed?

1. **Schema versioning**: A single queue serves the data from multiple tenant sources that subscribe to the same DW
   version, allowing us to solve the “versioning” challenge. Message queues like Apache Kafka or RabbitMQ or Azure
   Service Bus are good abstractions for such data pipelines.

2. **Overall, low latency** – The continuous data integration was realized by the asynchronous communication between
   multiple data sources and the DW through message queues and JSON message as the data exchange format. The
   queue was partitioned and many readers consumed messages simultaneously. Messages need not have to wait in ETL
   stage micro-batch for dependent message to arrive, as all the required dependencies to load, the DW is packed by source
   application in the message itself.

3. **Schema change management** – A thin layer of metadata was built to decode JSON messages containing changed
   data. The JSON message parser was generic enough to consume the messages in real-time, understand the entities
   and attributes, detect changes in schema and, based on the version of the messages, load data into appropriate tenant-
   specific stage tables.

   • Breaking schema changes were logged and notified and data remained in the queue for a configurable period
     to re-consume (until the database administrator develops a new DW schema version, developers apply pipeline
     logic and the tenant’s configuration is re-routed to the new pipeline).

   • Non-breaking changes also flagged, but the data landed in DW quickly.
Semi-structured sources can be supported in the future, thanks to this data pipeline. The JSON message format flexibility allows us (i) to decouple changed content from OLTP schema versions and (ii) to accommodate both changing relationships of structured data as well as semi-structured sources. Message consumers must read the appropriate nodes in the JSON message (for subscribed versions), and flatten the nested structure and push it in the form of rows in the stage tables. From here onwards, it was regular ETL to consume the messages and load the DW schema.

4. Scaling to volume and velocity of changes – The entire pipeline was distributed, achieving elasticity and scalability. Any number of data sources or consumers, from low to high volume, can be configured via automated scripts.

5. Messages arriving out-of-order: due to the asynchronous decoupling, it is likely that the insert message for an entity may be processed afterwards by message readers than a logically subsequent update on the same entity. A message re-order logic was used to tidy-up this, making sure ETL processes the messages in-order. It also ensured that we never lose an event.

Technical Stack

<table>
<thead>
<tr>
<th>Changed data transport</th>
<th>Application publishes messages to a Kafka topic in real-time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message queue</td>
<td>Apache Kafka</td>
</tr>
<tr>
<td>DW</td>
<td>Amazon AWS Redshift</td>
</tr>
<tr>
<td></td>
<td>Data distribution &amp; access strategy (DIST style = key</td>
</tr>
<tr>
<td></td>
<td>SORT style = aggregation or join columns)</td>
</tr>
<tr>
<td>Transformations</td>
<td>Custom Code, multiple readers packaged in AWS Lambda (λ)</td>
</tr>
<tr>
<td></td>
<td>functions</td>
</tr>
<tr>
<td>Stage Db</td>
<td>MySQL + AWS S3 (for unstructured data)</td>
</tr>
</tbody>
</table>

A few statistics of the solution currently deployed in production:

<table>
<thead>
<tr>
<th>Number of tenants</th>
<th>800</th>
</tr>
</thead>
<tbody>
<tr>
<td>OLTP Data Size</td>
<td>4 TB</td>
</tr>
<tr>
<td>DW storage</td>
<td>12 TB</td>
</tr>
<tr>
<td>Queue throughput</td>
<td>1200 K messages/min</td>
</tr>
<tr>
<td>Number of concurrent ETL load workers</td>
<td>40 to 50</td>
</tr>
<tr>
<td>Number of ETL nodes</td>
<td>8</td>
</tr>
<tr>
<td>ETL load window available</td>
<td>2 mins</td>
</tr>
<tr>
<td>Records processed</td>
<td>480 K</td>
</tr>
<tr>
<td>Avg. ETL load time</td>
<td>46 seconds</td>
</tr>
<tr>
<td>Avg. Latency</td>
<td>5 to 8 seconds</td>
</tr>
</tbody>
</table>
Obviously, this solution also has its own cons but it does address most of the issues we talked earlier.

**Current limitations of the solution**

1. Aggregations are not built in-flight and were generated in the DW post-load.

2. Custom ETL – on-boarding of new tenants and movement of tenants to new version requires maintenance of the ETL metadata and versions. This was automated to some extent using scripts.

3. Graceful recovery from failures – Component wise logging is not enough and we need a better automated monitoring of the data pipeline for any failures.

4. Schema migration for breaking changes is manual and some changes could be automated.

**The Takeaway**

This article talks about the challenges we faced while implementing a low latency, multi-tenant data warehouse at scale.

Starting from a single tenant architecture does not work. It is necessary to revisit the architecture when multiple tenants are involved – schema versioning will probably be needed, as well as elasticity and scalability to cope with the volume and velocity of changes. This was obtained by introducing queuing technology, allocating a single distributed, logical queue to serve data from multiple tenant sources subscribing to the same DW version, by decoupling queue writers and readers and by accommodating any number of writers and readers.
About Persistent Systems

Persistent Systems (BSE & NSE: PERSISTENT) builds software that drives our customers’ business; enterprises and software product companies with software at the core of their digital transformation. For more information, please visit: www.persistent.com

India
Persistent Systems Limited
Bhageerath, 402,
Senapati Bapat Road
Pune 411016.
Tel: +91 (20) 6703 0000
Fax: +91 (20) 6703 0009

USA
Persistent Systems, Inc.
2055 Laurelwood Road, Suite 210
Santa Clara, CA 95054
Tel: +1 (408) 216 7010
Fax: +1 (408) 451 9177
Email: info@persistent.com

DISCLAIMER: “The trademarks or trade names mentioned in this paper are property of their respective owners and are included for reference only and do not imply a connection or relationship between Persistent Systems and these companies.”

© 2017 Persistent Systems Ltd. All rights reserved.