Adaptive Database Synchronization for an Online Analytical Cloud-to-Edge Continuum

Daniel Costa
daniel.v.costa@inesctec.pt
INESC TEC and U. Minho
Portugal

José Pereira
jose.o.pereira@inesctec.pt
INESC TEC and U. Minho
Portugal

Ricardo Vilaça
ricardo.p.vilaca@inesctec.pt
INESC TEC and U. Minho
Portugal

Nuno Faria
nuno.f.faria@inesctec.pt
INESC TEC and U. Minho
Portugal

ABSTRACT

Wide availability of edge computing platforms, as expected in emerging 5G networks, enables a computing continuum between centralized cloud services and the edge of the network, close to end-user devices. This is particularly appealing for online analytics as data collected by devices is made available for decision-making. However, cloud-based parallel-distributed data processing platforms are not able to directly access data on the edge. This can be circumvented, at the expense of freshness, with data synchronization that periodically uploads data to the cloud for processing.

In this work, we propose an adaptive database synchronization system that makes distributed data in edge nodes available dynamically to the cloud by balancing between reducing the amount of data that needs to be transmitted and the computational effort needed to do so at the edge. This adapts to the availability of CPU and network resources as well as to the application workload.

CCS CONCEPTS

• **Information systems** → **Remote replication**; *Data federation tools.*

KEYWORDS

synchronization, cloud-edge environment, replication, data federation, analytical

ACM Reference Format:
Daniel Costa, José Pereira, Ricardo Vilaça, and Nuno Faria. 2022. Adaptive Database Synchronization for an Online Analytical Cloud-to-Edge Continuum. In *The 37th ACM/SIGAPP Symposium on Applied Computing (SAC '22), April 25–29, 2022, Virtual Event, .* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3477314.3507212

1 INTRODUCTION

Data analysis has traditionally been performed on dedicated systems, separate from operational databases, where data is collected and modified by a transactional workload. These online transaction processing (OLTP) systems focus on reliability and interactive performance. Periodically, data is extracted from operational systems, transformed to a representation amenable to analysis, and loaded into a second database that can then be used to run, often complex and resource-intensive queries. These online analytical processing (OLAP) systems emphasize the ability to consume large volumes of data for each operation.

In cloud systems, the freshness of the data that can be analyzed is improved by hybrid transactional analytic processing (HTAP) systems that are able to handle both workloads simultaneously from a single copy of data, avoiding the need for ETL.

However, such cloud-based parallel-distributed data processing platforms are not able to directly access data in the edge. The main reason for this is the inability of networks to cope with the volume of data that would reside in edge devices. Enlisting edge nodes as additional nodes in the parallel-distributed HTAP system would severely impact their performance by introducing a resource-constrained node with much higher latency.

The challenge is thus twofold: First, the data that is transferred from the edge to the cloud needs to be minimized by uploading only once what is needed to answer actual queries. Second, it needs to be done online, as a reaction to actual queries and not periodically, to keep up with what is expected from an HTAP system.

This paper tackles these challenges with a novel adaptive database synchronization system that makes distributed data in edge nodes available dynamically to the cloud by balancing between uploading recently modified data with pushing down query fragments, that perform part of the computation in the edge. Moreover, it adapts to the availability of CPU and network resources as well as to the application workload, allowing a reduction of the volume of data transferred to the cloud.

2 BACKGROUND AND ASSUMPTIONS

We consider a hybrid edge-cloud architecture – sketched in Figure 1 – as the basis of our system model as is commonly found in practice [4, 5]. The edge devices collect and pre-process data, storing them into their local databases, while the cloud servers run analytical distributed workloads over the data collected by the edge.

Each cloud server has access to each edge device's database by using federation wrappers, i.e., software components that hide network and serialization/deserialization details of remote data, making them appear as if they were local through a SQL interface.

This works as follows: A query submitted to the query engine involving a remote table is parsed as usual, producing a query plan. This plan determines the fetched data from remote data sources. We combine existing federation wrappers, for remote access, with a custom federation wrappers that manage local caching to reduce the latency of frequently accessed data, where each cloud server caches a snapshot of the data read.

