Meet Me Halfway: Split Maintenance of Continuous Views

Christian Winter, Tobias Schmidt, Thomas Neumann, Alfons Kemper
Technische Universität München
{winterch, tobias.schmidt, neumann, kemper}@in.tum.de

ABSTRACT

From Industry 4.0-driven factories to real-time trading algorithms, businesses depend on analytics on high-velocity real-time data. Often these analytics are performed not in dedicated stream processing engines but on views within a general-purpose database to combine current with historical data. However, traditional view maintenance algorithms are not designed with both the volume and velocity of data streams in mind.

In this paper, we propose a new type of view specialized for queries involving high-velocity inputs, called *continuous view*. The key component of continuous views is a novel maintenance strategy, splitting the work between inserts and queries. By performing initial parts of the view's query for each insert and the remainder at query time, we achieve both high input rates and low query latency. Further, we keep the memory overhead of our views small, independent of input velocity. To demonstrate the practicality of this strategy, we integrate continuous views into our Umbra database system. We show that split maintenance can outperform even dedicated stream processing engines on analytical workloads, all while still offering similar insert rates. Compared to modern materialized view maintenance approaches, such as deferred and incremental view maintenance, that often need to materialize expensive deltas, we achieve up to an order of magnitude higher insert throughput.

PVLDB Reference Format:
Christian Winter, Tobias Schmidt, Thomas Neumann and Alfons Kemper. Meet Me Halfway: Split Maintenance of Continuous Views. *PVLDB*, 13(11): 2620-2633, 2020.
DOI: https://doi.org/10.14778/3407790.3407849

1. INTRODUCTION

The ever-growing volume and velocity of data, generated from sensors, networks, and business processes, are accompanied by an increasing demand for analytics on this data. Currently, analysts have to choose from a range of systems, depending on the type of workload they face. Traditionally,

Proceedings of the VLDB Endowment, Vol. 13, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3407790.3407849

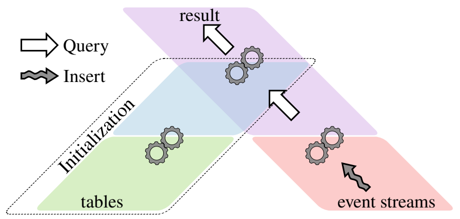


Figure 1: Exemplary split of a continuous view query plan for maintenance. Parts that are evaluated for each insert are marked with a grey arrow. Parts with a white arrow are evaluated for each query to the view. The framed part is evaluated once at view creation time.

such analyses have taken place in databases and data warehouses focusing on analytical query throughput. However, data is increasingly created in a continuous fashion and it is preferable to analyze it in real-time. Due to the sheer volume of data, storing it for later analysis in a data warehouse is often undesired or even infeasible. In addition, the after-the-fact analysis in dedicated warehouses cannot fulfill the demands for fast insights into the data, often referred to as real-world awareness.

For this emerging workload of high-velocity data, we propose a new type of view, which we call *continuous view*. These views are split for maintenance between inserts and queries, as outlined in Figure 1. By splitting the maintenance, we achieve the insert rates required for real-time stream processing, while simultaneously supporting fast analytical queries. We integrate continuous views into our newly developed Umbra database system [38]. Continuous views exploit Umbra's state machine-based approach on query execution to split the query efficiently for maintenance. Inserts perform the first logical steps for each arriving tuple, i.e., the necessary calculations for the first logical query pipeline (red). Only at times when the result of the view query is required are the remaining processing steps for the changes performed (purple).

In contrast, specialized systems for high-velocity data, often called Stream Processing Engines (SPEs), do not offer functionality to internally manage and modify historic data. Besides, many state-of-the-art SPEs such as Flink [8], Spark [50], and Storm [46] fail to properly utilize the underlying modern hardware, as shown by Zeuch et al. [51].

Modern view maintenance strategies such as higher-order incremental view maintenance (IVM) [24], on the other hand, are optimized for query-heavy workloads. While they offer high view refresh rates to better cope with high-velocity data, they refresh views on each insert and thereby optimize the query performance. By always having the current state of the view available, they trade insert for query performance. However, we argue that these constant full refreshes are not required for many insert-heavy streaming use cases. We will support this argument by using a small contrived example of an Industry 4.0 manufacturing plant, which we will also use as a running example throughout this paper:

Example: Consider a modern just-in-time car manufacturing plant that aims at keeping parts stored for as little time as possible. Parts are delivered to workstations just when they are needed for production. To avoid costly production halts, a car company might monitor the resupply needs with a query such as shown in Figure 2. Usage and resupply of parts at workstations are tracked automatically using the stream relations *part_usage* and *part_restock* for which exemplary entries can be found in Figure 3. The query reports those workstations where parts are running low, indicating problems in the supply chain. It is sufficient to check the current query result each second, or even only a few times per minute, to detect problems in time. Inserts, on the other hand, happen regularly, continuously, and with high velocity and can easily reach the tens or even hundreds of thousands per second.

Traditional view maintenance algorithms would update the at times expensive deltas for each tuple and refresh the view fully, even when the current result is not required. SPEs, on the other hand, are able to monitor the difference in parts for high insert rates. However, they lack the functionality to automatically combine the result with the necessary information held in databases without any additional overhead. In our example, SPEs would miss details about the station and the supplier in charge which are crucial for the query.

Our integration allows the user to access continuous views in regular database queries, and the underlying continuous queries have access to the database state just like regular views. Further, continuous views can be created using standard SQL. Umbra uses data-centric processing and the producer-consumer model [37] for compilation. Using the same concept for our continuous views, we keep the overhead of stream inserts low to compete with SPEs while simultaneously eliminating the need for a full stream or table scan at query time. In the categories defined by Babu et al. [4], continuous views are result-materializing queries with updates and deletes. While some research has been conducted on IVM maintained stream views in database systems [49], and continuous views also exist in open-source projects [42], we are not aware of other work proposing specialized view maintenance strategies for high-velocity stream workloads. Our contributions are as follows:

- We present a novel strategy for view maintenance, especially for views on stream inputs, that divides the work between inserts and queries. Our approach can support extensive analytical queries on streams without having to materialize the full stream input.
- We integrate stream processing using continuous views into our general-purpose database, using the query optimizer as well as specialized code generation to fully utilize the underlying hardware.

```
with used as (
  select     part, station, count(*)
  from       part_usage
  group by   part, station
),
restocked as (
  select     part, station, count(*)
  from       part_restock
  group by   part, station
)
select       r.part, s.location, s.supplier
from         station s, used u, restocked r
where        s.id = u.station and
             u.station = r.station and
             u.part = r.part and
             r.count - u.count < 5
```

Figure 2: Continuous query monitoring part supply on workstations within a manufacturing plant.

part_usage			
part_id	part	station	worker
1	Wheel	5	4
2	Seat	2	2
3	Seat	2	2
...	...	...	...

part_restock			
part_id	part	station	worker
1	Wheel	5	1
2	Seat	2	6
6	Engine	1	3
...	...	...	...

station			
id	plant	location	supplier
1	Spartanburg	Gate 2 - EN	+1 555 ...
2	Spartanburg	Gate 1 - SE	+1 555 ...
3	Regensburg	Tor 2	+49 555 ...
...	...	...	...

Figure 3: Exemplary *part_usage*, *part_restock* and *station* relations. *station* is considered a static table, *part_usage* and *part_restock* are streams.

- We describe how continuous views are created from regular SQL statements using only standard operators, thus allowing easy integration into existing systems.
- We demonstrate the capabilities of our system using the AIM benchmark [7], comparing it to state-of-the-art SPEs. We evaluate the performance limits using microbenchmarks and offer a comparison to the scale-up SPE Trill [10] and a database engine implementing higher-order IVM, DBToaster [24], on TPC-H.

The rest of this paper is structured as follows: In Section 2 we give a brief overview of the Umbra system developed by our group, focusing on the aspects relevant for this work. Afterward, we describe our novel split maintenance strategy for continuous views in Section 3. Section 4 shows the capabilities of our approach against state-of-the-art baselines and on a microbenchmark. In Section 5, we discuss relevant related work before concluding our paper in Section 6.

2. BACKGROUND

As stated in the introduction, the continuous views are integrated into our Umbra database system [38] and exploit some of its concepts to achieve fast and split view maintenance. While these concepts are outside the scope of this paper, we briefly describe them in this section to help the reader better understand our approach. Like its predecessor HyPer [37], Umbra compiles query plans into machine code. However, in contrast to HyPer, Umbra does not produce a monolithic piece of code and instead relies on a state machine to represent the different states of query execution.

