µWheel: Aggregate Management for Streams and Queries

Max Meldrum
mmeldrum@kth.se
KTH Royal Institute of Technology
Stockholm, Sweden

Paris Carbone
parisc@kth.se
KTH Royal Institute of Technology
RISE Research Institutes of Sweden
Stockholm, Sweden

ABSTRACT

Aggregate management is equally significant for both streaming and query workloads. However, the prevalent approach of separating stream processing and query analysis impairs performance, hinders aggregate reuse, increases resource demands, and lowers data freshness. μWheel addresses this problem by unifying aggregate management needs within a single system optimized for continuous event streams. μWheel pre-aggregates and indexes timestamped data arriving out-of-order, enabling the sharing of aggregates across arbitrary time intervals while respecting low watermarks. Our performance analysis demonstrates that μWheel dramatically outperforms current aggregate sharing techniques for high-volume streaming, particularly when handling numerous concurrent window slides. Crucially, μWheel also delivers performance comparable to specialized pre-aggregation indexes for supporting ad-hoc queries and does so with significantly reduced storage requirements. μWheel's efficiency stems from its compact wheel-based data layout, featuring implicit timestamps, a query-agnostic time hierarchy, and a query optimizer designed to minimize aggregate operations.

KEYWORDS

aggregate management, embedded analytics, stream processing

ACM Reference Format:
Max Meldrum and Paris Carbone. 2024. µWheel: Aggregate Management for Streams and Queries. In *The 18th ACM International Conference on Distributed and Event-based Systems (DEBS '24), June 24–28, 2024, Villeurbanne, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3629104.3666031

1 INTRODUCTION

Modern data processing needs impose the capability to perform both online streaming computation and offline ad-hoc analysis on the same input data. Currently, the combination of low-latency event streaming workloads with adhoc queries is not natively feasible to achieve in unison, requiring practitioners to execute each workload to its corresponding system silo. For instance, Uber [17] utilizes Apache Flink [11] to process data in real-time for stream processing needs and additionally leverages Apache Pinot [23] to serve exploratory queries on the same data since Flink and other stream processors [5, 20, 25, 32, 33] do not offer this capability. Contemporary stream processing systems employ state backends or indexes passively, focusing solely on managing the storage of aggregates for streaming queries. This eliminates the prospect of reusing these aggregates produced online for offline analytical aggregation. This practice results in resource overutilization when both forms of analytics are required, necessitating the deployment of a separate system. Despite aggregating the same data, different storage technologies are typically utilized while mandating an external ETL process which leads to excessive complexity. An additional consideration pertains to data freshness. Offline analysis traditionally lags behind real-time insights, with data often minutes or hours old due to reliance on bulk- instead of stream-ingestion. The prospect of utilizing a common system that manages and indexes aggregate state could jointly permit both online and offline queries to obtain exceptionally up-to-date results. However, given that these two diverse technologies have evolved and matured independently, the challenge lies at achieving shared aggregate management transparently without altering their principled design. Merely enhancing stream processors with query capabilities, or alternatively, introducing stream event ingestion into traditional OLAP databases has proven to be highly complex in practice due to the paradigm mismatch of these two technologies [15, 16, 34]. For example, state-of-the-art stream processors employ out-of-order processing semantics through low watermarking which is incompatible with the transactional nature of OLTP databases or the design of append-only OLAP databases.

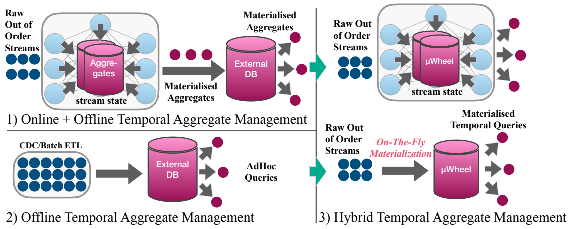

Figure 1: μWheel for streaming and adhoc queries

In this work, we investigate the prospect of a new form of aggregate management middleware that enables this marriage of workloads for combined stream and analytical use. To that end, we propose μWheel, an aggregate management system capable of sustaining high-throughput stream aggregates and external temporal adhoc queries. μWheel can be used as either a standalone system or embedded as a state backend plugin within existing stream processing engines. μWheel employs pre-materialization in multiple event-time dimensions while respecting event-time progress via its

DEBS '24, June 24–28, 2024, Villeurbanne, France

ACM ISBN 979-8-4007-0443-7/24/06.
https://doi.org/10.1145/3629104.3666031

native adoption of low-watermarking [2, 4, 30]. Furthermore, its built-in optimizer allows for seamless refinement and tuning of arbitrary temporal range-based aggregate functions used within sliding windows or ad-hoc exploration based on SIMD and other strategies that are termed optimal depending on the algebraic properties of the provided user-defined functions.

Contributions. In this paper we provide a thorough design overview of the μWheel aggregate management system which builds as an adaption and extension of compact hierarchical time wheel data structures [45] often used today in operating systems for efficient scheduling purposes. More concretely we detail the following contributions:

(1) We indicate motivating evidence towards the need for combined stream and query management under the same system architecture (section 2).
(2) We describe the internals and optimization mechanisms of μWheel (section 3), an aggregate management system for streams and queries written in Rust . μWheel features a novel wheel-centric query optimizer (section 3.4.4) that minimizes the number of required aggregate operations. Next, μWheel proposes a lazy aggregate synchronization (section 3.6) approach via low watermarking enhancing performance under high-volume *stream* workloads.
(3) We employ a detailed performance evaluation (section 4.2) of *μWheel* against state-of-the-art window aggregation sharing techniques (e.g., FiBA [39]) and show its distinct performance benefits. μWheel has up to 9x higher throughput under write-intensive streams in combination with a high number of concurrent window slides.
(4) Finally, we compare the analytical performance (section 4.3) on temporal queries in μWheel against both pre-aggregating and on-demand solutions.μWheel exhibits comparable performance to specialized pre-aggregation indexes with significantly less memory resources. In addition, μWheel outperforms on-demand solutions such as DuckDB [35] for queries on arbitrary time ranges by orders of magnitude.

2 MOTIVATION AND OVERVIEW

Common stream aggregations and temporal adhoc queries both evidently serve a common purpose, that is the need to materialize computations over time intervals. The core difference lies in the granularities of focus in these two types of computation. Namely, stream window aggregates [13, 47] operate at the length of seconds or minutes, whereas offline analytical queries require higher-order aggregation, e.g., at the level of hours, days or months. μWheel materializes aggregates on-the-fly over arbitrarily configured time dimensions to serve both of these needs.

	Techniques	Temporal Aggregation	Out-of-order Stream Ingestion	Query Optimizer
🟥 SQLite [18]	Index(B-Tree)	dynamic; fixed	✖	✔
🟥 DuckDB [35]	SIMD; Index(ART)	dynamic; fixed	✖	✔
🟦 PBA[51]/Cutty[14]	slicing; Multi-core	static; sliding	✖	✖
🟦 Scotty[44]	slicing	static; sliding	✔	✖
🟦 FiBA [39]	Index(Finger B-Tree)	dynamic; sliding	✔	✖
🟥 🟦 μWheel	slicing; Index(wheels); SIMD	dynamic; sliding	✔	✔

Table 1: Feature comparison of top-performing, embeddable 🟦 stream and 🟥 query aggregate management solutions

