Relation-Based In-Database Stream Processing

Christian Winter[1], Thomas Neumann[1] and Alfons Kemper[1]

[1] Technical University of Munich

Abstract

Data analytics pipelines are growing increasingly diverse, with relevant data being split across multiple systems and processing modes. In particular, the analysis of data streams, i.e., high-velocity ephemeral data, is attracting growing interest and has led to the development of specialized stream processing engines. However, evaluating complex queries combining such ephemeral streams with historic data in a single system remains challenging.

In this paper, we devise a novel stream processing technique that allows users to run ad hoc queries that combine streams and history tables in a relational database system. The backbone of our approach is a specialized ring-buffered relation, which allows for high ease of integration for existing database systems. We highlight the applicability of our approach by integrating it into the Umbra database system and demonstrate its performance against dedicated stream processing engines, outperforming them consistently for analytical workloads.

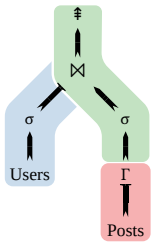
1. Introduction

There is an ever-increasing need for just-in-time analyses combining real-time data in the form of streams with information held in databases, such as user, customer, or billing data. Several solutions integrating durable data in stream processing engines (SPEs) have been proposed, either loading read-only data into the stream processing engine from local sources such as CSV files or offering interfaces to external databases [1, 2, 3, 4]. However, the reverse direction of integrating stream processing into relational databases has yet to receive much attention. While modern SPEs are capable tools for many workloads, they lack the functionality to manage historic data internally. We argue that the unmatched capabilities and performance of relational database systems for managing and analyzing relational data make them the ideal solution for processing durable relations and data streams.

In this paper, we devise a technique to integrate streams into database systems through a specialized streaming relation. By relying on a ring-buffered specialization of regular database relations, we can utilize the database system's full type and query support and gain access to a wide range of pre-built functionality and operators, such as efficient joins and string operations. Our approach relies on regular SQL to interact with streams, necessitating no changes to the database grammar. Thus, streams can be used with all tools commonly used for database access, such as object-relation mapping (ORM) libraries available for many programming languages. Furthermore, the SQL-based interface allows users to easily express queries incorporating streams and regular tables. We demonstrate the applicability of our approach by implementing it in the state-of-the-art database system Umbra [5].

```
with user_impact as (
  select    uid, avg(score) as score
  from      posts
  group by  uid
)
select      u.username, u.contact_info,
            u.rate, i.score
from        users u, user_impact i
where       u.id = i.uid
and         i.score >= 1000 and
            u.region = 'DE'
order by    u.rate / i.score
```

(a) SQL



(b) Query Plan

Figure 1: Exemplifying analytical query combining a stream and durable relations.

We outline the relation-based integration on an exemplifying workload, which we use as a running example throughout this paper: Consider a micro-blogging service where users can share and like simple text posts. For reporting, such a service might be interested in finding influential posters in a given region, e.g., users who achieved an average of at least 1'000 likes per post in the last month for promotional campaigns. Figure 1 depicts the corresponding query on an exemplifying schema. In many cases, business data, such as the contact info and payment details for paid bloggers, will be stored separately from the service data, such as posts. In the past, it was necessary to either find influential posters in the service database or materialize posts in the business database. The first option is undesirable as it involves analytical queries in a system likely optimized for simple lookup and update operations. In contrast, the second option would unnecessarily bloat the analytical database. On the other hand, our approach allows us to stream the

Joint Workshops at 49th International Conference on Very Large Data Bases (VLDBW'23) — Second International Workshop on Composable Data Management Systems (CDMS'23), August 28 - September 1, 2023, Vancouver, Canada

winterch@in.tum.de (C. Winter); neumann@in.tum.de (T. Neumann); kemper@in.tum.de (A. Kemper)

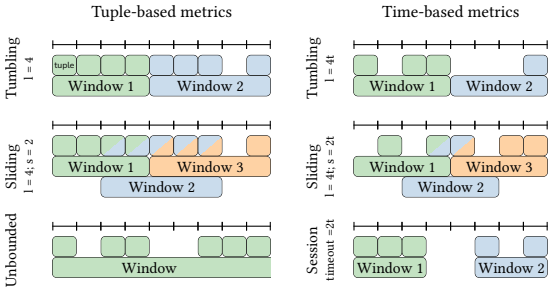


Figure 2: Overview of common stream windowing semantics.

last month's posts into the analytical database without materializing them, requiring no analytical functionality from the service database. This paper makes the following key contributions:

- We describe the integration of stream processing in database systems as a specialized ring-buffered relation.
- We discuss the streaming model achieved by our integration.
- We evaluate our streaming relation against dedicated stream processing engines, focusing on end-to-end query and insert performance on a TPC-H-based workload.