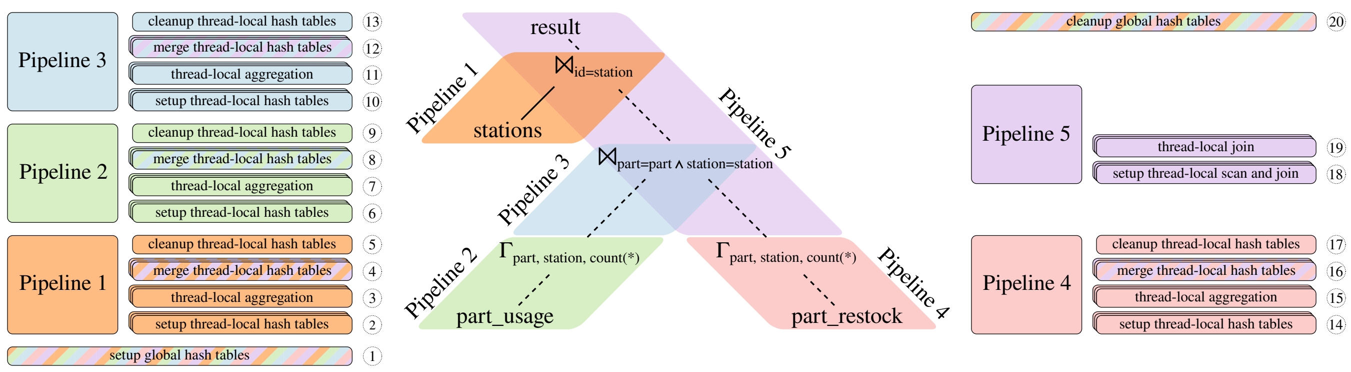

Pipeline 3
cleanup thread-local hash tables 13
merge thread-local hash tables 12
thread-local aggregation 11
setup thread-local hash tables 10
Pipeline 2
cleanup thread-local hash tables 9
merge thread-local hash tables 8
thread-local aggregation 7
setup thread-local hash tables 6
Pipeline 1
cleanup thread-local hash tables 5
merge thread-local hash tables 4
thread-local aggregation 3
setup thread-local hash tables 2
setup global hash tables 1
result
⋈id=station
stations
Pipeline 1
Pipeline 5
⋈part=part ∧ station=station
Pipeline 3
Γpart, station, count(*)
Pipeline 2
part_usage
Γpart, station, count(*)
Pipeline 4
part_restock
cleanup global hash tables 20
Pipeline 5
thread-local join 19
setup thread-local scan and join 18
Pipeline 4
cleanup thread-local hash tables 17
merge thread-local hash tables 16
thread-local aggregation 15
setup thread-local hash tables 14

Figure 4: Pipelines and corresponding logical steps for the query of Figure 2, joining two grouped streams and a durable table. Pipelines containing streams are dashed. Striped coloring indicates state shared between multiple pipelines is used. Colors indicate involved pipelines.

States of the state machine correspond to parts of the generated code. Transitions switch between code fragments, and thus between different phases of query execution. In the following, we will refer to the states and their associated code fragments as *steps*. Steps are responsible for processing parts of a query pipeline. A pipeline is a path in the logical query plan wherein a tuple does not have to be materialized in between operators, e.g., between the scan of *part_usage* and the grouping in pipeline 2 of Figure 4. The pipeline is composed of the logical steps for evaluating the *used* CTE of Figure 2, a simple aggregating group by. Umbra splits these steps into even finer-grained steps and might execute them in a slightly different order. However, this granularity is sufficient to describe our approach to split view maintenance. We will look at the categories of steps that can be commonly found in compiled query plans in more detail after briefly describing the memory regions we keep in Umbra.

Umbra differentiates between two kinds of memory that steps have access to: local and global state. Local state is used for intra-pipeline communication and state-keeping within steps and is not shared between pipelines. It is, however, possible for multiple steps of the same pipeline to access the same local state. Thus, steps within a pipeline have to be executed in the pre-defined order. One thread at most accesses a local state at any one time, meaning that there are several local state instances per pipeline for parallel tasks. Local state is designed to be inherently thread-local. Global state is used to share information between pipelines, and all threads working on a query have access to the same instance of the global state. Therefore, modification on the global state within parallel steps, e.g., when merging thread-local hash tables, requires data-dependent synchronization.

Steps are the basic building blocks that we use to define the different parts of our maintenance strategy. Each pipeline has four logical steps. In regular queries, ignoring recursive ones, each of these steps is traversed once, one after the other. However, not all of the steps are required for all pipelines. Furthermore, some of the steps, like the thread-local aggregation or the thread-local join in Figure 4, are run multiple times for different parts of the input, but we still consider them to be one logical step. To better illustrate the role of the steps in queries, we reference the steps of the query in Figure 2, shown in Figure 4.

Initialization. In the first step of each pipeline, we initialize all data structures relevant for in-pipeline processing. This includes temporary tuple buffers, simple data structures such as hash maps, as well as helpers for complex operators. In our running example, this includes steps ⑥ and ⑭ to set up the hash tables used for grouping, as well as ② and ⑩ to set up the structures needed for a join.

Thread-Local Execution. During the execution phase, the actual query processing is performed. This can include, e.g., scanning tuples from disk or buffers, applying filters and functions, aggregation, and joins. Umbra employs morsel-driven parallelism [29] and, therefore, this step is run in parallel, processing a single morsel for each invocation. The results are then stored in thread-local buffers, or directly reported for the top-most pipeline. Steps ③, ⑦, ⑪, and ⑮, e.g., store the corresponding inputs in a local hash table and either aggregate them or keep them unmodified to be later probed by the join. Step ⑲ probes the join hash tables in parallel and reports the resulting join tuples directly, without using a local buffer.

Merge. After thread-local processing is completed, the results are combined and stored in the global state to be accessible for the next pipeline. Possible examples for this step are merging the individual runs of the merge-sort for a sorting operator, or combining local buffers and hash tables. In our example, this happens in steps ④, ⑧, ⑫, and ⑯. Since pipeline 5 directly reports the final result, it is not necessary to merge local results.

Cleanup. Finally, the auxiliary data structures are cleaned up, and the memory used is freed (⑤, ⑨, ⑬, and ⑰).

3. APPROACH

Traditional approaches to materialized view maintenance fully process tuples. In deferred maintenance, all tuples are either buffered in memory or stored on disk. When the view result is requested, the database will reevaluate the query from scratch, requiring it to scan the materialized relation. All tuples, therefore, need to be materialized and kept accessible at a moment's notice. This is problematic for unbounded high-velocity data streams, which are often desired to be processed in an exactly-once non-materializing fashion.

For our system, a full materialization of the stream would either mean writing it to disk, or keeping it in memory. The first option would increase the disk IO tremendously, affecting other disk operations for other queries. Keeping the tuples in memory, on the other hand, reduces the memory available for query processing. Both could dampen the overall system performance, which means that we therefore have to rely on the inserter to handle the tuple.

Other approaches processing new tuples at insert time, like eager and incremental view maintenance, also avoid the high storage cost of stream data. However, they propagate the full change immediately. This is not trivial at the high frequency required for stream processing, and most systems require hand-written queries to handle the updates. Even modern high-velocity approaches tackling this problem without manual user input, like DBToaster, require specialized operators with support of deletes and updates at any part of the query. This makes it hard to integrate this approach into an existing database system efficiently. For example, DBToaster only exists as a stand-alone solution or as an external library, not fully integrated into a database.

3.1 Split Maintenance Strategy

Our approach can be seen as a combination of eager and deferred view maintenance, providing the best of both worlds. To keep the introduced overhead low, we propose processing inserts only as far as needed. In general, this means we want to process the input until the point where it has to be materialized for the first time, that is, the first pipeline breaker. This allows us to perform initial processing steps at insert time while reducing the memory consumption compared to deferred maintenance. Using pipeline breakers as a natural storage point also allows for easy integration. Tuples are never invalidated at pipeline breakers, e.g., join build sides. After materialization in pipeline breakers, the remainder of the query is oblivious to the nature of the input. Therefore, we do not require specialized operators that support removing or updating tuples at any given time, as is needed for incremental view maintenance. Further, in contrast to deferred view maintenance, we never need to materialize the full stream inputs. This greatly reduces the storage cost of stream processing. In the *used* CTE of Figure 2 in pipeline 2 of Figure 4, e.g., we would insert the tuple into the hash table of the grouping operator and update the aggregate accordingly. While in this simple query the overhead of updating the result tuple is negligible, this is not the case for more complex queries. Consider, e.g., the query plan for the full query displayed in Figure 4. The query still is rather simple; however, fully processing new tuples in this query is not. Inserts into pipeline 2 would trigger a recalculation of the join in pipeline 5. Since we do not inherently support updates in and deletes from join build sides in our system, the other event stream would have to be fully reprocessed. This basically requires us to run steps $\circled{7}$ to $\circled{19}$ for every insert if we process all tuples fully. Once the tuple is materialized within the first pipeline breaker, finalizing query processing when the query result is required is possible by running the remaining steps for the other pipelines. We will use the query in Figure 4 as a running example throughout this paper, assuming *stations* to be a regular database table.

This approach benefits from Umbra's novel state machine approach to query processing wherein we can stop and delay processing after any of the steps enumerated in Figure 4, and even run multiple input pipelines independent from one another. We use the remainder of this section to outline the supported queries, the different phases of the maintenance, and the integration into regular database query processing.