Motivating Example. Figure 1 showcases two alternative configurations used to support temporal adhoc queries: I) One alternative includes the combination of a streaming system with an external offline database. The streaming system in this case is responsible for materializing different window aggregates from out-of-order streams. The produced aggregates are then inserted into an external database which is responsible for answering external adhoc queries based on the pre-computed results. II) Alternatively, as depicted in Figure 1, input data can also be fed directly into a dedicated OLAP database (e.g., DuckDB [35]) through "change data capture" (CDC) or batching. This enables on-demand temporal queries on arbitrary time dimensions, offering ad hoc analysis flexibility. However, the lack of stream capabilities hinders real-time decision-making. Among the two cases (I, II) we observe an overarching need to compute and maintain aggregates. In case I this requirement is posed on the streaming system used to facilitate the pre-aggregates, at the expense of maintaining two redundant storage systems and writing custom aggregation logic in the streaming system. These limitations cannot be circumvented since stream aggregation solutions are limited to embedded use [10] and further lack the ability to pre-materialize multi-dimensional computations by design. Case II relies solely on an offline external database and therefore inherently lacks support for pre-materialization, necessitating queries to wait until all results are computed from scratch. As we illustrate in case III, μWheel can benefit both configurations by either enhancing a stream ETL pipeline with external query support or substituting the database altogether. As an example, consider the following aggregation queries involving online and offline use respectively:

```
# Online Stream Aggregation (Sliding Window)
SELECT HOP(window_start , INTERVAL '1' MINUTE),
SUM(page_views)
FROM website_traffic
GROUP BY HOP(TUMBLE(rowtime, INTERVAL '5' MINUTE)
AS window_start, INTERVAL '1' MINUTE)

# Offline Analytical Aggregation
SELECT SUM(page_views)
FROM website_traffic
WHERE rowtime
BETWEEN '2023-03-01 10:00:00'
AND '2023-03-20 10:30:00'
```

While both queries calculate page view aggregations, the first provides continuous updates on recent website traffic using a sliding window (Flink-style SQL), whereas, the latter targets offline analysis over a time range. μWheel with its unified stream and analytical storage can optimize both queries by effectively reusing pre-materialized aggregates. This leads to faster query responses, reduced computational overhead, and minimized resource utilization, having a single system to serve both queries.

A closer consideration of available embeddable stream and query aggregate management systems reveals a mismatch for this joint purpose. To that end, we overview a short feature comparison in Table 1 and further examine existing capabilities in more detail.

□ **Requirements for Stream Management.** Solutions well-suited for stream analysis share these core characteristics: 1) stream ingestion of out-of-order data, 2) event-time integration, and 3) native support for *sliding* temporal aggregations (e.g., periodic window aggregation). Furthermore, to enable flexible user-defined aggregate functions, streaming solutions often rely on low-level aggregation frameworks. Embedded query aggregate management solutions like DuckDB prioritize bulk inserts and append-only workloads, making them less efficient for streaming scenarios. Finally, OLTP databases like SQLite, designed for transactional workloads and high concurrency, are not ideal for stream analytics. The overhead of full SQL support and concurrency mechanisms designed for multiple writers can hinder performance in stream scenarios where there's typically a single writer.

□ **Requirements for Query Management.** The embedded query aggregate management landscape is dominated by DuckDB for relational analytics. Most existing row-based, transactional databases are considered unsuitable for advanced offline analytical aggregation that require features like multi-dimensional cubes and columnar aggregates. At the same time, query systems typically have low requirements in ingestion performance, but relatively higher requirements in the declarativity and generality of possible *dynamic* queries that can often be computed on the fly. Furthermore, to efficiently execute complex analytical queries on modern hardware, query systems typically feature query optimizers and vectorization (SIMD) support. These capabilities are notably absent in existing stream management solutions, as observed in Table 1.

3 DESIGN

μWheel is an event-driven aggregate management system for the ingestion, computing, and indexing of temporal stream aggregates. We first describe the underlying aggregation framework followed by the distinct internal details of its wheel-based system.

3.1 Aggregation Framework

The μWheel aggregation framework supports custom user-defined aggregation functions, tailored for both online stream-style pre-aggregation and ad-hoc offline aggregation over flexible time intervals. In the context of data streaming, μWheel makes no unrealistic in-order processing assumptions. Instead, it adopts out-of-order processing semantics via external low-watermarking [2]. To ensure correctness, the framework requires all provided aggregation functions to bear basic algebraic properties such as commutativity and associativity. Optionally, if present, invertibility is further exploited by μWheel to offer improved query performance as shown in Sec. 3.4.2.

μWheel shares similarities with existing frameworks [41] and builds on five of type functions: `lift`, `combine_mutable`, `freeze`, `combine` and `lower`. `Combine_mutable` and `Freeze` are two distinct functions to μWheel introduced to facilitate integration with low-watermarking. We denote MA as a mutable partial aggregate type, whereas A is immutable. The `lift`: $E \longrightarrow MA$ function maps an input record (type E) to a mutable partial aggregate type. `Combine_mutable`: $MA \odot E$ applies an in-place aggregation on a mutable partial aggregate type using an input record. `Freeze`: $MA \longrightarrow A$ converts a mutable partial aggregate type to an immutable one. `Combine`: $A \oplus A \longrightarrow A$ performs an incremental aggregation of two immutable partial aggregates, resulting in a new partial aggregate. Lastly, `Lower`: $A \longrightarrow E'$ transforms an immutable partial aggregate into a final aggregate type (e.g., sum/count ⟶ avg).

3.2 Architectural Overview

The architecture of μWheel adopts a single-writer, multiple-reader access pattern designed around the notion of low watermarks [2, 4, 30] as follows: A low watermark `w` indicates that all records with timestamps $t \leq$ `w` have been ingested. μWheel takes advantage of this property and the single-writer principle that is dominant in dataflow processing systems [11, 33] and separates the write and read paths. Partial aggregates are indexed by timestamps and can be updated until the system's low watermark has advanced above their assigned timestamps. All aggregate operations are handled by internal data structures called "wheels". Writes are handled by a *writer* wheel (sec. 3.3), which supports in-place aggregation and is optimized for single-threaded ingestion. Whereas, reads are managed by a hierarchical time-indexed *reader* wheel (sec. 3.4) equipped with a query optimizer whose cost function aims to minimize the total number of aggregate operations.

Interface. μWheel exposes five core functions:

- `create(a: Aggregator,` w_{start}`: Watermark, c: Config)` initializes a μWheel with a user-defined Aggregator containing the types and functions detailed in Sec. 3.1, a start low watermark w_{start}, and a config object which holds tunable parameters (detailed in sec. 3.3 and 3.4).
- `insert(v: E, t: Time)` inserts a record `v` at time `t`.
- `window(range: Time, slide: Time)` pre-configures a periodic sliding window. This is used to optimise and manage incremental window aggregates. Complete windows are returned by μWheel when advancing its watermark (sec 3.6).
- `advance_to(t: Time)` advances the internal watermark to time t if t is higher than the current watermark, finalizing all pre-aggregates up to time t and returning all complete window aggregates in a vector (if any). We often adopt the term "ticking" to denote the internal time advancement.
- `query(start: Time, end: Time)` returns the final aggregate computed between the start and end timestamps.

Example. Figure 2 details how the wheel-based system works. μWheel maintains multiple time-indexed circular buffers, which we call wheels, one for each respective time resolution (i.e., seconds, minutes, etc.). Each wheel is characterized by a head and tail pointer that indicates its elapsed rotation progress. The writer wheel and the reader wheel's lowest-level wheel are kept in sync through a shared granularity and watermark. Consequently, new insertions consistently occur at or above the latest computed low watermark. Figure 2 depicts the process of ticking a μWheel (i.e. adding time progress) which is done exclusively through watermark progressions via the `advance_to` function. Upon advancing the current low watermark to *09:00:00* in the example, a full rotation (shifting of H) occurs in all wheels, causing a summarization of aggregates across all wheels. The example query (sum over [6am-9am]) computes the sum over three slots in the hour dimension, accessing complete aggregate results since the watermark (H in hour-wheel) has progressed over the intended slots.

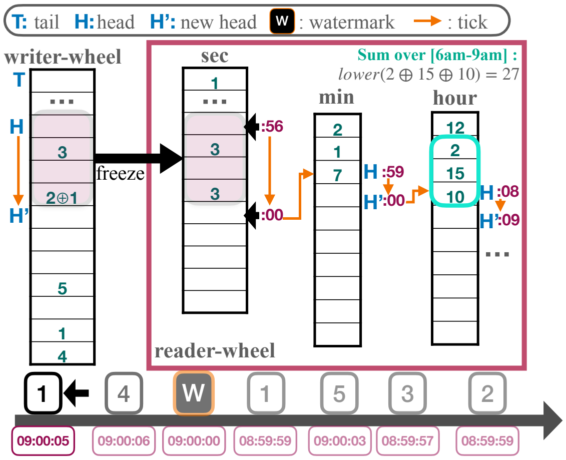

Figure 2: Ingestion, ticking and querying in μWheel

3.3 Writer Wheel

The writer wheel is a time-indexed circular buffer designed to ingest stream aggregates at high throughput. Internally, the wheel nests two different wheels: a `Write-ahead` and an `Overflow Wheel`.