The remainder of this paper is structured as follows: In Section 2, we discuss background and related work in stream processing. Following, we discuss our approach to in-database stream processing in Section 3, which we evaluate in Section 4. Finally, we conclude in Section 5.

2. Background

Over the years, stream processing has evolved into a diverse area of research, spanning a wide variety of data stream models optimized for different applications. In this section, we outline the model underlying our work and discuss related work in the intersection between stream processing and relational database systems.

2.1. Stream Model

While data stream models differ in many aspects, we focus on the two main categories most relevant to our work: windowing semantics and state management. Windowing semantics determines which subset of the stream qualifies for query evaluation at any given moment. Among the most common windowing semantics are sliding, tumbling, session, and unbounded windows, shown in Figure 2. The first two semantics are further subdivided into time- and tuple-based metrics. Tumbling and sliding windows define a fixed window size l, either in terms of the number of tuples or in a duration t. While tumbling windows always advance by the full window size, sliding windows advance by a set length s, which can result in overlapping windows. Session windows are separated from one another by periods of inactivity where no tuple arrives. Finally, unbounded windows consider the entire stream for a query and are, thus, unsuitable for infinite streams.

The window semantic closest to regular relation processing is the unbounded window. However, this would mean that queries over infinite streams will never report a result as database queries only advance to the next step, i.e., operator, once an input is fully depleted. Therefore, we instead follow the session window semantics, which is still close to relation scan semantics, and assume a scan as depleted when no new stream tuples have arrived for the specified inactivity duration. Note that sliding and tumbling windows can still be achieved on top of this session window using the SQL `WINDOW` operator.

The second category, state management, determines where and how systems manage the state of streaming queries. This state mainly comprises intermediate query results, such as aggregates, but also includes routing and meta information, e.g., for worker and checkpoint management. In their survey, To et al. [6] identify four different state models for stream processing. Of those, we most closely follow the *operator view* of Fernandez et al. [7] wherein query progress and state are materialized within operators. However, due to Umbras pipeline-based query execution model, query state only occurs at pipeline breakers, not at all individual operators. Further, distributing tuples to workers is handled through morsel-driven parallelism [8] in our approach. Therefore, routing decisions for tuples are made by downstream operators pulling new morsels, not actively and push-based by upstream operators.

2.2. Related Work

Our approach overlaps with two primary research areas in data analytics: relational database systems and stream processing. Both have seen vast amounts of research, and we, therefore, focus our discussion of related work on their intersection.

Durable data in stream processing engines. Recent years have seen increased demand for analytics combining both historic and streamed data. Consequently, stream processing engines such as Apache Flink [2] and Apache Spark [1] enable the use of historic data in analytical queries over data streams. However, they do not support managing historic data internally and instead rely on external sources. These external sources can be file formats like Parquet and CSV or database systems through connectors such as JDBC.

While stream processing engines do not offer capabilities for managing historic data, modern SPEs manage state for long-running and complex queries internally [6, 9, 10]. To prevent conflicts between multiple queries on a shared state, some SPEs rely on transaction semantics commonly used by database systems [11, 12, 13]. TSpoon [14] extends Apache Flink [2] with a transaction model, thereby enabling a queryable state for data stream analytics at configurable isolation levels. In addition, Meehan et al. [15] build upon the OLTP database system H-Store [16] and utilize H-Store's transactional processing model for data streams, enabling the ACID-compliant execution of streaming and transactional database queries in a single system.

In-database stream processing. Combining streams and relational data in a single system has been proposed in the context of data warehouse architectures [17, 18]. However, these architectures rely on two separate engines for internal relational query and stream processing. Some works propose a unified SQL-based query language to express queries over both streams and durable relations easily [19, 20].