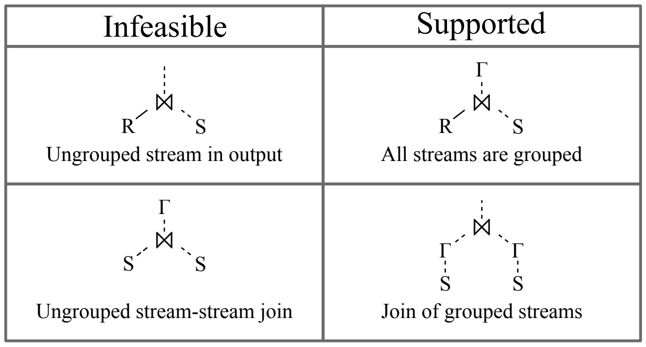


Figure 5: Supported and infeasible queries for continuous views. Streams are marked *S* and regular relations *R*. Pipelines containing streams are dashed. Subtrees containing no streams are not restricted.

3.2 Supported Queries

Similar to SPEs, our system restricts the queries it supports to protect the user from undesired side effects, such as state explosion and ambiguous query results. In the following section, we describe the rules for queries in Umbra. Moreover, we offer an overview of the supported queries in other systems.

3.2.1 Umbra

For Umbra, Figure 5 visualizes and summarizes the most important rules that we motivate in the following section.

Input Pipelines. We require the first pipeline breaker of every stream input to be a grouping operator, as shown in the first row of Figure 5. This reduces the risk of state explosion compared to allowing arbitrary pipeline breakers. Most pipeline breakers, like sorting or join build-sides, would still materialize at least parts of the stream leading to memory shortages that would impact the performance of the system. As the grouping itself is mostly oblivious about its input, it is still possible to query the entire stream by grouping for a key if this is desired by the user. This would, however, lead to the entire stream being materialized.

Transactions. For us, streams are inherently not transactional. Keeping streams in a transaction would mean that once an analytical query is started within another transaction, newly arriving stream tuples would not qualify for this query. The user, however, will always expect to see the current state of the stream, independently of the current transaction. We still want to isolate the continuous view from changes to regular tables. This isolation is necessary to guarantee that previously reported join results are not invalidated by later arriving tuples. Consider, e.g., a join as in the first row of Figure 5. Removing a row from the relation on the left-hand side would mean that results reported for that tuple as join partner are no longer valid. This would lead to ambiguous and even inconclusive results for the user, which we want to prevent. To achieve this isolation without requiring transaction handling, a continuous view reads all committed changes at creation time and uses exactly this state of the database throughout its lifetime. This isolation furthermore allows us to run pipelines not involving streams only once and keeping the results cached. In essence, this means views read the committed state of all relations once at creation time [39].

Table 1: Comparison of stream processing approaches. Dashes indicate the system's documentation either directly states a feature is not supported, or it does not contain enough information to indicate support.

		Stream Processing Engines					In-Database Stream Processing		
Aspect		**Flink** [8]	**Spark Structured** [50]	**Storm Trident** [46]	**Trill** [10]	**Saber** [25]	**PipelineDB** [42]	**DBToaster** [24]	**Umbra**
Version		1.10	3.0.0	2.2.0	2019.9.25.1	7be036c	1.0.0	2.3	-
Stream Deletes/Updates		-	-	-	-	-	-	Yes	-
Early Results		Limited	Limited	Windowed	Yes	Windowed	Yes	Yes	Yes
Historic Data		External	External read-only	External read-only	External read-only	-	Internal w/ Postgres	External read-only	Internal
Scale-Up		Yes	Yes	-	Yes	Yes	Limited	-	Yes
Scale-Out		Yes	Yes	Yes	-	-	-	-	-
Aggregates		Extensive	Basic	Extensive	Extensive	Basic	Extensive	Basic	Extensive
Windowing		Built-in support	Built-in support	Built-in support	Built-in support	Built-in support	Built-in support	Using GROUP BY	Using GROUP BY
Stream-Joins	Equi	Yes	Yes[*]	Batchwise	Yes	Windowed	Yes[‡]	Yes	Yes[‡§]
	Theta	-	Yes[*]	-	-	Windowed	Yes[‡]	Yes	Yes[‡§]
	Outer	Equi	Windowed[*†]	Batchwise[†]	Yes[†]	-	Yes[†‡]	-	Yes[†‡§]
	Semi	Yes	-	-	-	-	Yes[†‡]	Yes	Yes[†‡§]
	Anti	Limited	-	-	Yes[†]	-	Limited[†‡]	Yes	Yes[†‡§]

[*]Only unaggregated streams [†]Single direction [‡]No stream-stream joins [§]Restricted only for ungrouped streams

Joins. Streams are often problematic for joins, and most SPEs, therefore, restrict joins in some way. Joins of unbounded streams without windowing can lead to a state explosion as both sides would need to be fully materialized to ensure join correctness. While there are viable solutions for this problem in many SPEs, in the form of windowed joins, we want to allow queries spanning entire streams. We plan to enable windowed processing based on algorithms for persisted data [30] in a future version. Since even non-blocking joins like the XJoin [47] are not designed to handle unbounded inputs, we ensure that streams are on the probe side if only one input depends on a stream. This way, we never have to materialize the stream and can simply probe the pre-calculated build side. For stream-stream joins we utilize the restrictions mentioned above, forcing streams to be grouped. We further extend this restriction and require both inputs of a stream-stream join to be grouped as seen on the bottom of Figure 5. When joining only previously grouped streams, we can ensure that we do not report join results that would later be invalidated by new tuples arriving on either side, again preventing inconclusive results in the view. We do not restrict joins between regular tables.

3.2.2 Other Approaches

To better illustrate how the rules motivated above compare to existing systems, Table 1 provides an overview of natively supported features of similar stream processing approaches. We group the approaches into specialized SPEs and stream processing in the context of databases. The table is based on features described in the documentation of the most recent release. While some of the described systems have additional restrictions and features, we believe the table provides a good overview of those most important.

One can see that Umbra offers an extensive functionality, especially for joins, second only to DBToaster. Further, there is a notable difference in the focus of the systems. While the SPEs focus on windowed queries with limited join options, the in-database approaches offer only basic or manual windowing. However, they support more complex queries and even manage historic data internally. Umbra's restriction of joins only applies to ungrouped streams and can be bypassed, e.g., by grouping for a key. On the downside, this will likely lead to a performance decrease. Information in streams has to be condensed for analysis, and grouping is a common way to achieve that. Therefore, we argue that requiring the grouping of streams does not gravely limit the applicability of our approach. Queries without grouping or aggregation, i.e., map-like or filtering queries popular for SPEs, are also possible within Umbra. As these queries seldom materialize their result, and early results in the form of materialized views are therefore not of interest, we consider them to be outside the scope of this paper.

3.3 Query Planning

Like all other tables and views, continuous views are created using the regular SQL interface. This means we have to enforce the aforementioned restrictions in the semantic analysis of the statement and reshape the query plan according to our needs. As a first step, we translate the query representing our continuous view into an unoptimized logical query plan. Performing a depth-first traversal of the query plan, we remember for each operator whether its input contains a stream, and if so, whether there is a grouping in between. Given this mapping we modify the query plan in a second traversal, taking steps for three operator types:
Group By. We again keep track of groupings, but this time we remember if there is a grouping higher up in the tree. Each time we encounter a stream input we ensure that its parents contain a grouping, thus verifying we do not store ungrouped streams.
Order By with Limit. Contrary to stream inputs, we ensure that the parents of sorting operators do not include a grouping. As we require stream inputs to be grouped somewhere, we know that there is a grouping below. By doing so, we avoid sorting ungrouped streams. Sorting operators without a `LIMIT` clause are simply dropped per the SQL standard. As Umbra, like many other systems, does not guarantee tuples to be processed in scan order, scanning the materialized result at query time will anyway lose the order.

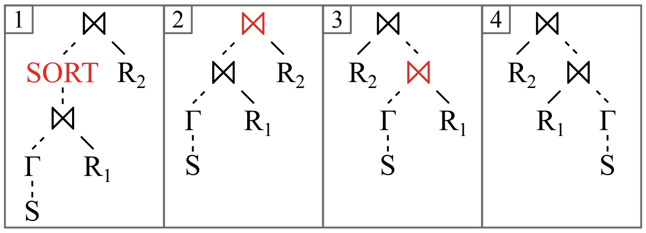


Figure 6: Modifications performed on a query plan prior to compilation. Streams are marked *S* and regular relations *R*. Pipelines containing streams are dashed. Red color marks the currently modified operator. Removing unused sort (1), moving streams to the probe side of joins (2 and 3) top down, and finished plan (4).

Therefore, if a sorted result is desired, it has to be requested when querying the materialized view.