Write-ahead Wheel. A pre-allocated fixed-sized wheel that supports in-place aggregation of slots above or equal to the current low watermark. Users of μWheel may configure the write-ahead capacity at creation time by passing it in as a configuration. For instance, a write-ahead wheel configured with a capacity of 64 slots allows pre-aggregation up to 64 time units (e.g., seconds). Inserts into the write-ahead wheel are efficient as the slots are event-time indexed, allowing direct access to any given slot.

Overflow Wheel. Events with timestamps that exceed the capacity of the write-ahead wheel are scheduled into a Hierarchical Timing Wheel [45] and inserted into the write-ahead wheel once the low watermark has advanced far enough.

Mechanism. Algorithm 1 shows the pseudocode for both insertions and ticking (i.e., time advancement). The insert function checks for three possible scenarios. First, if the given timestamp is below the current watermark, the record is considered late and is rejected (line 3). Furthermore, if the record fits within the write-ahead wheel (line 6), then the function aggregates the record through a `combine_mutable` ⊙ operation if the target wheel slot contains an existing entry (line 9) or using the `lift` operator to fill the empty slot (line 11). Otherwise the record is scheduled into the overflow wheel (line 13) if the timestamp is too far ahead of the low watermark.

Ticking the writer wheel (line 15) shifts the low watermark and advances the overflow wheel across consecutive atomic time units (seconds) (lines 16-17). Previously early records that now fit within the write-ahead wheel's time range are added (line 19). Lastly, the function updates the internal head and tail pointers before returning the aggregate of the old low watermark (line 23) which is to be inserted into the reader wheel (sec. 3.4).

Complexity Analysis. Insertions in the writer wheel have a time complexity of O(1) since both the write-ahead and overflow wheels

Algorithm 1 Writer Wheel Insert and Tick

```
1:  function INSERT($self, record, timestamp$)
2:      if $timestamp < self.watermark$ then
3:          return Err($Late$)
4:      else
5:          $seconds \leftarrow timestamp - self.watermark$
6:          if $self.can\_write\_ahead(seconds)$ then
7:              $slot \leftarrow self.lookup(seconds)$
8:              if $self.slots[slot] \neq$ None then
9:                  $self.slots[slot] \odot record$
10:             else
11:                 $self.slots[slot] \leftarrow$ lift($record$)
12:         else
13:             $self.schedule\_overflow(timestamp, data)$
14:     return Ok(())
15: function TICK($self$)$\rightarrow$ $Aggregate$
16:     $self.watermark \leftarrow self.watermark + seconds(1)$
17:     $entries \leftarrow$ self.overflow.advance(self.watermark)
18:     for (record, timestamp) in $entries$ do
19:         $self.insert(record, timestamp)$
20:     $head \leftarrow self.wrap\_add(head, 1)$
21:     $tail \leftarrow self.tail$
22:     $self.tail \leftarrow self.wrap\_add(tail, 1)$
23:     return $slots[tail]$
```

maintain constant complexities. The writer wheel demands O(n) space due to the overflow wheel's linear complexity in the number of slots it can schedule.

3.4 Reader Wheel

The reader wheel indexes complete aggregates hierarchically across multiple event-time dimensions. It employs a novel wheel-centric query optimizer whose cost function minimizes the required number of aggregate operations for a given query.

3.4.1 Overview. Driven by the hierarchical nature of time, a reader wheel is composed of different circular-based data structures called `wheels`, each maintaining event-time indexed aggregates across a different time granularity. We organize wheels into a data structure which we call Hierarchical Aggregate Wheel. Its hierarchical layout in combination with implicit timestamps enables a compact representation of aggregates with a low memory footprint across time dimensions.

Mechanism. Each wheel within the Hierarchical Aggregate Wheel maintains regular slots, each representing a concrete time unit (e.g., 1 hour). In addition, the wheel also manages a partial aggregate of its current rotation and returns it after completing a full rotation. This way, a wheel can be re-used across different hierarchical levels (e.g., seconds, minutes, hours, and days). For instance, a wheel representing hours would fully rotate once ticked to the rotation point of 24 and return a complete one-day rolled-up partial aggregate. In this paper, we assume granularities from seconds to years. For example, to represent aggregates spanning 10 years, we make use of the following wheels: *seconds(60), minutes(60), hours(24), days(7), weeks(52), and years(10).* Such a configuration involves a total of

213 wheel slots to support rolling up aggregates across ten years with second granularity. User-defined aggregate schemes may be configured at creation time (sec. 3.2). By design, wheels limit data retention by not maintaining slots exceeding their capacity. However, In order to support a wider range of analytics, data retention policies within wheels can be customized. This configuration guarantees slots are held for a specific period. For example, if consistent access to the previous day's aggregates at second-level granularity is necessary, the seconds wheel should be instantiated with an appropriate retention policy.

3.4.2 General Optimizations.

This section covers optimization techniques used during query planning and execution.

SIMD. The μWheel aggregation framework accommodates this capability via an optional `Combine_Simd` user-defined function. Users can implement this function with explicit SIMD instructions for accelerated queries across numerous wheel slots. If not provided, the framework defaults to decomposing a time-slice of partial aggregates into a series of combine operations (e.g., $x1 \oplus x2 \oplus .. \oplus xn$).

Invertibility. μWheel's aggregation framework introduces an optional `Combine_Inverse:` $A \ominus A \longrightarrow A$ function that instruments deduction of partial aggregates. This function offers significant speedups in scenarios where existing higher-order aggregates can be reused. By directly calculating the inverse of a partial aggregate, it avoids the need to recalculate from scratch, saving substantial computation time.

Prefix-sum Wheels. A prefix-sum [19] is an optional optimisation targeting O(1) time complexity for range queries at the expense of double the number of buffers needed to maintain wheels. An additional prefix-sum wheel is used to take a binary associative operator $\oplus$ and apply it on an array of N elements returning the following $[x0, (x0 \oplus x1), \ldots, (x0 \oplus x1 \oplus \ldots \oplus xn-1)]$. This allows for arbitrary range-sum queries to execute in O(1) time using the prefix-sum wheel. Wheels can be optionally be prefix-enabled at creation time, given that a `combine_inverse` function is also provided.

3.4.3 Data Model and Wheel Operations.

To detail the aggregation framework we further explain the data and aggregation semantics used within a reader wheel.

Wheels. We denote each wheel granularity with lowercase letters. For instance, a seconds wheel is defined as `s`. e.g., `s[begin, end)` refers to a closed-open interval of aggregates in a "seconds" wheel.

Wheel Aggregation. f_w computes aggregates within a single wheel. For example, the function $f_w(s[0, 2))$ combines two partial aggregates from a "seconds" wheel.

Combined Aggregation. Since wheel aggregations return a single aggregate, they can compose arbitrary aggregations in a single wheel or across wheels (e.g., $f_w(s[0, 2)) \oplus f_w(m[5, 10))$).

Landmark Aggregation. A landmark aggregation f_L is an operation that considers data across the whole hierarchy of wheels. In the reader wheel, this is defined by the range $[w_{start}, w_{now})$ where w_{start} represents the initial start low watermark and w_{now} the current low watermark. Each wheel in the hierarchical structure maintains a total partial aggregate for its current rotation, and we denote this aggregate as `t`. A landmark aggregation is broken down to a sequence of $\oplus$ operations across wheels:

$$f_L = s.t \oplus m.t \oplus h.t \oplus d.t \oplus w.t \oplus y.t$$

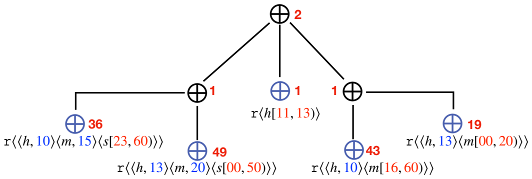

Figure 3: Combined aggregation for range [10:15:23, 13:20:50)

Inverse Landmark Aggregation. This operation takes advantage of the invertibility property if available and reuses higher-order aggregates from the landmark aggregation to compute an interval of $[t_{start}, t_{end})$. For example, assuming that the current low watermark $w_{now} > 8000$ a range query $[3000, 8000)$ may be computed as such: $f_L \ominus f_w[w_{start}, 3000) \ominus f_w[8000, w_{now})$. This calculation requires two inverse operations. However, if $t_{end} \geq w_{now}$, the operation only requires a single inverse operation. For instance, if t_{end} and w_{now} is 10000 then the range query $[3000, 10000)$ can be answered through: $f_L \ominus f_w[w_{start}, 3000)$.