Past research integrating both stream processing and durable data in a single engine often rely on materialized views [21, 22] to realize continuous queries [23, 24, 25] through continuous views [26]. DBToaster [4, 27] implements higher-order incremental view maintenance in a standalone engine to enable high insert and query throughput for views combining both static and dynamic data. In addition, PipelineDB [28] supports stream processing in the full-fledged database system PostgreSQL using dedicated streaming views.

3. Approach

Having defined the theoretical streaming model of our ring-buffer-based in-database stream processing approach, we can describe its design and implementation. The core difference between our in-database stream processing to processing relational data is that streamed data is not fully and permanently materialized within the database at the start of a query. For regular queries, the data that a query is working on is determined by its transaction. Each transaction has a single state of the database that it will evaluate all queries on. To achieve this, all data must be fully materialized at the start of a query, and any parallel changes must be handled transparently. Streams, on the other hand, are not transactional. Stream entries are ephemeral and are only cached in the database for a short time. Queries, therefore, cannot rely on the availability of all stream elements for query evaluation. Furthermore, queries involving streams have to handle stream arrivals during a query.

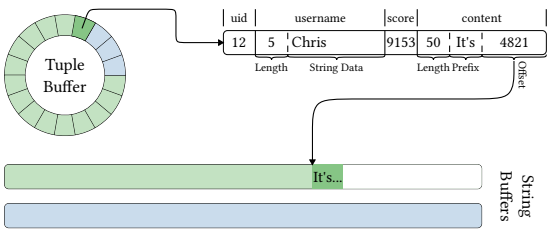


Figure 3: Overview of the caching layer consisting of a ring buffer for fixed size tuple parts and two string buffers used in alternation.

3.1. Interface

We want to rely purely on regular SQL grammar to interact with data streams. However, we must also ensure the database can infer which relations to treat as a stream and which to persist. For this, we follow a syntax similar to that of PipelineDB [28] and create streams as foreign tables with a reserved server name *stream*. For our example stream *posts*, this will result in the SQL statement:

```
create foreign table posts (
  uid       integer,
  username  varchar,
  score     integer,
  content   varchar
) server stream
```

Following the creation of this foreign table, all operations can be kept oblivious to the streaming nature of the relation. Inserts, therefore, can use regular SQL insert semantics, e.g.,

```
insert into posts values (
  12, 'Chris', 9153,
  'It's great to process streams
  in a database system!'
)
```

to insert the tuple displayed in Figure 3.

3.2. Caching Layer

Before discussing inserts and queries to data streams, we must establish how streaming data is stored within the database. Conceptually, ephemeral streams do not need to be materialized outside of queries and instead can be processed fully and directly on arrival. However, it is beneficial to cache chunks of the stream before processing. For one, a cache can compensate load spikes in the input stream where inserts occur too frequently for queries to keep up. Using a cache can help alleviate such short spikes by accepting tuples to be scanned by queries later on. Furthermore, a cache allows re-using existing scan logic and interfaces for durable relations, reducing the integration overhead and enabling efficient morsel-wise input processing.

For our approach, we implement a caching layer based on a ring buffer. Our buffer has two main components, displayed in Figure 3. The first component, the tuple buffer, stores fixed-size tuple data. This data includes all fixed-size columns, such as integer, numeric, and floating-point values, as well as metadata for variable-sized types, such as strings. In our example *posts* stream of Figure 1, these are the values for the integer columns *uid* and *score* and the metadata for the string columns *username* and *content*. The number of tuples to be held in this buffer can be configured to fit the expected load. By only storing metadata for variable-sized data and not the data itself [5], we ensure that all tuples have the same size, allowing us to easily re-use slots without checking for overlaps with still valid data.

The data for variable-sized types are instead stored in resizable buffers pointed at by the metadata. In the exemplifying tuple in Figure 3, one can see the two different storage formats for strings used by Umbra. Both formats first store the 4 byte length of the string. Short strings up to a length of 12 characters, such as this *username*, are stored inline in the remaining 12 bytes, requiring no additional buffer storage. For longer strings, here for the *content*, we store a 4 byte prefix and an offset into a separate storage region. For streams, this region is one of the string buffers. The prefix helps quickly answer some comparisons and filter predicates without loading the full string. Note that Umbra picks the string format for each individual string value, meaning a longer username for another tuple might be stored externally.