Join. Joins require the most intrusive modification of the query plan. These depend on the order and stream containment of the join inputs. We say an input pipeline *contains* a stream if there is a stream input somewhere upstream of the pipeline. If no input contains a stream, we leave the join unmodified. When both inputs have a stream, we ensure that both streams have a grouping operator between the join and the stream input using the pre-calculated containment map. In cases where only one input contains a stream, we try to modify the query plan to have the stream on the probe side of the join, independent of grouping. This allows us to later pre-calculate the build side and rerun the join for changed input with little overhead. While this is easy for simple uncorrelated joins independent of the join type, switching the input is not always possible. For correlated joins, e.g., only some cases can be unnested in a way that supports switching the inputs. When switching is not possible, we again simply reject the view.

As the query of our running example is too simple to show all these modifications, we show them in a more complex plan in Figure 6. As a first step, the order by without a limit is removed ($\boxed{1}$). Afterward, we modify both joins to have the streams on the probe side ($\boxed{2}$ and $\boxed{3}$). Note that in $\boxed{3}$, the join is modified even though there is no direct stream input, only one upstream through the other join. After enforcing the aforementioned restrictions, we optimize the query plan using Umbra's optimizer, ensuring the restrictions are not violated by optimizations.

3.4 Code Generation

After adapting the logical query plan for a continuous view, we look at the code generation for the identified phases of maintenance. As outlined in Figure 1, we distinguish three different tasks: The view initialization, handling stream inserts, and query evaluation. Each of these tasks consists of individual steps of pipelines, like those described in Figure 4. In general, each pipeline of a query plan is mapped to exactly one of these tasks. In the following sections, we will refer to the pipelines as follows:

Static Pipeline: A pipeline that has no stream input, either directly or through one of its child pipelines (pipeline 1 in our example). These pipelines' steps are handled by the view initialization.

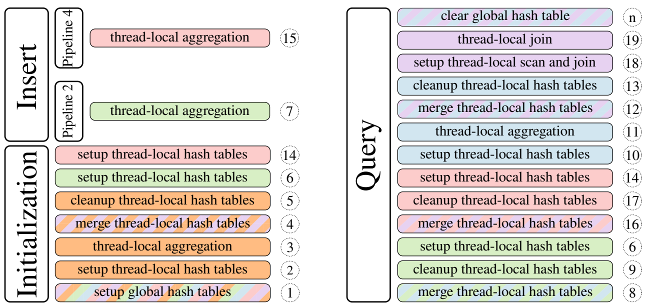

Insert
Pipeline 4
thread-local aggregation 15
Pipeline 2
thread-local aggregation 7
Initialization
setup thread-local hash tables 14
setup thread-local hash tables 6
cleanup thread-local hash tables 5
merge thread-local hash tables 4
thread-local aggregation 3
setup thread-local hash tables 2
setup global hash tables 1
Query
clear global hash table n
thread-local join 19
setup thread-local scan and join 18
cleanup thread-local hash tables 13
merge thread-local hash tables 12
thread-local aggregation 11
setup thread-local hash tables 10
setup thread-local hash tables 14
cleanup thread-local hash tables 17
merge thread-local hash tables 16
setup thread-local hash tables 6
cleanup thread-local hash tables 9
merge thread-local hash tables 8

Figure 7: Generated callbacks for initialization, insert handling, and query evaluation for the running example. Colors, steps and step numbers are consistent with the query plan shown in Figure 4. *n* denotes a new step required for rebind.

Stream Pipeline: A pipeline that directly processes a stream input (pipelines 2 and 4).

Upper Pipeline: A pipeline that has an indirect stream input, used for query evaluation (pipelines 3 and 5).

To better illustrate the generated callbacks discussed within this section, we summarize the steps involved in Figure 7.

3.4.1 Initialization

After having established the fundamental logical components of each pipeline, we describe which steps have to be run at view initialization, i.e., at view creation time. Based on a query plan modified according to the rules described in Section 3.3, we can prepare the view. The initialization consists of 2 main tasks: **(1)** processing static pipelines, and **(2)** preparing all stream pipelines for inserts. Below, we describe these steps in theory and using our running example.

(1) As discussed before, we do not want changes to the view's table inputs to be propagated to the view. Therefore, we can calculate the results for all pipelines affected only by non-stream inputs exactly once. By doing this at view creation time, we immediately gain independence from the underlying database tables as we cache the results in memory. As a first step, we generate the code for the global setup 1. In the following step, we handle all steps of static pipelines. One can easily see that static pipelines always form subtrees in the views query plan, with the root being the operator where the pipeline intersects with a stream or upper pipeline (pipeline 1). Each of these subtrees is handled as it would be in regular query processing, in a left-to-right bottom-up fashion. For our example, this means generating code for steps 2 to 5.

(2) In addition to executing static pipelines, we also initialize all stream pipelines. Delaying pipeline initialization to the insert callbacks would drastically increase the insert time for a few tuples. For regular queries, Umbra ensures that there is exactly one worker thread accessing a local state instance. Currently, we cannot keep this strict worker-to-state mapping for performance reasons, and therefore have to be aware of possible race conditions. Consider, for example, a second insert callback arriving at a pipeline that is currently being initialized. At this point, an insert might be performed for a not fully allocated hash table, resulting in a lost tuple, or worse, a corrupt memory access. In our example we have two stream pipelines, pipelines 2 and 4, and therefore generate code for steps 6 and 14.

3.4.2 Insert Handling

Handling inserts is the most performance-critical part of the view's maintenance, as this part is performed for every single tuple upon arrival. Therefore, we want this step to execute as little code as possible by only running the thread-local execution. We modify the local execution code slightly to process single tuples instead of morsels. This way, we can later call this step in a callback fashion for each inserted tuple. However, as inserts in Umbra are processed morsel-wise, this will still lead to morsel-wise processing of updates for larger inserts. The callback will then be triggered for each tuple within the morsel. These callbacks are generated for each stream pipeline independently. As we have two stream inputs for our example in Figure 4, we also generate two callbacks consisting of steps ⑦ and ⑮ respectively.

3.4.3 Query Evaluation

Finally, we need to generate the code to combine the cached static pipelines and the dynamic stream pipelines to obtain the query result. This step is again composed of two main parts: **(1)** processing all upper pipelines, and **(2)** resetting the view's internal state to be ready for the next query.
(1) As a first step, we want to execute the merge phase of all stream pipelines. This makes all locally held results available for the upper pipelines using the global state. Additionally, we reset the stream pipelines by running the local cleanup and initialization. This way we make the local state available for new inserts as fast as possible. For our running example this translates to steps ⑧, ⑨, and ⑥, as well as ⑯, ⑰, and ⑭, in that order. After these steps have been completed, we can run all upper pipelines as if they were a separate query. In our running example, this includes pipelines 3 and 5, and therefore, steps ⑩ to ⑬, as well as ⑱ and ⑲. Finally, we generate code to store the view's query result, similar to a materialized view. For one thing, this allows us to compare different versions of the view with one another. Furthermore, slightly outdated versions of the view can then be read without having to run the query evaluation if requested.
(2) After materializing the view's current result, we have to reset the internal state. We cannot, however, clean up and reinitialize the global state as we did for the local state. The global state still holds all cached results for static and stream pipelines, which would be lost at a full reset. For our example, we would want to avoid rebuilding the build side of the join between pipelines 1 and 5. Most operators already provide some functionality for such a reset in the form of an operator rebind, normally used for recursive queries. We can safely rebind all operators that exclusively belong to upper pipelines, i.e., all except those that are top-level operators of stream pipelines and static pipeline trees. Further, we reset all join operators that have a stream pipeline on the build side, e.g., the join between pipelines 3 and 5 (ⓝ in Figure 7).

Resetting states is necessary to prevent tuples from being processed multiple times for the same query: In between queries, all tuples are held in local hash-tables, such as the grouping of the *used* CTE of our example. If we do not clear the local hash table after a query, the tuples would be included in both the local and global state at the same time. The next query would then again merge the local state into the global state and, thereby, include the tuples twice.

Algorithm 1: Insert handling

```
   input : ContinuousView v, Queue q, Mutex m, Tuple t
 1 if m.lock() successful then
 2     if t2 ← q.dequeue() successful then
           /* execute matching insert callback of Figure 7 */
 3         processTuple(t2)
 4     end
 5     processTuple(t)
 6     v.hasNewInput ← true
 7     m.unlock()
 8 else
 9     q.enqueue(t)
10 end
```

3.5 Runtime Integration & Optimizations

Now that we have the required code fragments to handle continuous view maintenance, we have to integrate them into our database runtime. The initialization is executed once at view creation time. Next, the view is prepared to handle inserts through the aforementioned callbacks. We register the callbacks with each stream input. From there on out, each insert into the stream triggers the callback and handles the local processing of the tuple. Queries to the view work in a similar fashion: Each query triggers the materialization callback and writes the results to a temporary table. Materializing the results is necessary to have a consistent state of the view within a query, e.g., for self-joins. This materialized result is scanned for the actual query processing of the database as if it were a regular table.