3.4.4 Wheel-centric Query Optimizer.

We now describe the internals of the wheel-centric query optimizer whose cost function minimizes the number of aggregate operations.

Optimizer Hints. The optimizer leverages framework-provided hints, such as SIMD compatibility and invertibility, to further optimize query execution. For instance, if SIMD support is enabled then the query planner will favour plans where it can be fully exploited.

Cost Function. μWheel uses a cost-based model in combination with optimizer hints. Since data is pre-aggregated and event-time indexed, there is no need for cardinality estimation. The exact number of aggregates to compute is known given a range $[t_{start}, t_{end})$. However, relying on a purely cost-based model may lead to suboptimal execution times under different circumstances. For example, a range query of `[12:00:00, 18:30:00)` can be computed in two ways. Firstly, it can be calculated directly using 390 minutes: $f_w(m[n, n + 390))$. Alternatively, the range can be split and computed using *combined aggregation* : $f_w(h[12, 18)) \oplus f_w(m[0, 30))$, with a cost of 37 aggregate operations.

While the second approach involves fewer operations, the first may be faster due to optimized memory access. Furthermore, with SIMD support, single-wheel aggregation likely becomes the faster option since it can coalesce multiple operations into a single instruction. When SIMD support is absent, minimizing aggregate operations through `combined aggregation` takes priority. Consider this example query with the range `[10:15:23, 13:20:50)`. To efficiently execute this query, the generated execution plan splits the range into multiple non-overlapping ranges, as highlighted in Figure 3. Each leaf (blue $\oplus$) represents a f_w within a given granularity. This plan yields the final aggregate with up to 152 aggregate operations, a 73x lower cost compared to a naive wheel aggregation at the seconds level, which would need 11,127 $\oplus$ calls.

Finding the Best Plan. Figure 4 illustrates a flow chart of the query optimizer which aims to select the optimal execution plan. An execution plan refers to a wheel operation (sec. 3.4.3) that can be used to compute a query. The optimizer first checks whether the prefix-sum optimization (sec 3.4.2) is available to compute the

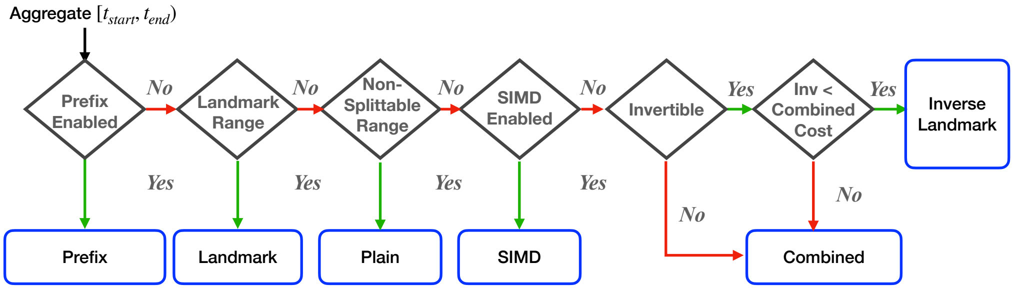

Figure 4: Query optimizer logic showcasing various wheel aggregation strategies (boxes in blue)

range $[t_{start}, t_{end})$ using the query's lowest time granularity. This operation requires at most one aggregate operation. If not, it checks if the query range falls within the landmark range $[w_{start}, w_{now})$, using landmark aggregation if possible since it requires very few aggregate operations. Failing that, if the range cannot be split into multiple non-overlapping ranges (e.g., the duration is below 60 seconds), a plain wheel aggregation is used.

Next, the optimizer assesses SIMD capabilities. By default, μWheel utilizes SIMD-enabled wheel aggregation at the query's lowest granularity if possible. Users can fine-tune this behavior with a configurable *SIMD Threshold* hint to control whether to prioritize SIMD execution or focus on reducing the number of aggregate operations. If the plan's computational cost (i.e., $|\oplus|$) exceeds the threshold, the optimizer will continue to explore ways to reduce the number of operations.

The optimizer then checks for invertibility. Without invertibility, the range is split for combined wheel aggregation across multiple time granularities. With invertibility, the optimizer compares the cost of the of inverse landmark aggregation against combined aggregation, executing the lower-cost option.

Complexity Analysis. A range `query` $[t_{start}, t_{end})$ can be computed using various wheel-based aggregation operations as highlighted in Figure 4. While the number of aggregations varies across operations, the fixed size of wheels (time dimensions) and their slot capacities ensure a constant upper bound. The chosen aggregation scheme configuration determines this specific upper bound.

3.5 Window Aggregation

In addition to base aggregates, the system internally incrementalizes and manages window aggregates specified in the API (sec. 3.2). μWheel utilizes pairs [26], a state-of-the-art stream slicing technique on periodic intervals. Pairs split the stream into two alternating slices (p1, p2) as such: *p2 = r mod s and p1 = s - p2.*

For each active window provided by the user, μWheel materializes pair generation through a circular-based window aggregator that implements the following functions:

- `push(p: Pair)` Pushes a pair into the back.
- `compute()` Computes the current window.
- `pop()` Removes the oldest pair from the front.

If any window specification is configured then μWheel will, during time advancement (sec 3.6), populate stream slices (*push*), compute window results (*compute*), and clean up old stream slices (*pop*). Since aggregates below the low watermark won't be modified, μWheel is able to use specialized in-order data structures for computing window aggregates. More specifically, μWheel uses a TwoStacks [38] based solution for non-invertible aggregate functions and Subtract-on-Evict [22] for invertible ones.

Algorithm 2 μWheel Time Advancement

```
function ADVANCE_TO($self, watermark$) $\longrightarrow Vec[Window]$
    $seconds \leftarrow watermark - self.watermark$
    $windows \leftarrow \emptyset$
    for 0 to seconds do
        $self.reader.tick(self.writer)$
        if $self.window\_configured()$ then
            $handle\_window(windows)$
    return $windows$
function TICK($self, writer$)                              ▷ Reader Wheel
    $self.watermark \leftarrow self.watermark + seconds(1)$
    if $ma \leftarrow writer.tick()$ then
        $partial\_agg \leftarrow$ freeze($ma$)
        $seconds\_wheel.insert(partial\_agg)$
    if $agg \leftarrow seconds\_wheel.tick()$ then
        $minutes\_wheel.insert(agg)$
        ...
        if $agg \leftarrow weeks\_wheel.tick()$ then
            $years\_wheel.insert(agg)$
            $years\_wheel.tick()$
function HANDLE_WINDOW($self, windows$)
    if $self.next\_pair\_end == self.watermark$ then
        $from \leftarrow self.watermark -$ len($pair$)
        $pair \leftarrow self.reader.query(from, self.watermark)$
        $self.window.push(pair)$
        if $self.next\_window\_end == self.watermark$ then
            $windows.insert(self.window.compute())$
            $self.window.pop()$
```

3.6 Lazy Synchronization

Synchronization in μWheel refers to the advancement of time, a process which shifts aggregates from the writer wheel to the reader. μWheel only performs synchronization lazily once its low watermark is advanced. This design choice has a twofold purpose. First, it enables μWheel to avoid costly index maintenance (e.g., tree rebalance typically employed in aggregation trees [39, 41]) in the hot path of writes. Secondly, it creates a clear separation between writes and reads, enabling concurrent ingestion and querying.