SAC '22, April 25–29, 2022, Virtual Event,

ACM ISBN 978-1-4503-8713-2/22/04.
https://doi.org/10.1145/3477314.3507212

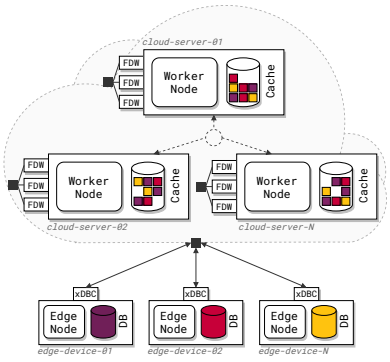

Figure 1: Overview of the system model.

3 APPROACH

Our proposed approach aims to reduce the impact of data replication on both network delays and load on the edge device. This means balancing concepts such as only sending the data necessary by the cloud at any given time to reduce transfer latencies and omitting some filters to reduce execution time on the edge.

3.1 Minimizing data transfer

To minimize data sent, we must not only avoid sending duplicate data, but also only send the rows required by the server that issued the replication. To solve the first challenge, we cache in the cloud server the data replicated, which also improves read latency on subsequent queries that reuse the same data. For the second challenge, we keep track of both the timestamps of the data received, and the query filters used by the cloud server. When the server issues a new query, it gets translated into an equivalent one which excludes the already cached data, resulting in the minimum amount of data transfer possible.

With a practical example, consider table *Demo*, whose schema and data is presented in Figure 2a.

Figure 2 displays the state of a cloud server's cache after successive queries. Newly retrieved rows are highlighted in yellow, already cached rows have a gray background, and remote-only rows have a white background. Initially, there are no cached rows in the cloud server (Figure 2a).

First, the server performs the query `SELECT * FROM Demo WHERE A = 1`. As there are no rows that were previously cached, the entire query is sent to the edge as is. The returned rows are cached in the cloud server. In addition to the data itself, we also store the filter and the maximum timestamp into the single predicate $A = 1 \wedge Ts \leq 4$, to be later combined with subsequent queries.

Next, the server performs the query `SELECT * FROM Demo WHERE B = 1`. Looking at Figure 2b, we see that we already have one row out of the two that will be returned by the query, which means it does not need to be transferred through the network. To accomplish this, we look at the previously stored predicates P and combine them with the current query Q, using the expression $Q \wedge \neg P$. In this case, this results in the query `SELECT * FROM Demo WHERE B = 1` **`AND NOT (A = 1 AND Ts <= 4)`**. Once again, we store the filter and the maximum timestamp, resulting in the exclusion predicate $(A = 1 \wedge\ Ts \leq 4) \vee (B = 1 \wedge Ts \leq 2)$.

Lastly, the cloud server performs the final query `SELECT * FROM Demo WHERE B = 1`. Between this query and the previous one, the edge device inserted the row $\langle 5, 5, F, 1, 1\rangle$ into *Demo*. Using again the same process, the current query is translated into `SELECT * FROM Demo WHERE B = 1` **`AND NOT ((A = 1 AND Ts <= 4) OR (B = 1 AND Ts <= 2))`**, which only returns the new row ($Id = 5$).

As more and more queries are issued by the server, the number of predicates could grow indefinitely. Therefore, there are issues related with the storage overhead in the cloud server and large query text that becomes increasingly more complex to parse and send through the network. To mitigate this, our algorithm employs a coalescing strategy to simplify stored filters.

With a practical example, consider again table *Demo*, with the same schema of Figure 2 but different data. When we execute the query `SELECT * FROM Demo WHERE A=1 AND B=1` and receive the maximum timestamp of 1, we store the predicate $A = 1 \wedge B = 1 \wedge Ts \leq 1$.