The described integration is limited in two ways: First, local processing is not thread-safe as multithreaded operations on the same instance of the local state were not intended for Umbra. Without controlled access to the local processing, i.e., through a lock, we could experience race conditions for some operators. Second, query processing resets the local state of all stream pipelines. Hence, querying requires an exclusive lock to prevent inserts during query processing. This lock, however, can be released as soon as the local states have been reset. Preliminary experiments showed that acquiring the lock is often costlier than the insert itself, and delays to inserts are mainly introduced by queries. To reduce these delays, we introduce a lock-free queue in front of the local state. Newly arriving tuples are buffered in this queue when either a query or another insert is blocking the local state.
Insert Handling. Algorithm 1 describes the modified insert handling: We again first try to obtain the lock for the view. If another thread holds the lock, we enqueue the tuple in the buffer queue and retry for the next tuple to be inserted. As there can be arbitrarily large gaps between queries, the queue could grow indefinitely when we empty it only at query-time. Therefore, we need to empty the queue between queries. As we do not want to integrate a dedicated background worker within our system, which would introduce scheduling overhead, the queue is emptied by inserts. Once an insert has acquired the lock, it additionally dequeues tuples from the queue and processes them. While dequeuing a single tuple proved sufficient in our experiments, dequeuing multiple tuples or the whole queue is possible as well.
Query Evaluation. We consider all previously inserted tuples for queries. Therefore, we empty the queue whenever a query arrives. To prevent race conditions from inserts,

we redirect all tuples to the queue until the local states of stream pipelines have been reset. If no tuples have arrived since the last materialization, the processing is skipped, and the last materialized result is used instead.

3.6 SQL Interface

After having described the inner workings of our continuous views, we briefly show how they are created using the SQL interface of our database system. To minimize the necessary changes in the SQL dialect all required operations can be performed using regular statements.

Creating Streams. As a first step, the user needs to create the streams that will be evaluated in a continuous view. Streams can be created as *foreign tables*:

`CREATE FOREIGN TABLE` **stream_name** `(...)`
`SERVER` **stream**

We require that no server named *stream* can be created.

Creating Continuous Views. After the required streams have been created, users can create continuous views using simple `CREATE VIEW` statements:

`CREATE VIEW` **continuous_view_name** `AS` **query**

During the semantic analysis of the query, we check whether any of the queried relations is a stream. If so, we automatically create a continuous view, else we create a regular view. All modifications and checks described in Sections 3.2 and 3.3 are only performed for continuous views.

Inserting Data. Inserts can be expressed as regular `INSERT` statements, and streams can be copied from CSV using `COPY`:

`INSERT INTO` **stream_name** {`VALUES (...)|` **query**}

`COPY` **stream_name** `FROM` **'filename'** `CSV`

query can be an arbitrary SQL query that does not contain the stream itself. It is possible to attach new continuous views and queries to streams that are currently fed with data, but those only see tuples arriving after their creation.

3.7 Updates

Up until this point, we only discussed inserts into streams, not changes to the static tables involved. Both updates and deletes to tables exclusively used in upper pipelines, like *stations*, can be realized at any time. To incorporate changes to these tables, we simply reevaluate all static pipelines affected. For our running example, this means rerunning all steps in the initialization callback that correspond to pipeline 1 (2 to 5) and replacing the build side of the top-most join with the new stations. The next time the view is queried, *all* stream tuples are then evaluated using the changed tables, guaranteeing consistent results for every materialized state.

Changes to static tables and subtrees joined directly with stream pipelines (e.g., top right of Figure 5) are not supported. In order to support changes to such tables, we would have to either (a) keep all stream tuples materialized to reevaluate the join correctly, or (b) join stream tuples with the state of tables at arrival time. Case (a) would lead to extensive memory consumption and high refresh times, which is exactly what we aim to avoid with our continuous views. While case (b) is used in some systems, like PipelineDB, we refrain from using this approach. The results of the view query would otherwise be dependent on the processing order of stream events and table updates, leading to inconclusive and inconsistent results. Like many other SPEs (c.f. Table 1), we consider streams to be append-only. Therefore, we do not support deletes or updates of stream tuples.

3.8 Fault Tolerance

While it is not the focus of our paper, we want to briefly address fault tolerance. For continuous views, we can utilize the integration into Umbra, and the fact that we use the SQL interface to interact with the database. This way, we have access to Umbra's logging mechanism and can replay not fully processed tuples in case of an error. However, replaying the full stream in case of an error can lead to a considerable delay during recovery. To reduce the number of tuples to be replayed, we can combine this with checkpoints which are common in SPEs for recovery, e.g., in Flink. Here, we can utilize our custom memory management and the separation of global and local state. Global state is only modified during materialization and, therefore, captures a consistent state of all operators between queries. We can take regular snapshots of the global state as checkpoints and restore the last snapshot in case of a failure. This way, only tuples that arrived since the last materialization have to be replayed.

Another aspect we want to mention is how our strategy can deal with a high load. Many SPEs utilize publish-subscribe-like asynchronous interfaces to accept data, allowing them to delay processing in case of a high load without influencing the inserter. The interface of our continuous views, on the other hand, is SQL-based and, therefore, synchronous. We offer some load balancing in the form of the lock-free queue described in Section 3.5. The queue can help overcome short spikes, but, for a prolonged high load, it can grow indefinitely. This can lead to a decrease in the overall system performance. As we do not consider load shedding, which would mean losing information, Umbra should not be used in a completely standalone fashion when an extremely high load is expected for a long time. Instead, we envision it integrated into an ETL scenario with an external data collection engine, as described by Meehan et al. [32].

3.9 Portability

While our described implementation of continuous views is optimized for Umbra's execution engine, our approach is in no way limited to Umbra. Continuous views, as described above, can be realized in many database systems using stored procedures, auxiliary tables, and materialized views, albeit less optimized than our fully integrated approach. To demonstrate the feasibility of such an integration, we implemented continuous views in PostgreSQL using its procedural languages.[1] Our implementation parses view queries and generates SQL statements for keeping continuous views up-to-date. When inserting new tuples, designated insert functions update the aggregates instead of materializing the stream.

As in Umbra, we distinguish between static, stream, and upper pipelines. Static pipelines are evaluated and cached during initialization. For stream pipelines, we generate insert functions that incrementally update the previous results in an auxiliary table. A single view combines all upper pipelines and makes the query result available to users on refresh. In contrast to eager maintenance, the update logic for our continuous views can be generated automatically.

[1]Available at: `https://github.com/tum-db/pg-cv`

Table 2: Continuous view AIM queries, slightly simplified for readability. $\alpha \in$ [0, 2], $\beta \in$ [2, 5], $\gamma \in$ [2, 10], $\delta \in$ [20, 150], $t \in$ SubscriptionTypes, $cat \in$ Categories, $v \in$ CellValueTypes

	In evaluation: $\alpha = 2$, $\beta = 2$, $\gamma = 4$, $\delta = 25$, $t = 2$, $cat = 1$, $v = 2$

Q1	SELECT avg(t_duration) FROM (
	SELECT sum(duration) AS t_duration FROM events
	WHERE week = current_week() GROUP BY entity_id
	HAVING count(local_calls) > α)

Q2	SELECT max(max_cost) FROM (
	SELECT max(cost) AS max_cost FROM events
	WHERE week = current_week() GROUP BY entity_id
	HAVING count(*) > β)

Q3	SELECT sum(t_cost) / sum(t_duration) AS cost_ratio
	FROM (
	SELECT sum(cost) AS t_cost, sum(duration) AS t_duration,
	count(*) AS num_calls FROM events
	WHERE week = current_week() GROUP BY entity_id
	) GROUP BY num_calls LIMIT 100

Q4	SELECT city_zip, avg(num_calls), sum(duration_calls)
	FROM (
	SELECT entity_id, count(*) AS num_calls,
	sum(duration) AS duration_calls FROM events
	WHERE NOT long_distance AND week = current_week()
	GROUP BY entity_id
	HAVING count(*) > γ AND sum(duration) > δ
	) e, customers c WHERE e.entity_id = c.id GROUP BY city_zip

Q5	SELECT c.region_id, sum(long_distance_cost)
	AS cost_long, sum(local_cost) AS cost_local
	FROM events e, customers c
	WHERE e.entity_id = c.id AND week = current_week() AND
	c.type = t AND c.category = cat GROUP BY c.region_id

Q7	SELECT sum(cost) / sum(duration)
	FROM events e, customers c WHERE e.entity_id = c.id AND
	week = current_week() AND c.value_type = v

4. EVALUATION

First, we evaluate our approach for multiple parallel streaming queries using the AIM benchmark, comparing it to an SPE and to an in-database solution. To show real-world performance, we use full query round-trip times on the AIM benchmark. To highlight the raw analytical performance, we further evaluate it against competitors on a TPC-H workload modified for streaming in Umbra and in PostgreSQL on isolated queries. Subsequently, we evaluate the load balancing capabilities using a microbenchmark. We will refer to continuous views within Umbra as *Umbra* throughout this section. All systems, approaches, and experiments in this section are run on a machine equipped with an Intel Xeon E5-2660 v2 CPU (2.20 GHz) and 256 GB DDR3 RAM.