Mechanism. Algorithm 2 illustrates the synchronization process. It starts by calculating the number of atomic units (seconds) it can advance (line 2) and initializes an empty vector (line 3) for window results (if window support is enabled). For each second of advancement, it triggers the reader's tick function (line 5) using the writer wheel as input. The tick function updates the reader wheel by advancing the low watermark (line 10), ticking the writer wheel (line 11), freezing the result into a partial aggregate (line 12), and inserting it into the lowest granularity wheel (line 13). Next, the hierarchical wheels within the reader are also ticked as needed.

Finally, if μWheel has any windows configured (line 6) it goes ahead with window management. Window aggregates (sec. 3.5) are managed by checking if the next pair (stream slice) has ended (line 21), creating pairs by querying the reader wheel and inserting it into the window aggregator (line 23-24), computing windows when the low watermark reaches the end of a window and inserting it into the *windows* vector (line 26), and cleaning up old window aggregates (line 27).

Complexity Analysis. Time advancement in μWheel has a time complexity of O(n) where n is the number of ticks (seconds) required to advance to the new low watermark.

4 EVALUATION

The aim of μWheel is a versatile system that can be used for both stream and ad-hoc analytics. We evaluate μWheel in each aspect through experiments with online stream aggregation (sec. 4.2) and offline analytical aggregation (sec. 4.3). μWheel is publically available and provided as a Rust library[1].

4.1 Experimental Setup

All experiments were carried out on a Linux machine running kernel 5.18.0 equipped with: 2.30Ghz Intel(R) Xeon(R) Gold 5218 CPU with 251GB DRAM and 32 physical cores. The machine has access to AVX512 SIMD instructions. Our experiments include systems developed in C++ and Rust. We compile code in the former using g++ version 9.4.0 and the latter using rustc 1.77.0-nightly. Furthermore, each experiment was repeated five times. For throughput metrics, we calculated the average. To analyze the overall distribution of latency, we merged latency percentiles (e.g., p95th) across the runs.

4.2 Online Stream Aggregation

We now aim to investigate how μWheel compare to the existing state-of-the-art for streaming window aggregation.

Baselines: We compare μWheel against FiBA [39]. We include two versions of FiBA, one pre-aggregating data at the same granularity as μWheel (1-second), which we refer to simply as FiBA and a coarse-grained (CG) variant that slices at the granularity of the window slide. We opted out of including a modern stream processor such as Apache Flink since FiBA has been shown to outperform it [40].

Data. We use two real-world datasets in our evaluation. First, we replay the NYC Citi Bike dataset used in the FiBA paper which contains events from Aug-Dec 2018 with a total of 8,010,578 events, with an average of below 1 events/s and 45.76% arriving out-of-order. Secondly, we use the DEBS12 Grand Challenge dataset [24] of manufacturing machines, a common dataset used to evaluate streaming systems [14, 42] with 32,390,519 events recorded between 22nd of Feb 2012 and 20th of March 2012, with an average of 100 events/s and 1.5% events arriving out-of-order.

Configuration: For μWheel we use two versions with write-ahead sizes of 64 and 512 slots. In addition, we disable optimizations related to the invertibility property. For FiBA, we use two bfinger variants with tree arity of 4 and 8 as in the original paper. We include bulk eviction functionality from the latest work [40] but no bulk inserts, as the evaluation focuses on online stream aggregation. In addition, bulk inserts in FiBA require batches to be sorted in timestamp order. FiBA and its tree structure are not designed with a pre-allocated write-ahead section. However, we configure FiBA to use the mimalloc [27] allocator, which lowers the tail latencies.

Queries. We run sliding window aggregation queries with varying range and slide values. For all executions, we use a sum aggregation function of a 64-bit unsigned integer. For the NYC Citi Bike dataset, we calculate the sum of trip durations, whereas for DEBS12, we compute the aggregated energy consumption for one of the sensors. We use the same watermark generator configured with a frequency of 100 for all solutions. i.e., every 100 events the generator progresses and disseminates the watermark which causes complete window computations downstream.

4.2.1 *Impact of range and slide ratio.*

We now study the impact of different range and slide ratios. High ratios are easier to manage and compute due to the low number of concurrent slides involved compared to low ratios, which could potentially require the management of thousands of concurrent slides for each window.

Workload. We run two sliding window aggregation setups with a slide ratio of 3 and 300. We increase the range and slide in both setups while maintaining the same ratio.

High ratio. As shown in Figure 5 μWheel and coarse-grained FiBA exhibit comparable and consistent performance over all NYC Citi Bike dataset window setups. Regular FiBA, on the other hand suffers from a significant performance drop having a growing window slide size as it has to maintain more tree nodes. μWheel with a write-ahead capacity of 64 slots have reduced performance compared to the 512-slot variant. To our knowledge, this is due to excessive out-of-order events exceeding its capacity. In turn, this leads to aggregates being scheduled into μWheel's overflow wheel (sec. 3.3) to be processed later once the watermark has advanced. In the more write-intensive DEBS12 machine dataset, we observe that μWheel outperform FiBA in all setups. In this mostly in-order dataset, all events fit within the wheels write-ahead section, contributing to consistent performance.

Low ratio. Figure 6 reveals that both FiBA variants suffer from significant drops in throughput when there is an increase in the number of concurrent window slides. The best performing FiBA variant has on average 9x worse throughput in the DEBS12 dataset and 11x in the NYC Citi Bike dataset.

Summary. We observe that μWheel shows consistent performance with varying slide ratios and growing window sizes. In addition, μWheel outperforms FiBA when the write intensity is high or when there is an increasing number of concurrent window slides.