In contrast to the fixed-size tuple data, we cannot use a ring buffer for strings. When we would insert a string at slot n in a ring buffer longer than the string of the tuple previously cached at slot n, it would at least partially overwrite the string data of the tuple still held in slot $n+1$. This tuple is, however, still accessible through the cache. To prevent overwriting still valid and accessible tuples, we alternate between two resizeable string buffers for different runs through the ring buffer, using one for even and one for uneven runs. Alternating between buffers guarantees that string offsets are valid at least until the tuple is overwritten in the tuple buffer while avoiding expensive allocations for every individual string.

3.3. Insert Processing

Having outlined the layout of our caching layer, we can describe the insert process for stream tuples. While tuple-at-a-time inserts are also possible, we optimize for bulk inserts from an external streaming broker like Kafka [29] or SQL insert statements, e.g., reading from CSV files. The process is outlined in Algorithm 1 conceptually for a single tuple. In our implementation, we perform such inserts at morsel-granularity instead, collecting inserted tuples thread-locally before merging them into the relation in bulk for performance reasons. For each insert, we acquire a tuple identifier for the new tuple (Line 2). This id determines the ring buffer slot to store the tuple in. Furthermore, it determines the string buffer to be used for external strings. Before writing the first slot of the ring buffer, we additionally mark the corresponding string buffer for cleanup (Line 7). Note that while there are no longer any direct references into the string buffer from the ring buffer, we still do not free the memory region to not interfere with queries still processing buffer entries that were just overwritten. Instead, the last scan to finish on this string buffer will free the associated memory when it is completed. We will discuss string buffer memory management when discussing queries in Section 3.4.

Algorithm 1 Stream caching layer insert processing

```
1:  function processInsert(Tuple t)
2:      tid ← writeTid.fetchAdd(1)
3:      slot ← tid mod bufferSize
4:      odd ← ⌊tid / bufferSize⌋ mod 2
5:      stringBuffer ← stringBuffers[odd]
6:      if slot = 0 then
7:          stringBuffer.clear()
8:      for val ∈ tuple do
9:          if val.isString() then
10:             if stringLength(val) > 12 then
11:                 buffer.storeExtString(slot,
    stringBuffer.store(val))
12:             else
13:                 buffer.storeStringInPlace(slot, val)
14:         else
15:             buffer.store(slot, val)
16:     /* Delay scan visibility until all previous tuples are visible */
17:     while validTuples < tid - 1 do
18:         wait()
19:     validTuples ← tid
```

Following, we write the tuple data into the ring buffer slot. For strings, we decide between the two storage layouts outlined in Section 3.2 based on the string length. All other values are stored in-place in the ring buffer. Finally, we mark the tuple as valid to make it visible for scans (Line 19). To prevent partially-written cache entries of parallel inserts from being accessible for scans, we only mark new tuples as visible once all previous tuples are visible.

3.4. Query Processing

Through our specialized relation, the only operator in a query plan aware of an input's streaming nature is the table scan. All other operators can be kept oblivious about the nature of their input. While this integration is minimally invasive, it requires a careful design of the scan operator. Scans of regular relations rely on table metadata to determine the range of tuples to scan at query planning time which further determines the boundaries for the scan at execution time. For streams, however, we cannot rely on the scan boundaries to be known as tuples will

Algorithm 2 Stream scan operator morsel picking

```
1:  function selectScanRange
2:      morsel ← {}
3:      while now() − lastPick.load() < timeout do
4:          limit ← validTuples.load()
5:          position ← lastScanned.load()
6:          loop
7:              if limit ≤ position then
8:                  updateLimit()
9:                  break
10:             lastPick.exchange(now())
11:             if pickRange(morsel, limit, position) then
12:                 return morsel
13:     return ScanDone
```

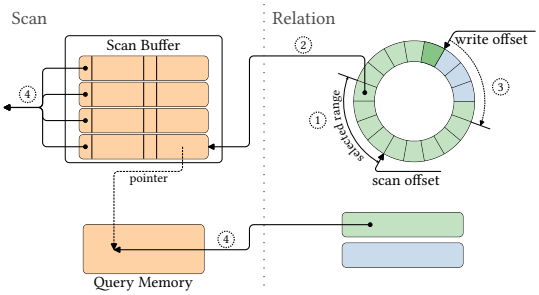


Figure 4: Overview of the four phases of the stream scan operator. Locating the desired range in the ring buffer (1), copying the range to the scan-local buffer (2), checking for potential conflicts (3) and reporting the tuples to the downstream operator (4).

still arrive during the query execution. Even cardinality estimates for query optimization can be unreliable as past stream behavior does not necessarily reflect future behavior. Therefore, we need to adapt query processing in two areas to handle streams efficiently: query planning and scan operator design.