4.1 AIM Benchmark

The AIM telecommunication workload, as described in [7], has been previously used to evaluate the performance of modern database systems against SPEs and specialized solutions [23]. We want to extend this evaluation with the comparison of stream processing in databases, in the form of our continuous views, with SPEs. As a second approach to stream processing using continuous views, we choose the open-source PostgreSQL extension PipelineDB [42].

In the AIM Benchmark, calls for a number of customers are tracked as marketing information. On these calls, analytical queries are run to make offers or suggest different plans to certain customers. In contrast to [23], we do not want to store all possible aggregates for ad-hoc queries and instead only aggregate what is necessary for the AIM queries specified in [7]. Table 2 shows the AIM queries modified for use as a continuous view. As we cannot express the original query 6 as a single view we omitted it from all experiments. Furthermore, as neither PipelineDB nor our continuous views support changing view queries at runtime, we select one value by random for each query parameter ($\alpha, \beta, \dots$) from the range specified in Table 2.

4.1.1 Configuration

Both the client and the server run on the same machine and, to recreate the setup of [23], the client generates events using one thread and queries using another thread. For the experiments, we vary the number of threads of the server. When speaking of *number of threads* in the remainder of this section, we always mean the number of server threads. The number of client threads remains untouched. The database is initialized with 10M customers unless stated otherwise. We implement the approaches as follows:

Flink. To represent classical SPEs, we use Apache Flink [8]. As the internal state of Flink can only be accessed by point lookups, and to recreate the setup of [23] as closely as possible, we implement a custom operator for the workload. Our custom operator keeps track of the required aggregates per customer (e.g., number of local calls this week), and queries are run exclusively on these aggregates. While Flink can handle all individual queries on its own, we use the custom operator to process all queries at once. The operator further allows us to share aggregates between queries. All experiments are performed on Flink 1.9.2.

PipelineDB. We use the latest available version of PipelineDB (short *PDB*), 1.0.0, and integrate it into PostgreSQL 11.1, both in default configuration. We slightly adapt the queries in Table 2 to fit the syntax of PipelineDB. PipelineDB requires at least 18 worker threads, thus we cannot limit the total number of threads. Instead, we vary the number of threads per query (*max_parallel_workers_per_gather*). In our experiments PipelineDB inserts timed out occasionally, but this did not limit the reported query throughput. However, because of this, we consider the numbers reported for PipelineDB to be an upper bound.

Umbra. For Umbra we create the continuous views as specified in Table 2. The number of threads maps to Umbra's worker threads. In addition, we run a single thread that handles all connections to the database and is not involved in query processing.

Unless stated otherwise, all experiments for both Umbra and PipelineDB are measured using the pqxx library to connect to the systems. Events are generated inside the database. Throughput averages are calculated over three minutes of execution based on full query round-trip times.

4.1.2 Experiments

Concurrent Access. First, we look at the overall performance of the systems under concurrent write and read accesses with an increasing number of threads. Each query in Table 2 is executed with equal probability, and inserts are performed at 10K and 50K events per second respectively. We report the results in Figure 8. Flink scales nicely with an increasing number of threads but keeps behind both PipelineDB and Umbra. For PipelineDB we expected to see little scale-up as most of the maintenance is handled by background workers, which we could not limit to the given thread counts. However, the throughput is still very unstable, most likely attributable to the aforementioned problems

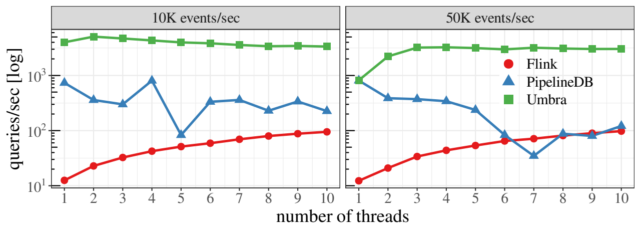


Figure 8: Concurrent throughput of queries and inserts with increasing number of threads.

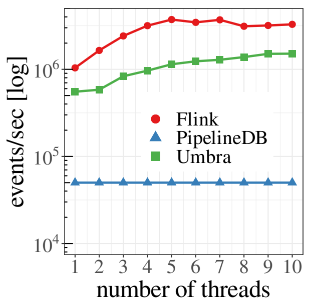


Figure 9: Isolated insert throughput with increasing number of threads.

Table 3: Average query throughput in queries per second without concurrent writes

	SF	Umbra	PDB	Flink
Q1	1	**10748**	6725	40
Q2	1	**10761**	7274	46
Q3	1	**10677**	6123	19
Q4	1	**10710**	125	17
Q5	1	**10501**	6987	13
Q7	1	**10709**	7957	26
Q1	10	**11161**	208	4
Q2	10	**11171**	991	4
Q3	10	**11061**	100	2
Q4	10	**11074**	124	2
Q5	10	**10835**	10619	1
Q7	10	**11169**	9582	2

with inserts. For 50K inserts per second PipelineDB performance degrades with increasing thread count. This is likely caused by interference between insert and query threads.

Since Umbra can handle most of the queries' workload single-threaded, we do not notice a scale-up beyond three threads. Despite not utilizing all threads, we are still able to outperform the competitors consistently, in parts by over an order of magnitude. For higher insert rates, initial scaling is better, since the inserts take more time, and thus parallel execution of queries has a larger impact.

Insert Throughput. As the goal of our continuous views is to handle insert-heavy workloads, we further investigate the isolated insert throughput without concurrent queries. While we do not issue queries, we still create the corresponding views for Umbra and PipelineDB and track all required aggregates for Flink. For both Flink and Umbra, we report the throughput under full load. For PipelineDB, we report the highest measured throughput. The results are shown in Figure 9, again for an increasing thread count. PipelineDB's throughput peaks at around 50K events per second consistently, which seems to be the limit of the non-scaling background workers. Both Flink and Umbra scale well with an increasing thread count for up to 5 threads. Flink does not scale beyond 5 threads. This can be expected as Flink is optimized for scale-out, not for multiple concurrent queries on a single node. However, Flink still offers the highest insert rates. Umbra scales better for higher thread counts and achieves insert rates of more than 1.5M events per second.

Query Throughput. For the final AIM experiment, we look at the isolated read throughput for individual queries. To report actual query results, we initialize the aggregates with random values for Flink and 10K random events for both PipelineDB and Umbra. We report the average query throughput for all individual queries, exemplary for two threads, in Table 3. As the reported query throughput is for full query round-trips for both PipelineDB and Umbra, we only see a slight difference for most queries. The majority of the time is spent in parsing and sending the query, not in execution. Still, Umbra is able to outperform PipelineDB consistently throughout all queries. As Flink does not materialize full query results, it needs to calculate parts of the query on the fly, thus staying behind for all queries. When increasing the number of customers to 100M (SF=10), we can see the advantages of result caching for Umbra and PipelineDB. While Flink degrades linearly, PipelineDB's throughput stays constant for queries not grouping by customer, and Umbra's throughput stays constant for all queries.

4.2 TPC-H Benchmark

After evaluating our strategy on a concurrent streaming benchmark, we compare it to modern view maintenance using DBToaster [24], and the scale-up SPE Trill [10], on a more analytical workload based on TPC-H. We configure all systems to treat *lineitem* as a stream and all other relations to be static. For our experiments, we choose queries 1, 3, and 6, where most of the work of our approach happens at insert time. For these queries, all systems should act similarly. Further, we choose query 15 as a grouped stream-stream join and query 20 to represent queries where the majority of analytical evaluation is performed on a grouped stream. As neither Trill nor DBToaster support order by or limit in their streaming API, we remove all order by and limit clauses from the queries. We rewrite queries for our competitors to fit their syntax and feature set where necessary.

4.2.1 *Configuration*

We configure and implement our competitors as follows:
DBToaster. We use DBToaster v2.3[2] and allow it to handle streams as append-only using the `IGNORE-DELETES` flag. Further, we distinguish two versions for our experiments: The query-optimized DBT^Q performing full refreshes, and the insert-optimized DBT^I. DBT^I is allowed to perform only partial refreshes at insert time (`EXPRESSIVE-TLQS` flag).
Trill. We use Trill v2019.9.25.1[3] and implement all queries using the LINQ-style interface. We only operate on cached inputs for both streams and tables and pre-calculate all joins and subtrees not involving lineitem. Both tables and streams are configured to have an infinite lifetime to keep them in the same join-window. We optimized join order and execution hints to the best of our knowledge.
Flink. We implement the queries in Flink 1.9.2 using the BatchTables API and the SQL interface.
Umbra^D. We implement deferred maintenance in Umbra based on full query evaluation at refresh.

As the semantics of concurrent queries is quite different for all approaches, we focus on insert throughput of streams and view refresh times. For all experiments, we report the average of 10 runs after performing 3 warm-up runs.