[1] https://github.com/uwheel/uwheel

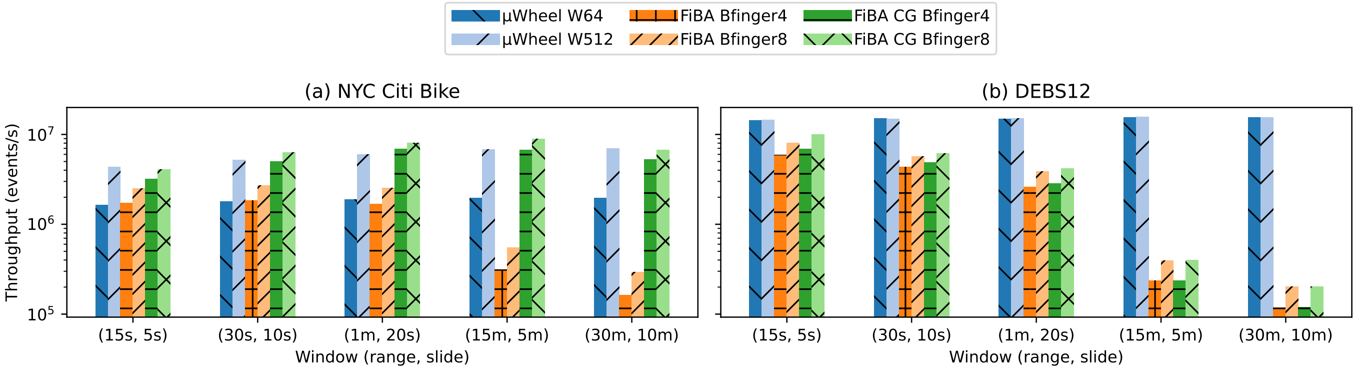

Figure 5: Streaming Window Aggregation (SUM) Throughput - High ratio

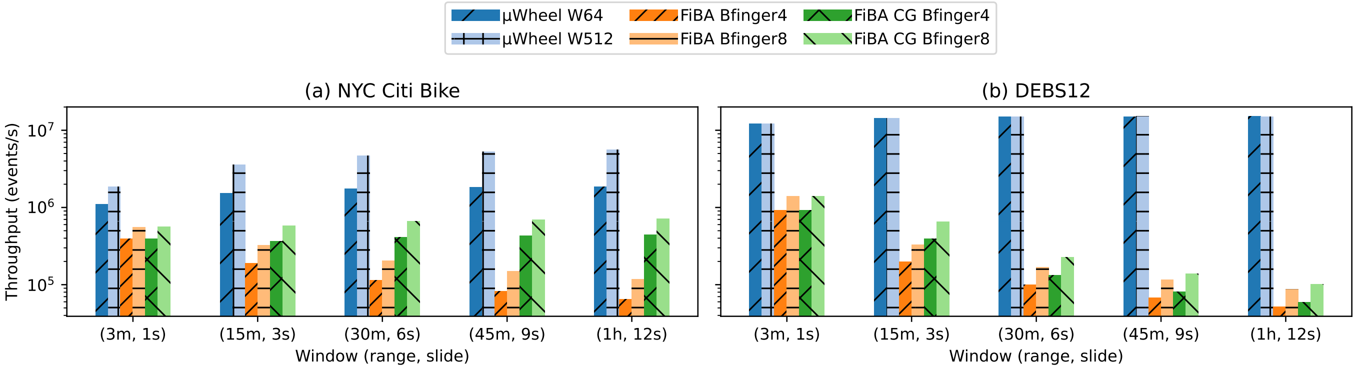

Figure 6: Streaming Window Aggregation (SUM) Throughput - Low ratio

4.2.2 Lazy Synchronization. What is the impact of Lazy Synchronization (sec 3.6)?. First,we look at the workload distribution of two real-world datasets to see how much time is spent on inserts. Secondly, we study the latency of insertions. For the remainder of this section, we'll refer to the low-ratio execution described in Section 4.2.1

Workload Distribution. Figure 7 shows the workload distribution. The NYC Citi Bike dataset has a lower insert rate of events per second, which can be observed in the distribution. With the lowest slide of 1s, NYC Citi Bike is spending around 38% of the time on inserts, whereas it is 85% in the DEBS12 dataset. The time spent on inserts increases with growing slide sizes.

Insert Latency. Figure 8 highlights the p95 latency of inserts. μWheel execute inserts with low latency and remain consistent over each window setup. FiBA, on the other hand, experiences higher tail latencies and worsens with growing window sizes in the write-intensive DEBS12 dataset.

Summary. We show using real-world data that inserts in most window setups dominate the execution compared to queries and advancing the watermark. Tree-based aggregate stores such as FiBA couple inserts and index maintenance together, which increases the write complexity. μWheel takes advantage of the fact that low watermarking is used in practice [3, 6, 11, 48] and decouples the write and read paths, and performs synchronization lazily when the low watermark is advanced. This separation enables μWheel to achieve lower insert latencies, improving the overall performance.

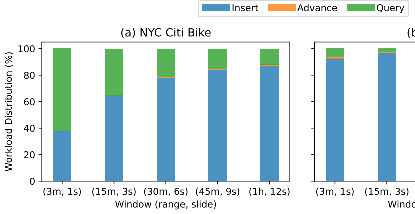

Figure 7: Workload Distribution - Low Ratio

4.2.3 Summary. The results show that μWheel is not penalized to the same degree as FiBA regarding varying out-of-orderness, write intensity, and number of concurrent window slides. μWheel outperforms the existing state-of-the-art by orders of magnitude under high write-intensity scenarios in combination with a high number of concurrent window slides. FiBA only outperforms μWheel when both the write intensity and the number of concurrent window slides are low. μWheel however has comparable performance under

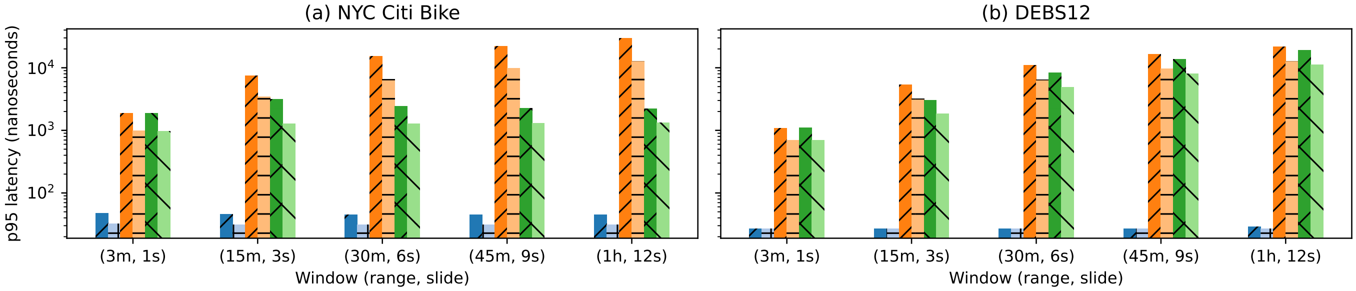

Figure 8: Streaming Window Aggregation (SUM) Insert Latency - Low Ratio

these circumstances while simultaneously organizing aggregates for other diverse analytical needs.

4.3 Offline Analytical Aggregation

We now study the analytical query performance of μWheel. Our evaluation compares μWheel to other embeddable query aggregation solutions, including both pre-aggregation and on-demand approaches. We include three versions of μWheel in the evaluation:

- μWheel: a wheel with optimization hints turned off (i.e., query planner simply picks the plan with the minimum amount of aggregate operations).
- μWheel-hints: a wheel with framework hints enabled (i.e., optimizer favors SIMD-centric aggregate plans, maximimizing contiguous memory access).
- μWheel-prefix: a wheel which supports prefix-sum queries on all wheels (i.e., optimizer prioritizes pre-aggregation reuse for invertible functions at the expense of additional space requirements).

We implement and use the same aggregator for all μWheel variants, configured to retain all wheel slots for the experiment. The aggregator is equipped with SIMD capabilities and we set the *SIMD Threshold* (sec. 3.4.4) to 15000, meaning that μWheel-hints favors single-wheel aggregations when the number of $\oplus$ calls is below this threshold.

Data. We synthetically generate our event stream data using a subset of fields from the schema of the NYC Taxi dataset [1]. Specifically, we include the dropoff timestamp and fare amount fields where the former is used for time filtering and the latter as a measure for analytical purposes. We store each variable as a 64-bit unsigned integer.

Baselines. We categorize our baseline approaches into on-demand aggregation and pre-aggregation methods:

BTree: A BTree implementation from the Rust standard library that indexes the raw data and computes aggregates on demand.

DuckDB: A state-of-the-art embedded OLAP DB [35] that is often used today for high-performance on-demand aggregation. Our implementation utilizes DuckDB (version 0.10.0) with Rust bindings[2]. After our own experimentation, we've chosen to operate DuckDB in disk mode for two primary reasons: 1) negligible performance loss compared to in-memory operation, and 2) disk-based storage's support for data compression. In addition, this is how the database is most commonly used.

FiBA: A specialized pre-aggregation data structure [39], for sliding window aggregation that also supports arbitrary time-range queries. We evaluate FiBA with varying fanout degrees: 2, 4, and 8.

Segment Tree: A representative state-of-the-art static data structure [8] optimising for pre-computed aggregations with no update support.

Metrics. We measure query execution latency across all systems. Note that with the exception of DuckDB, the systems employ statically compiled functions. DuckDB's reliance on SQL string parsing introduces a degree of variability. However, the impact of this parsing overhead is likely to be minor relative to DuckDB's overall execution time. In addition to latency, we measure storage consumption during experiments. For DuckDB, we utilize the PRAGMA database_size command to access the storage information. Space usage in the remaining systems is primarily calculated based on the size of stored aggregates, with timestamps included in systems where they are maintained.

Queries. We evaluate the systems using two types of data aggregation queries, categorized into queries Q1 and Q2.