When we execute a second query `SELECT * FROM Demo WHERE A=1 AND C=1` and receive the maximum timestamp of 3, we store the additional predicate $A = 1 \wedge C = 1 \wedge Ts \leq 3$. However, storing both $A = 1 \wedge B = 1 \wedge Ts \leq 1$ and $A = 1 \wedge C = 1 \wedge Ts \leq 3$ is suboptimal, as $A = 1$ is stored twice. A more efficient way is to store $A = 1 \wedge ((B = 1 \wedge Ts \leq 1) \vee (C = 1 \wedge Ts \leq 3))$. Thus, the first optimization step is to remove repeated filters.

Consider now that we execute a third query `SELECT * FROM Demo WHERE A=1` and receive the maximum timestamp of 6. As the predicate $A = 1 \wedge Ts \leq 6$ encompasses all the data returned by the previous two predicates, we can simply store only this one. Similarly, if we execute fetch all table, we can simply remove all other stored filters. Thus, the second optimization step is to coalesce filters when they are subsets of other filters.

3.2 Filter pruning

We can perform filter simplifications to remove redundant filters. However, the complexity of the statement increases for each unique condition. If the query is complex enough, we might have a case where the execution on the edge device outweighs the reduced network costs. Therefore, a trade-off must be made, in which we consider removing some filters – (possibly) leading to higher data transfers – in favor of a faster execution in the edge.

To do so, our algorithm tries to balance the cost of filtering and the cost of transferring the data. Consequently, it requires parameters that define the relative costs of each operation. Firstly, the cost of filtering the data source can be defined as $c_f = c_c * c * r$ where c_f is the total cost of filtering, c_c denotes the cost of each condition, c represents the number of conditions, and r is the number of rows in the data source.

On the other hand, the cost of transferring the data can be defined as $c_t = c_b * r * w$ where c_t represents the total cost of transmitting the data, c_b denotes the cost per byte transferred, r expresses the number of rows, and w is the average width of each row in bytes. Finally, if the cost of filtering is greater than the required to transfer the data, some filters are superfluous.

In our algorithm, we eagerly evaluate the gains of removing each filter. We estimate the number of bytes returned from the filter by requesting the number of rows and the average width of the

Id	Ts	Deleted	A	B
1	1	F	0	0
2	2	F	0	1
3	3	F	1	0
4	4	F	1	1

(a) Inital state of table *Demo*.

Id	Ts	Deleted	A	B
1	1	F	0	0
2	2	F	0	1
3	3	F	1	0
4	4	F	1	1

(b) State of *Demo* after the first query `'SELECT * FROM Demo WHERE A = 1'`.

Id	Ts	Deleted	A	B
1	1	F	0	0
2	2	F	0	1
3	3	F	1	0
4	4	F	1	1

(c) State of *Demo* after the second query `'SELECT * FROM Demo WHERE B = 1'`.

Id	Ts	Deleted	A	B
1	1	F	0	0
2	2	F	0	1
3	3	F	1	0
4	4	F	1	1
5	5	F	1	1

(d) State of *Demo* after the new row $\langle 5, 5, F, 1, 1 \rangle$ and query `'SELECT * FROM Demo WHERE B = 1'`.

Figure 2: Cached data of table *Demo* after successive queries. The yellow rows represent the newly cached data, the gray rows represent the data already cached, and the white rows represent data not present in the cache.

data source. The estimated number of bytes can be expressed as $b_e = r_e * w_e$ where b_e is the estimated number of bytes, r_e denotes the estimated number of rows and w_e represents the estimated average width of each row. Firstly, we test the filter with the lowest expected byte return and remove it if it decreases the overall costs of the system. This is the case if $c * r < c_b * r_f * w$ where c is the marginal filtering cost of the filter per row, r is the estimated number of rows in the data source, c_b is the cost per byte transferred, r_f represents the marginal number of rows filtered by the filter, and w symbolizes the average width of each row. If so, the filter is removed and the next filter is tested. On the other hand, we stop the process if there are no more filters or the cost of removing the filter increases the overall cost.