4.2.2 *Experiments*

Insert Throughput. We compare the insert throughput of all systems when inserting the *lineitem* table once. As the DBToaster release currently only offers a single-threaded execution, we measure all systems using one thread. To obtain

[2]`https://dbtoaster.github.io/download.html`
[3]`https://www.nuget.org/packages/Trill/`

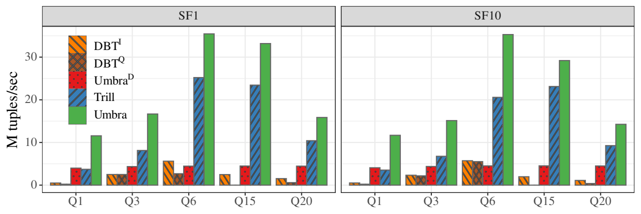
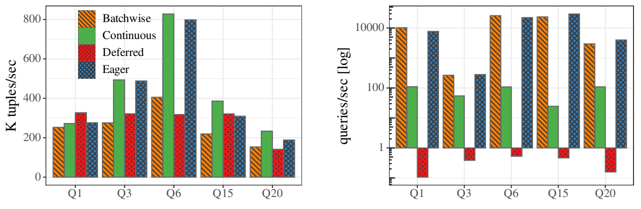
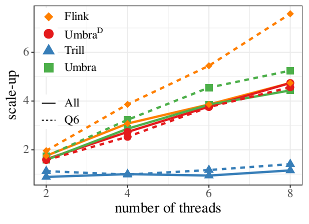


Figure 10: Insert throughput against TPC-H baselines for scale factors 1 and 10.



(a) Insert **(b) Query**

Figure 11: Comparison of continuous views with traditional views in PostgreSQL on TPC-H. Queries consist of refreshing and fetching the result.



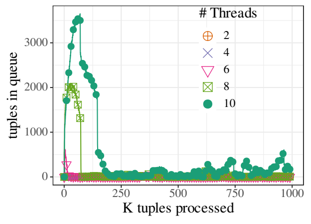


Figure 12: Scale-up of Trill, Flink, and Umbra views relative to single-threaded performance.

Figure 13: Queue length of a single continuous view with varying number of threads.

comparable results, we hold data in a cached stream buffer for Trill. For both DBToaster and Umbra, we only measure processing time without input parsing. We cannot easily recreate this processing-time-only setup in Flink, therefore we exclude it from this experiment. To even the scores, we add the cost of a single view refresh to all systems that do not provide the result immediately. As the runtime for query 15 in DBT^Q exceeded reasonable limits for the full table, we report throughput based on a 1000-element sample. The throughput, displayed in Figure 10, is largely independent of the scale factor with minor fluctuation, e.g., at Q6. Overall, Umbra and Trill offer the highest throughput, with a slight advantage for Umbra. Both stay ahead of both DBT variants independent of query and scale factor. Umbra^D stays ahead of DBToaster for most queries and can even outperform Trill for Q1. For the simple queries Q1 to Q6, both DBT approaches perform similarly. However, partial refreshes pay off for DBT^I for Q15 and Q20.

Scalability. To investigate the scalability of our approach, we repeat the insert experiment above for scale factor 10 with multiple threads. Figure 12 reports the average scaling relative to single-threaded performance for each system based on full query execution times. Flink and both Umbra variants scale nicely, Flink even shows near-perfect scaling for Q6. However, we see little improvement for Trill. We found that, while simple queries like Q6 scale in Trill, for complex queries, the majority of work still happens single-threadedly, and we see negative scale-up. We attribute this to the complex nature of the queries and the introduced scheduling overhead in multi-threaded execution.

Traditional View Maintenance. Finally, we want to look at the trade-off space between traditional view maintenance and continuous views on a common runtime. We implement continuous views in PostgreSQL, as described in Section 3.9. Further, we manually implement eager and batchwise incremental views for the TPC-H queries and include PostgreSQL's deferred views. Incremental views buffer 10K tuples before propagating changes. Neither eager nor continuous views store any input tuples. We again insert all tuples of lineitem for scale factor 1. As we measure query round-trip times, we insert chunks of 1K tuples to reduce overhead. Figure 11 reports the insert and query throughput of the approaches. For inserts, continuous views are fastest overall, except for Q1, due to the high contention on few groups. The lower throughput for deferred maintenance for Q20 is caused by an index, which we need to maintain to keep the refresh from timing out. For simple queries, like Q1 to Q3, continuous and eager views behave the same conceptually, as all processing occurs at insert time. The measured difference in query throughput is only caused by the current PostgreSQL implementation of continuous views, wherein we needlessly copy the result from the stream pipeline view to the result view for simple queries. As expected, the eager and incremental approaches offer the highest query throughput. Nevertheless, even on PostgreSQL, continuous views can offer up to a hundred refreshes per second without the need of hand optimizing views, as is the case for eager views, while still offering higher insert throughput on average.

4.3 Microbenchmark

Finally, we want to focus on the feasibility of our load balancing, which cannot be seen in query-level experiments.

Configuration. We create a workload of one stream. Its tuples comprise 6 integer values between 1 and 10,000. Views group by a single key and calculate the sum of one column.

Experiment. To help with load spikes, our queue should not grow indefinitely, as high memory consumption for the queue could decrease the overall performance. A single view creates a maximal load for the queue, as all concurrent inserts then focus on it. Hence, this is the worst case for potential unlimited queue growth. We display the results of this experiment in Figure 13. Queue length is reported after every 100 inserts. The average queue length for 10 threads seems to grow slightly after the initial spike. However, one can see that it drops to near zero consistently throughout the experiment. This could indicate the need for active maintenance for even higher thread counts. However, please keep in mind that this experiment represents the worst case. For fewer threads, we only see a small initial spike at around 10K to 50K tuples, with queue length scaling with thread count. Overall the queue works as intended, accepting tuples in the beginning when the load is high due to hash table groups being created. Once all groups have been created, the buffered tuples are processed, and the queue empties again.

5. RELATED WORK

To the best of our knowledge, we are the first system to implement analytical views for high-velocity data with a specialized maintenance strategy. There is, however, extensive previous work on materialized view maintenance, stream processing, and continuous query evaluation. We will use this section to summarize those most relevant to our work.

5.1 View Maintenance

Managing materialized query results in the form of materialized views is a well-studied problem [18,44]. Our continuous views can be thought of as materialized views specialized for streaming data, where full refreshes are only performed at query time. Materializing parts of queries has been previously suggested by Mistry et al. [34] in the context of multi-view optimizations. They share common parts of queries between multiple views for more efficient maintenance. The concept of reusing partial results has been extended to regular query processing [26, 53].

Delaying the maintenance task has been previously described in the form of both deferred maintenance [12] and lazy evaluation [52]. In contrast to our approach, these systems need to have access to the changes to the base table, either in the form of deltas or of auxiliary tables. Storing only changes required for maintenance is known as incremental view maintenance [6,16,49]. We apply similar techniques for stream pipelines where incremental deltas are held in local states and materialization in the global state. Incremental view maintenance has been further optimized using hierarchical deltas by Koch et al. [24]. In contrast to our insert-optimized approach, they optimize for query throughput, mostly triggering full refreshes for every insert.

5.2 Stream Processing

Stream processing is a broad area of data processing, spanning both dedicated systems and traditional databases. We will focus on recent work and analytical systems.

Modern Stream Processing. There is a wide range of recent work in dedicated stream processing engines, recently surveyed by Zeuch et al. [51], but most systems are optimized for stateless or simple stateful queries. To increase performance, Grulich et al. [17] also utilize query compilation for stream processing. Incremental updates, which we use for data streams, are utilized by SPEs as well [1, 10, 35, 50]. To support aggregating queries in stream processing more efficiently, some work proposes sharing state between long-running queries [15, 28]. As many SPEs do not support complex analytical workloads, dedicated solutions for more stateful queries on data streams have been developed [7, 10, 13, 19, 43]. While we know of no other work implementing stream views with specialized maintenance for database systems, using views to speed up simple analytical queries over streams [14, 49], as well as materializing continuous query results [3] has been previously suggested. For analytical streaming systems, custom query languages are common, as shown in recent surveys [20, 27]. With our integration into Umbra, we allow for stream processing using regular SQL statements. Finally, accessing historic stream data with new queries, as is possible by querying our continuous views, has been described by Chandrasekaran et al. [11].

Stream Processing in Databases. Apart from dedicated and stand-alone solutions for stream processing, recent work has focused on integrating data streams in relational databases and data warehouses [36, 48]. Jain et al. propose a streaming SQL standard [21], an idea that has been further refined by Begoli et al. [5] to allow a single standard and simplify the usage of both tables and streams in SQL statements. Meehan et al. [33] describe how stream processing and OLTP transactions with isolation guarantees can be combined in a single system. Others have made a case for a greater need for in-database processing of signal streams [40]. There have also been some open-source extensions to databases that allow for stream processing within the database [42]. Most work for high-velocity data processing focuses on the previously described maintenance of materialized views. Nikolic et al. [41] further extend higher-order incremental view maintenance to batches and distributed environments, which are common in streaming systems, e.g., in Spark [50]. They further outline the advantages of pre-filtering and aggregation in these batches for incremental maintenance, which we extend to join processing. We apply this to the whole stream instead of batches in our stream pipelines. Our work continues all these efforts by describing an insert-optimized stream view completely expressible with regular SQL statements.