Q1: Data Aggregation

```
select sum(fare_amount) from rides
```

Q2: Data Aggregation with Time Filter

```
select sum(fare_amount) from rides
where timestamp between ? and  ?
```

4.3.1 General Performance. In this section, we study the performance of the systems by executing queries Q1 and Q2.

Workload. We generate datasets containing one event per second, spanning two distinct time periods: one day and seven days respectively. The start date is defined as 2023-10-01 00:00:00. For each period, we execute 50,000 queries per query type, recording the p95 latency. The p95 latency metric allows for a focus on the performance experienced by the vast majority of queries. In query Q2, we apply time filters at varying granularity levels. For example, q2-seconds signifies that the smallest represented time unit is seconds (e.g., a filter from 12:15:30 to 13:20:15). We generate time filters

[2] https://github.com/duckdb/duckdb-rs

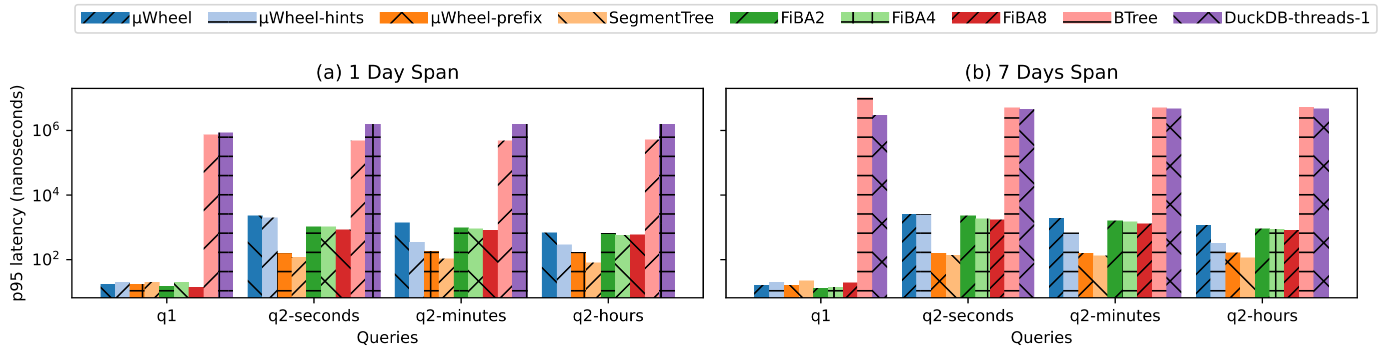

Figure 9: Offline Analytical Aggregation - P95 Query Latency

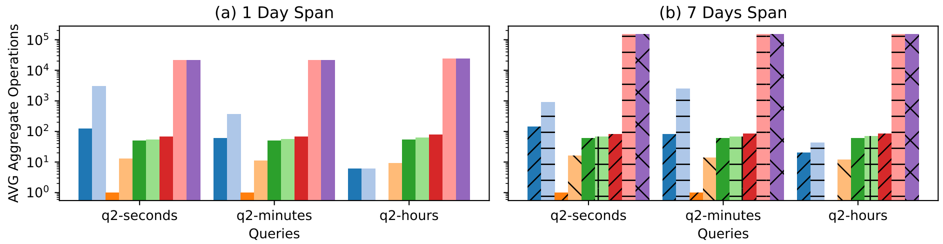

Figure 10: Average Operations for Time Filter Queries in Figure 9

with uniformly distributed start dates and durations. To achieve this, we first select a random starting point within the period and then determine a random duration, ensuring the end of the filter falls before or on the overall end date.

Results. Figure 9 shows the p95 query latency under both the one-day and seven-day span. Furthermore, Figure 10 highlights the average number of aggregate operations for time-filter queries. As expected, on-demand baselines exhibit significantly higher latency compared to pre-aggregating solutions. In Q1, pre-aggregation variants achieve sub-microsecond latencies, while BTree and DuckDB operate in the sub-millisecond range. The static Segment Tree performs best for time filter queries, with μWheel and FiBA variants demonstrating comparable performance. Despite requiring more aggregate operations than the standard μWheel (as observed in Figure 10), μWheel-hints achieves lower latencies. This performance gain stems from bypassing the overhead of constructing and executing complex combined aggregation plans. Instead, μWheel-hints favours directly accessing individual wheels and capitalizing on SIMD optimizations. Prefix-enabled μWheel offers the lowest latency among μWheel variants due to using a pre-computed prefix-sum array. DuckDB and BTree execute time filter queries with latencies exceeding 100 microseconds, while pre-aggregation solutions remain below 10 microseconds.

Summary. μWheel offers performance rivaling or exceeding specialized pre-aggregation indexes, while providing a far greater range of optimization possibilities. These optimizations include leveraging framework-specific hints during query planning for faster execution and using prefix-enabled wheels for low-latency queries with invertible aggregate functions. Another advantage is that μWheel exhibits consistent performance as time periods increase, unlike on-demand solutions that experience latency increases. When the workload is known apriori, μWheel significantly outperforms on-demand solutions. μWheel trades upfront materialization overhead for low-latency query execution, a tradeoff that makes it ideal for real-time applications where minimizing query latency is critical.

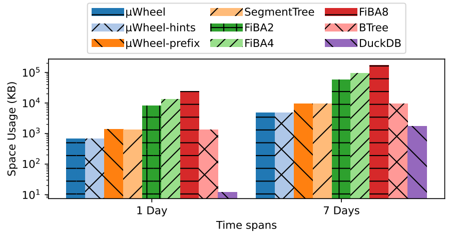

Figure 11: Offline Analytical Aggregation - Space Usage

4.3.2 *Space Usage.*

We now study the space usage of each system during the execution of each period.

Results. Figure 11 illustrates space usage, demonstrating that μWheel consistently requires the least space compared to all pre-aggregation systems across both periods. Notably, μWheel consumes 11.8x less space than the most space-efficient FiBA variant and 35x less than the best performing one. Furthermore, μWheel exhibits a smaller space footprint than an on-demand BTree indexing raw data with timestamps and aggregates. While DuckDB introduces an additional primary key field per table entry, its specialized columnar compression techniques result in high space efficiency. Consequently, DuckDB requires less space than μWheel in both time periods.

Summary. μWheel demonstrates high space efficiency, and is only outperformed by DuckDB which utilizes compression. Unlike pre-aggregation solutions like FiBA and Segment Tree, μWheel leverages the inherent hierarchical nature of time when creating higher-order aggregates. In adition, μWheel achieves significant space savings by using implicit timestamps, eliminating the need to store timestamps as required by aggregation trees like FiBA. Furthermore, μWheel has the potential to further reduce its space footprint by incorporating lightweight compression techniques [21].

4.3.3 Summary. The results demonstrate that μWheel is significantly better suited for workloads requiring real-time responsiveness compared to on-demand solutions like DuckDB. However, it is worth noting that DuckDB is a general-purpose system that supports a wider range of operations. Also, μWheel requires the trade-off of using a pre-defined aggregation function. Additionally, μWheel exhibits comparable performance to specialized aggregation tree indexes but with a substantially smaller memory footprint. Finally, we observe the advantage of μWheel 's query optimizer, which leverages framework-provided hints to generate more efficient execution plans.

5 RELATED WORK

Streaming Window Aggregation. Prior research addresses various challenges in streaming window aggregation. Solutions like FiBA [39] and CPiX [9] focus on handling out-of-order streams. Others, like Pairs [26], Panes [28], Cutty [14], Scotty [44], and LightSaber [42], target efficient execution of multiple concurrent window queries over streams. TAG [31], Disco [7], Desis [49], and NebulaStream [50] also explore streaming window aggregation within decentralized environments. Our work complements these systems by augmenting time-based out-of-order streaming window aggregation with query management support, bridging the gap between real-time insights and ad-hoc exploration of streaming data.

Queryable Stream State. The prospect of queryable state has been investigated as one of the emerging needs in the domain of stream processing technologies [12, 16]. The main challenge towards integrating an externally queriable (read-only) state contradicts with the design of modern stream processors which assume materialized stream inputs and outputs as the core interaction point with these systems, not the internal state. S-Query [46] attempts such an integration by enhancing relational ACID databases with stream processing capabilities on top of an existing transactional processing mechanism. S-query, however, does not support warehousing or temporal queries (e.g., arbitrary temporal windows) and is limited to regular OLTP ad-hoc queries on live and snapshot states. Stream processing systems today further support incremental window aggregation analytics through the use of specialised indexes [9, 29, 39, 43]. Yet, these solutions are specialized for computing temporal aggregates on recent data and do not support historical querying on varying time granularities. Once a window is triggered in a stream processor, the underlying state is evicted and unavailable for on-demand queries. StreamingCube [36] proposes using a special cubify operator that maintains a multi-dimensional data lattice. Their solution is tightly integrated within the stream processor, whereas μWheel is an embeddable aggregate management system that can be integrated within a stream processor or used as a standalone system.