3.4.1. Query Planning

For query planning, especially for join ordering, database systems rely on cardinality estimates for scans and filter predicates. These estimates are sourced from statistics maintained by inserts and updates to the relation. We cannot assume that previous statistics are available and reliable for streams. However, we still want the optimizer to produce an optimized query plan, especially to reduce materialized intermediate result sizes in the case of high-volume data streams. We assume streams are of the largest cardinality and, therefore, want to only materialize them if necessary. While the optimized query plan of Figure 1b only materializes the aggregated scores per user, an unoptimized plan without statistics might first join the *users* and *posts* relations, potentially materializing the *posts* stream in the join hash table. To avoid this, we hint to the optimizer that streams will always comprise the most data, thus ordering them to the probe side of joins.

3.4.2. Scan Operator

In contrast to scan operators for durable relations, our stream scan operator has to mask two things: unknown input bounds and ephemerality of tuples. As we, apart from the scan, entirely rely on existing database operators for query processing, we also have to adhere to the execution model of the database system. In our system Umbra, this is the producer-consumer model [30]. Generally, database execution models will process a blocking operator entirely before starting work on the next. Streaming models, on the other hand, replace this blocking semantics with running or windowed aggregates. To achieve similar semantics for database systems, we must determine when we can move query processing to the next operator. There are two different possibilities to achieve this: Sending a dedicated end-of-stream tuple or message signaling that the input is depleted or detecting input depletion from metadata, such as arrival rate. We focus on the latter, which is most consistent with our approach of handling streams transparently using an SQL interface. All tuples arriving after detected depletion can be handled in a new window for the query or are considered not part of the stream.

For simplicity, we rely on session window semantics to achieve this behavior, advancing to the next step of query processing when no new tuples have arrived for a predefined timeout. For our integration into Umbra, we integrate this session window semantic into the morsel selection [8] where range-based metrics reside for regular scans. Algorithm 2 outlines the resulting strategy. Before obtaining a scan range, we check the timeout condition (Line 3). If no thread detected new arrivals during this timeout, we continue with the next query processing task as we consider the stream input depleted. Following, we fetch the latest range information and check if tuples are available for processing (Line 7). If not, we return to the timeout check of line 3. Once tuples are available, we update the timeout condition (Line 10) and try to pick a morsel. Note that the unchecked change to the timeout condition can lead to a slight imprecision for the timeout, as it might lead to the loss of a more recent timestamp. However, we deem this acceptable as it allows us to reduce synchronization overhead.

We still must mask the second property of streams, their ephemerality. The ephemeral nature of streaming data does not impact the range selection outlined above, which is performed entirely on tuple ids. However, once the scan tries to access the corresponding values in the tuple buffer, we must ensure that the scan can never encounter values overwritten by concurrent inserts. This is especially important for string values with externally

stored data, such as the *content* value of Figure 3, where trying to access an invalid value could lead to a segmentation fault. Figure 4 depicts our scan strategy. First (1), we find the selected range in the buffer based on the tuple ids and prevent the deletion of the corresponding string region through reference counting. Following (2), we copy all fixed-size tuple data in the range into a separate buffer residing in the scan operator. After copying all tuples in the range, we check to see if concurrent writes have overwritten any tuples that we have scanned (3). In case of an overlap, we cannot guarantee that all tuples in the scan buffer are the desired tuples and, thus, have to abort the scan. If all scanned tuples are still valid, we report them to the downstream operator of the pipeline (4). At this stage, the first downstream operator materializing the tuples relocates the out-of-place content for strings into query memory, thereby protecting them from deletion and preventing segmentation faults in case of concurrent inserts. After all tuples of the scan range were processed by the downstream operators of the scan's pipeline, we release our reference to the relation's string buffer, freeing the corresponding memory region if we held the last reference.

4. Evaluation

Having outlined our relation-based approach to in-database stream processing, we evaluate its performance against two popular dedicated stream processing engines, Apache Flink [2] and Apache Spark [1]. We focus our evaluation on data ingestion rates, scalability, and performance on a mix of simple stream aggregation and complex analytical queries.