Continuous Query Evaluation. Finally, our work touches upon continuous queries. While continuous views are not full continuous queries, the underlying concept of updating the query result for arriving tuples and reporting intermediate results is similar. Continuous query evaluation [4, 31, 45] focuses on keeping track of changes to an underlying query like we do within the views. In contrast to our approach, the goal is to alert users whenever the query matches defined triggers. There has been some work on whole systems dedicated to such monitoring and change detection [2, 9, 22]. While we currently do not support triggers, those could be realized by periodic queries to the continuous views.

6. CONCLUSION

In this paper, we introduced continuous views, materialized views optimized for high-velocity streaming data. To maintain these views, we introduced a novel split maintenance strategy, performing parts of the query at insert time and finalizing query processing when results are required. We demonstrated the feasibility of our approach by integrating these views into our state-of-the-art database system Umbra, using the compiling query execution engine as well as the query optimizer for fast and low-overhead maintenance. We explained how this integration allows users to access stream results in analytical workloads and durable state for queries on streams efficiently.

To demonstrate the capability of our views' split maintenance strategy, we compared it to modern stream processing engines, as well as view maintenance strategies in both Umbra and PostgreSQL. Our approach outperforms the former on analytical workloads and the latter on insert throughput, often by an order of magnitude, creating an ideal fusion of analytical query processing and high-velocity data.

7. ACKNOWLEDGMENTS

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 725286).

8. REFERENCES

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik. The design of the Borealis stream processing engine. In *CIDR*, pages 277–289. www.cidrdb.org, 2005.

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB J.*, 12(2):120–139, 2003.

[3] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. STREAM: the Stanford stream data manager. In *SIGMOD Conference*, page 665. ACM, 2003.

[4] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Rec.*, 30(3):109–120, 2001.

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

[6] J. A. Blakeley, P. Larson, and F. W. Tompa. Efficiently updating materialized views. In *SIGMOD Conference*, pages 61–71. ACM Press, 1986.

[7] L. Braun, T. Etter, G. Gasparis, M. Kaufmann, D. Kossmann, D. Widmer, A. Avitzur, A. Iliopoulos, E. Levy, and N. Liang. Analytics in motion: High performance event-processing AND real-time analytics in the same database. In *SIGMOD Conference*, pages 251–264. ACM, 2015.

[8] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[9] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Monitoring streams - A new class of data management applications. In *VLDB*, pages 215–226. Morgan Kaufmann, 2002.

[10] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *PVLDB*, 8(4):401–412, 2014.

[11] S. Chandrasekaran and M. J. Franklin. Streaming queries over streaming data. In *VLDB*, pages 203–214. Morgan Kaufmann, 2002.

[12] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *SIGMOD Conference*, pages 469–480. ACM Press, 1996.

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

[14] T. M. Ghanem, A. K. Elmagarmid, P. Larson, and W. G. Aref. Supporting views in data stream management systems. *ACM Trans. Database Syst.*, 35(1):1:1–1:47, 2010.

[15] L. Golab, K. G. Bijay, and M. T. Özsu. Multi-query optimization of sliding window aggregates by schedule synchronization. In *CIKM*, pages 844–845. ACM, 2006.

[16] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *SIGMOD Conference*, pages 328–339. ACM Press, 1995.

[17] P. M. Grulich, S. Breß, S. Zeuch, J. Traub, J. von Bleichert, Z. Chen, T. Rabl, and V. Markl. Grizzly: Efficient stream processing through adaptive query compilation. In *SIGMOD Conference*, pages 2487–2503. ACM, 2020.

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

[19] D. Gyllstrom, E. Wu, H. Chae, Y. Diao, P. Stahlberg, and G. Anderson. SASE: complex event processing over streams (demo). In *CIDR*, pages 407–411. www.cidrdb.org, 2007.

[20] M. Hirzel, G. Baudart, A. Bonifati, E. D. Valle, S. Sakr, and A. Vlachou. Stream processing languages in the big data era. *SIGMOD Rec.*, 47(2):29–40, 2018.

[21] N. Jain, S. Mishra, A. Srinivasan, J. Gehrke, J. Widom, H. Balakrishnan, U. Çetintemel, M. Cherniack, R. Tibbetts, and S. B. Zdonik. Towards a streaming SQL standard. *PVLDB*, 1(2):1379–1390, 2008.

[22] D. Kifer, S. Ben-David, and J. Gehrke. Detecting change in data streams. In *VLDB*, pages 180–191. Morgan Kaufmann, 2004.

[23] A. Kipf, V. Pandey, J. Böttcher, L. Braun, T. Neumann, and A. Kemper. Scalable analytics on fast data. *ACM Trans. Database Syst.*, 44(1):1:1–1:35, 2019.

[24] C. Koch, Y. Ahmad, O. Kennedy, M. Nikolic, A. Nötzli, D. Lupei, and A. Shaikhha. DBToaster: higher-order delta processing for dynamic, frequently fresh views. *VLDB J.*, 23(2):253–278, 2014.

[25] A. Koliousis, M. Weidlich, R. C. Fernandez, A. L. Wolf, P. Costa, and P. R. Pietzuch. SABER: window-based hybrid stream processing for heterogeneous architectures. In *SIGMOD Conference*, pages 555–569. ACM, 2016.

[26] Y. Kotidis and N. Roussopoulos. A case for dynamic view management. *ACM Trans. Database Syst.*, 26(4):388–423, 2001.

[27] N. Koudas and D. Srivastava. Data stream query processing. In *ICDE*, page 1145. IEEE Computer Society, 2005.

[28] S. Krishnamurthy, C. Wu, and M. J. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD Conference*, pages 623–634. ACM, 2006.

[29] V. Leis, P. A. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD Conference*, pages 743–754. ACM, 2014.

[30] V. Leis, K. Kundhikanjana, A. Kemper, and T. Neumann. Efficient processing of window functions in analytical SQL queries. *PVLDB*, 8(10):1058–1069, 2015.

[31] L. Liu, C. Pu, R. S. Barga, and T. Zhou. Differential evaluation of continual queries. In *ICDCS*, pages 458–465. IEEE Computer Society, 1996.

[32] J. Meehan, C. Aslantas, S. Zdonik, N. Tatbul, and J. Du. Data ingestion for the connected world. In *CIDR*. www.cidrdb.org, 2017.

[33] J. Meehan, N. Tatbul, S. Zdonik, C. Aslantas, U. Çetintemel, J. Du, T. Kraska, S. Madden, D. Maier, A. Pavlo, M. Stonebraker, K. Tufte, and H. Wang. S-Store: Streaming meets transaction processing. *PVLDB*, 8(13):2134–2145, 2015.

[34] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *SIGMOD Conference*, pages 307–318. ACM, 2001.

[35] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *SOSP*, pages 439–455. ACM, 2013.

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

[37] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

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

[39] T. Neumann, T. Mühlbauer, and A. Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *SIGMOD Conference*, pages 677–689. ACM, 2015.

[40] M. Nikolic, B. Chandramouli, and J. Goldstein. Enabling signal processing over data streams. In *SIGMOD Conference*, pages 95–108. ACM, 2017.

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

[42] PipelineDB - high-performance time-series aggregation for postgresql. `https://github.com/pipelinedb/pipelinedb`.

[43] S. A. Shaikh and H. Kitagawa. Streamingcube: A unified framework for stream processing and OLAP analysis. In *CIKM*, pages 2527–2530. ACM, 2017.

[44] O. Shmueli and A. Itai. Maintenance of views. In *SIGMOD Conference*, pages 240–255. ACM Press, 1984.

[45] D. B. Terry, D. Goldberg, D. A. Nichols, and B. M. Oki. Continuous queries over append-only databases. In *SIGMOD Conference*, pages 321–330. ACM Press, 1992.

[46] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. V. Ryaboy. Storm@twitter. In *SIGMOD Conference*, pages 147–156. ACM, 2014.

[47] T. Urhan and M. J. Franklin. XJoin: A reactively-scheduled pipelined join operator. *IEEE Data Eng. Bull.*, 23(2):27–33, 2000.

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

[49] Y. Yang, L. Golab, and M. T. Özsu. ViewDF: Declarative incremental view maintenance for streaming data. *Inf. Syst.*, 71:55–67, 2017.

[50] 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, and I. Stoica. Apache Spark: a unified engine for big data processing. *Commun. ACM*, 59(11):56–65, 2016.

[51] S. Zeuch, S. Breß, T. Rabl, B. D. Monte, J. Karimov, C. Lutz, M. Renz, J. Traub, and V. Markl. Analyzing efficient stream processing on modern hardware. *PVLDB*, 12(5):516–530, 2019.

[52] J. Zhou, P. Larson, and H. G. Elmongui. Lazy maintenance of materialized views. In *VLDB*, pages 231–242. ACM, 2007.

[53] J. Zhou, P. Larson, J. C. Freytag, and W. Lehner. Efficient exploitation of similar subexpressions for query processing. In *SIGMOD Conference*, pages 533–544. ACM, 2007.