One downside is that this process has associated compute costs. For example, estimating the number of rows is a compute-intensive process that requires connection to the data source. Filter pruning is executed exclusively if $c_f > c_t + c_e * f$ where c_f is the cost of filtering, c_t denotes the estimated cost of transferring the missing data, and c_e is the cost of estimating the number of rows per filter and f represents the number of stored filters.

4 IMPLEMENTATION

Our implementation was designed to answer analytical queries in the cloud and transaction queries in the edge. We use PostgreSQL as the cloud database engine. The analytical queries are parsed by PostgreSQL, which transfers the query plan to the Multicorn. The Multicorn passes the filters and projections to our modules.

The modules were implemted using Python. The cache module notifies the synchronization module of what data is required. This module uses `psycopg2` library to establish the connection to the edge. The sync module generates the query to be executed by the edge by combining the filters passed from PostgreSQL and the negation of previous stored filters.

After executing the query, it receives the data and updates the cache. Finally, it appends the new filters with the associated timestamp and coalesces the predicate. To store the filters and their timestamps, we use a SetTrie [6], which improves the performance of searching for subsets and supersets of filters and allows us to avoid storing repeated prefixes.

5 DISCUSSION

We presented a synchronization middleware component suitable to a cloud-edge environment that dynamically balances between filtering data in the edge and transferring more data to the cloud. Our proposal considers the edge's CPU capabilities, the network resources, and the application workload to reduce the execution time. Our proposition takes advantage of the filter pushdown capabilities of the data source to transfer only recently modified and requested data.

Existing database systems often provide mechanisms for asynchronous replication that uses the transactional log and the log sequence number (LSN) in each record to reduce the amount of data that needs to be transferred [1]. Another alternative uses the Bose–Chaudhuri–Hocquenghem (BCH) error-correcting code to determine differences between replicas [2]. The FSYNC [7] is a differential synchronization algorithm for storage. Another alternative is the use of Invertible Bloom Lookup Table [3]

The algorithm described in this paper is suitable to be used in an online analytical cloud-to-edge continuum because it is fitted to answer queries on-demand and reduce the data transferred from the edge. Besides, this component can decrease the impact of the analytical workload on the edge and the network.

ACKNOWLEDGMENT

Partially funded by project AIDA – Adaptive, Intelligent and Distributed Assurance Platform (POCI-01-0247-FEDER-045907) co-financed by the European Regional Development Fund (ERDF) through the Operational Program for Competitiveness and Internationalisation (COMPETE 2020) and by the Portuguese Foundation for Science and Technology (FCT) under CMU Portugal.

REFERENCES

[1] PostgreSQL developers. 2021. PostreSQL Documentation – Chapter 27. High Availability, Load Balancing, and Replication. https://www.postgresql.org/docs/14/high-availability.html.

[2] Yevgeniy Dodis, Leonid Reyzin, and Adam Smith. 2004. Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data. In *Advances in Cryptology - EUROCRYPT 2004*, Christian Cachin and Jan L. Camenisch (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 523–540.

[3] Michael T. Goodrich and Michael Mitzenmacher. 2015. Invertible Bloom Lookup Tables. arXiv:1101.2245 [cs.DS]

[4] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. 2013. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future generation computer systems* 29, 7 (2013), 1645–1660.

[5] Partha Pratim Ray. 2016. A survey of IoT cloud platforms. *Future Computing and Informatics Journal* 1, 1-2 (2016), 35–46.

[6] Iztok Savnik. 2013. Index Data Structure for Fast Subset and Superset Queries. In *Availability, Reliability, and Security in Information Systems and HCI*, Alfredo Cuzzocrea, Christian Kittl, Dimitris E. Simos, Edgar Weippl, and Lida Xu (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 134–148.

[7] Tian Wang, Jiyuan Zhou, Anfeng Liu, Md Zakirul Alam Bhuiyan, Guojun Wang, and Weijia Jia. 2019. Fog-Based Computing and Storage Offloading for Data Synchronization in IoT. *IEEE Internet of Things Journal* 6, 3 (2019), 4272–4282. https://doi.org/10.1109/JIOT.2018.2875915