4.1. Setup

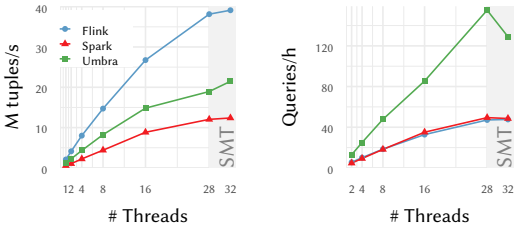
We perform all experiments in this section on a server equipped with 256 GB DDR3 main memory and an Intel Xeon E5-2660 v2 CPU with 28 physical cores. All data for the experiments is stored on a Samsung 970evo NVME SSD. Results reported in this section are based on the geometric mean of 5 runs taken after 2 warmup runs.

Workload. We base our experiments on the TPC-H benchmark [31] at scale factor 100. To transform TPC-H to a stream analytics workload, we consider the largest relation by far, *lineitem*, to be a stream. All other relations are considered durable and materialized at the start of a query. Consequently, we only include queries with exactly one scan of the lineitem relation in our experiments. Due to issues with Flink, we had to remove queries 5 and 8 from our benchmark. We do not print the query result in any of the systems.

Flink. We implement a standalone Flink executable based on Flink version 1.6.1 and express all relations using the batch table API based on CSV data. Further, we allow Flink's optimizer to re-order joins by setting the `TABLE_OPTIMIZER_JOIN_REORDER_ENABLED` flag. We submit all queries to Flink using its SQL interface. All intermediate results that Flink requires for processing are located in an in-memory file system.

Spark. We implement our TPC-H-based workload in Spark version 3.3.2. All queries are expressed using the Spark SQL API on external CSV data frames. Jobs are submitted to a local standalone spark cluster using `spark-submit`.

Umbra. We implement a streaming relation in Umbra as outlined in the previous section. Further, we create all relations except for *lineitem* as durable relations in Umbra. Lineitem is created as a stream using the interface described in Section 3.1. We subtract the session window timeout from Umbra's query runtimes as the system is idle during this period. Both Flink and Spark are run in non-windowed configuration and are, thus, not introducing similar delays.



(a) Insert (b) Query

Figure 5: Insert and query throughput in Umbra, Flink and Spark.

4.2. Stream Ingestion

As a first experiment, we examine the data ingestion rate offered by the three systems. For this, we fully insert the lineitem relation once into each system. As Spark and Flink rely on pull-based semantics for efficient analytical queries and do not offer full support for push-based inserts, we express inserts to them as `SELECT COUNT(*) FROM lineitem` queries. In contrast, we rely on push-based semantics as this never delays inserting workers and, therefore, imposes fewer requirements on inserting systems. For Umbra, we use the bulk insert command `COPY lineitem FROM CSV`.

Figure 5a shows the insert performance in millions of tuples per second along the number of insert threads. All systems show near-linear scalability until simultaneous multithreading (SMT) is reached. Overall, Flink offers the best insert performance, outperforming Umbra by a factor of 1.8 for 32 threads. This advantage can be attributed

Table 1
Speedup of using a stream minimized to Q6-relevant columns compared to all columns at 32 worker threads.

Approach	Insert	TPC-H Q6
Flink	1.52×	1.21×
Spark	2.19×	1.94×
Umbra	2.44×	2.43×

to the slight difference in the semantics of our insert queries. Our approach has to fully process all columns to materialize them in the ring buffer. In contrast, Flink can simply count the number of rows without parsing them entirely. While we expect this advantage to disappear for more complex queries where multiple columns must be parsed, it is very beneficial for such simple workloads. Furthermore, the lineitem relation comprises far more columns than are used by the average query. In our running example of Figure 1, we would, of course, only stream the necessary columns into the system, which benefits both Spark and Umbra. Table 1 shows the resulting speedup of inserting only the four columns relevant for TPC-H Q6 over a full lineitem insert. While Flink also benefits from less data being processed, Umbra and Spark can more than double their insert throughput. For this reduced lineitem stream, Umbra almost closes its insert performance gap with Flink.

4.3. Query Performance

Having analyzed the data ingestion capabilities of all approaches, we now focus on their analytical capabilities. For this, we run a combined workload of the twelve TPC-H queries selected for our benchmark, running each query once. For Umbra, the specified number of threads is the total number available to the system, which must be shared between query and insert processing. We concurrently schedule two queries in Umbra, one inserting the lineitem stream from CSV and another evaluating the TPC-H query on the inserted stream. Figure 5b shows the throughput in queries per hour for all three systems. Even though Umbra has to split the available workers between insert and query processing queries, it consistently outperforms both Flink and Spark, independent of the number of available worker threads. Furthermore, we see that the advantage Flink had when ingesting data into the system does not transfer to query analytics, where multiple columns have to be parsed. Furthermore, we can see scale-up issues with Flink as its relative performance degrades with increasing thread count, being overtaken by Spark when using more than eight threads. Having considered the analytical performance of all approaches for the full lineitem stream, we want to investigate the influence of processing only the columns required for a

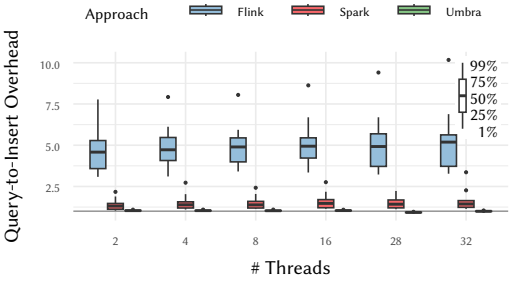


Figure 6: Overhead of queries to data ingestion for Flink, Spark and Umbra along increasing thread count.

query. Table 1 shows the speedup in query performance for TPC-H Q6 for a stream of only the four required columns. As for inserts, we can see that Umbra and Spark benefit more from this minimized stream than Flink, with Umbra achieving the highest speedup. Finally, we want to investigate the overhead that queries introduce in the system on top of the work for data ingestion. Figure 6 shows the relative overhead that evaluating our TPC-H-based benchmark creates for each system. The overhead of Spark is nearly constant, independent of the number of threads used. However, Flink's queries scale worse than its inserts. The drastically higher overhead for Flink confirms our assumption of Section 4.2 and indicates that Flink heavily optimizes for the `count(*)` query that we used to emulate inserts. For Umbra, we see next to no overhead when executing queries in addition to inserts. This further highlights the advantage of streaming only required columns into the database, as the speedup we have seen for inserts in Table 1 fully translates to query speedup.

5. Conclusion

In this paper, we devised a technique for relation-based stream processing in relational database systems. Relying on a ring-buffered relation for stream processing provides high ease of integration for existing database systems, enabling database systems to handle stream-enrichment queries combining transient with durable data. To demonstrate the applicability of this relation, we integrated it into the code-generating Umbra database system.

Using the implementation within Umbra, we demonstrated the performance of our streaming relation in a number of end-to-end benchmarks against dedicated stream-processing engines. Our approach consistently outperforms dedicated stream processing engines on analytical streaming workloads while requiring only minimal changes to the database system's execution model.

References

[1] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, I. Stoica, Apache spark: a unified engine for big data processing, Commun. ACM 59 (2016) 56–65.

[2] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, K. Tzoumas, Apache flink™: Stream and batch processing in a single engine, IEEE Data Eng. Bull. 38 (2015) 28–38.

[3] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, J. C. Platt, J. F. Terwilliger, J. Wernsing, Trill: A high-performance incremental query processor for diverse analytics, Proc. VLDB Endow. 8 (2014) 401–412.

[4] C. Koch, Y. Ahmad, O. Kennedy, M. Nikolic, A. Nötzli, D. Lupei, A. Shaikhha, Dbtoaster: higher-order delta processing for dynamic, frequently fresh views, VLDB J. 23 (2014) 253–278.

[5] T. Neumann, M. J. Freitag, Umbra: A disk-based system with in-memory performance, in: CIDR, www.cidrdb.org, 2020.

[6] Q. To, J. Soto, V. Markl, A survey of state management in big data processing systems, VLDB J. 27 (2018) 847–872.

[7] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, P. R. Pietzuch, Integrating scale out and fault tolerance in stream processing using operator state management, in: SIGMOD Conference, ACM, 2013, pp. 725–736.

[8] V. Leis, P. A. Boncz, A. Kemper, T. Neumann, Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age, in: SIGMOD Conference, ACM, 2014, pp. 743–754.

[9] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, R. H. Campbell, Stateful scalable stream processing at linkedin, Proc. VLDB Endow. 10 (2017) 1634–1645.

[10] B. D. Monte, S. Zeuch, T. Rabl, V. Markl, Rethinking stateful stream processing with RDMA, in: SIGMOD Conference, ACM, 2022, pp. 1078–1092.

[11] S. Zhang, J. Soto, V. Markl, A survey on transactional stream processing, CoRR abs/2208.09827 (2022).

[12] I. Botan, P. M. Fischer, D. Kossmann, N. Tatbul, Transactional stream processing, in: EDBT, ACM, 2012, pp. 204–215.

[13] P. Götze, K. Sattler, Snapshot isolation for transactional stream processing, in: EDBT, OpenProceedings.org, 2019, pp. 650–653.

[14] L. Affetti, A. Margara, G. Cugola, Tspoon: Transactions on a stream processor, J. Parallel Distributed Comput. 140 (2020) 65–79.

[15] J. Meehan, N. Tatbul, S. Zdonik, C. Aslantas, U. Çetintemel, J. Du, T. Kraska, S. Madden, D. Maier, A. Pavlo, M. Stonebraker, K. Tufte, H. Wang, S-store: Streaming meets transaction processing, Proc. VLDB Endow. 8 (2015) 2134–2145.

[16] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. B. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, D. J. Abadi, H-store: a high-performance, distributed main memory transaction processing system, Proc. VLDB Endow. 1 (2008) 1496–1499.

[17] Y. Watanabe, S. Yamada, H. Kitagawa, T. Amagasa, Integrating a stream processing engine and databases for persistent streaming data management, in: DEXA, volume 4653 of *Lecture Notes in Computer Science*, Springer, 2007, pp. 414–423.

[18] K. Nakabasami, T. Amagasa, S. A. Shaikh, F. Gass, H. Kitagawa, An architecture for stream OLAP exploiting SPE and OLAP engine, in: IEEE BigData, IEEE Computer Society, 2015, pp. 319–326.

[19] N. Jain, S. Mishra, A. Srinivasan, J. Gehrke, J. Widom, H. Balakrishnan, U. Çetintemel, M. Cherniack, R. Tibbetts, S. B. Zdonik, Towards a streaming SQL standard, Proc. VLDB Endow. 1 (2008) 1379–1390.

[20] E. Begoli, T. Akidau, F. Hueske, J. Hyde, K. Knight, K. L. Knowles, One SQL to rule them all - an efficient and syntactically idiomatic approach to management of streams and tables, in: SIGMOD Conference, ACM, 2019, pp. 1757–1772.

[21] O. Shmueli, A. Itai, Maintenance of views, in: SIGMOD Conference, ACM Press, 1984, pp. 240–255.

[22] H. Gupta, Selection of views to materialize in a data warehouse, in: ICDT, volume 1186 of *Lecture Notes in Computer Science*, Springer, 1997, pp. 98–112.

[23] D. B. Terry, D. Goldberg, D. A. Nichols, B. M. Oki, Continuous queries over append-only databases, in: SIGMOD Conference, ACM Press, 1992, pp. 321–330.

[24] S. Babu, J. Widom, Continuous queries over data streams, SIGMOD Rec. 30 (2001) 109–120.

[25] L. Liu, C. Pu, R. S. Barga, T. Zhou, Differential evaluation of continual queries, in: ICDCS, IEEE Computer Society, 1996, pp. 458–465.

[26] C. Winter, T. Schmidt, T. Neumann, A. Kemper, Meet me halfway: Split maintenance of continuous views, Proc. VLDB Endow. 13 (2020) 2620–2633.

[27] M. Nikolic, M. Dashti, C. Koch, How to win a hot dog eating contest: Distributed incremental view maintenance with batch updates, in: SIGMOD Conference, ACM, 2016, pp. 511–526.

[28] PipelineDB, PipelineDB - high-performance time-series aggregation for postgresql, 2023. URL: https://github.com/pipelinedb/pipelinedb.

[29] J. Kreps, N. Narkhede, J. Rao, Kafka: A distributed messaging system for log processing, in: Proc. NetDB, volume 11, 2011, pp. 1–7.

[30] T. Neumann, Efficiently compiling efficient query plans for modern hardware, Proc. VLDB Endow. 4 (2011) 539–550.

[31] T. P. P. C. (TPC), Tpc benchmark h: Standard specification, 2021. URL: http://www.tpc.org/.