6 CONCLUSIONS & FUTURE WORK

This paper proposes μWheel, an aggregate management system for streams and queries. μWheel addresses the lack of a unified data management architecture for online stream and offline analytical workloads, enhancing aggregate reuse, delivering highly up-to-date results for both, and minimizing resource utilization. μWheel can be used in either standalone mode or embedded within any stream processor. Our experimental analysis underscores μWheel's effectiveness for both streams and queries. μWheel's lazy aggregate synchronization, driven by low watermarks, significantly improves performance for high-volume stream workloads. Moreover, its wheel-centric query optimizer facilitates low-latency performance in ad-hoc query scenarios.

Future Work. Our future work on μWheel encompasses several key expansions. Firstly, we will introduce multi-key capabilities to broaden μWheel's analytical scope. Secondly, we'll explore lightweight compression techniques to optimize its space efficiency further. Finally, we plan to investigate a CRDT [37] variant of μWheel that can be merged and used to facilitate analytics in a decentralized environment.

7 ACKNOWLEDGEMENTS

This work has been supported by Vinnova (Grant No.: 2022-03036), the Swedish Foundation of Strategic Research (Grant No.: BD15-0006), and Wallenberg AI NEST (DataBound Computing). Finally, we thank the anonymous DEBS reviewers who provided feedback on how to improve the paper.

REFERENCES

[1] Tlc trip record data. https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page. (Accessed on 03/03/2023).

[2] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: Fault-tolerant stream processing at internet scale. In *VLDB*, 2013.

[3] T. Akidau, E. Begoli, S. Chernyak, F. Hueske, K. Knight, K. Knowles, D. Mills, and D. Sotolongo. Watermarks in stream processing systems: Semantics and comparative analysis of apache flink and google cloud dataflow. *Proc. VLDB Endow.*, 14(12):3135–3147, jul 2021.

[4] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, et al. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *In VLDB*, 2015.

[5] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia. Structured streaming: A declarative api for real-time applications in apache spark. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 601–613, New York, NY, USA, 2018. Association for Computing Machinery.

[6] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia. Structured streaming: A declarative api for real-time applications in apache spark. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 601–613, New York, NY, USA, 2018. ACM.

[7] L. Benson, P. M. Grulich, S. Zeuch, V. Markl, and T. Rabl. Disco: Efficient distributed window aggregation. In *Proceedings of the 23nd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*, pages 423–426. OpenProceedings.org, 2020.

[8] J. L. Bentley and J. B. Saxe. Decomposable searching problems i: Static-to-dynamic transformation. *J. Algorithms*, 1(4):301–358, 1980.

[9] S. Bou, H. Kitagawa, and T. Amagasa. Cpix: Real-time analytics over out-of-order data streams by incremental sliding-window aggregation. *IEEE Transactions on Knowledge and Data Engineering*, 34:5239–5250, 2021.

[10] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas. State management in Apache Flink: Consistent stateful distributed stream processing. *In VLDB*, 2017.

[11] P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl, and K. Tzoumas. Apache Flink: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin*, 2015.

[12] P. Carbone, M. Fragkoulis, V. Kalavri, and A. Katsifodimos. Beyond analytics: The evolution of stream processing systems. In *Proceedings of the 2020 ACM SIGMOD international conference on Management of data*, pages 2651–2658, 2020.

[13] P. Carbone, A. Katsifodimos, and S. Haridi. Stream window aggregation semantics and optimization., 2019.

[14] P. Carbone, J. Traub, A. Katsifodimos, S. Haridi, and V. Markl. Cutty: Aggregate sharing for user-defined windows. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management.* ACM, 2016.

[15] S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. *SIGMOD Rec.*, 26(1):65–74, mar 1997.

[16] M. Fragkoulis, P. Carbone, V. Kalavri, and A. Katsifodimos. A survey on the evolution of stream processing systems. *The VLDB Journal*, 33(2):507–541, nov 2023.

[17] Y. Fu and C. Soman. Real-time data infrastructure at uber. *CoRR*, abs/2104.00087, 2021.

[18] K. P. Gaffney, M. Prammer, L. Brasfield, D. R. Hipp, D. Kennedy, and J. M. Patel. Sqlite: Past, present, and future. *Proc. VLDB Endow.*, 15(12):3535–3547, sep 2022.

[19] S. Geffner, D. Agrawal, A. El Abbadi, and T. Smith. Relative prefix sums: an efficient approach for querying dynamic olap data cubes. In *Proceedings 15th International Conference on Data Engineering (Cat. No.99CB36337)*, pages 328–335, 1999.

[20] C. Gencer, M. Topolnik, V. Ďurina, E. Demirci, E. B. Kahveci, A. Gürbüz, O. Lukáš, J. Bartók, G. Gierlach, F. Hartman, U. Yılmaz, M. Doğan, M. Mandouh, M. Fragkoulis, and A. Katsifodimos. Hazelcast jet: low-latency stream processing at the 99.99th percentile. *Proc. VLDB Endow.*, 14(12):3110–3121, jul 2021.

[21] L. Heinzl, B. Hurdelhey, M. Boissier, M. Perscheid, and H. Plattner. Evaluating lightweight integer compression algorithms in column-oriented in-memory dbms. In *12th International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2021, Copenhagen, Denmark, August 16, 2021*, 8 2021.

[22] M. Hirzel, S. Schneider, and K. Tangwongsan. Sliding-window aggregation algorithms: Tutorial. In *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems*, DEBS '17, page 11–14, New York, NY, USA, 2017. Association for Computing Machinery.

[23] J.-F. Im, K. Gopalakrishna, S. Subramaniam, M. Shrivastava, A. Tumbde, X. Jiang, J. Dai, S. Lee, N. Pawar, J. Li, and R. Aringunram. Pinot: Realtime olap for 530 million users. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 583–594, New York, NY, USA, 2018. Association for Computing Machinery.

[24] Z. Jerzak, T. Heinze, M. Fehr, D. Gröber, R. Hartung, and N. Stojanovic. The debs 2012 grand challenge. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, page 393–398, New York, NY, USA, 2012. Association for Computing Machinery.

[25] A. Koliousis, M. Weidlich, R. Castro Fernandez, A. L. Wolf, P. Costa, and P. Pietzuch. Saber: Window-based hybrid stream processing for heterogeneous architectures. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 555–569, New York, NY, USA, 2016. Association for Computing Machinery.

[26] S. Krishnamurthy, C. Wu, and M. Franklin. On-the-fly sharing for streamed aggregation. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, page 623–634, New York, NY, USA, 2006. Association for Computing Machinery.

[27] D. Leijen, B. Zorn, and L. de Moura. Mimalloc: Free list sharding in action. Technical Report MSR-TR-2019-18, Microsoft, June 2019.

[28] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Rec.*, 34(1):39–44, mar 2005.

[29] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, page 311–322, New York, NY, USA, 2005. Association for Computing Machinery.

[30] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier. Out-of-order processing: A new architecture for high-performance stream systems. *In VLDB*, 2008.

[31] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):131–146, dec 2003.

[32] M. Meldrum, K. Segeljakt, L. Kroll, P. Carbone, C. Schulte, and S. Haridi. Arcon: Continuous and deep data stream analytics. In *Proceedings of Real-Time Business Intelligence and Analytics*, BIRTE 2019, New York, NY, USA, 2019. Association for Computing Machinery.

[33] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell. Samza: Stateful scalable stream processing at Linkedin. *In VLDB*, 2017.

[34] P. Pedreira, O. Erling, M. Basmanova, K. Wilfong, L. Sakka, K. Pai, W. He, and B. Chattopadhyay. Velox: meta's unified execution engine. *Proc. VLDB Endow.*, 15(12):3372–3384, aug 2022.

[35] M. Raasveldt and H. Mühleisen. Duckdb: an embeddable analytical database. In P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1981–1984. ACM, 2019.

[36] S. A. Shaikh and H. Kitagawa. Streamingcube: Seamless integration of stream processing and olap analysis. *IEEE Access*, 8:104632–104649, 2020.

[37] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, page 386–400, Berlin, Heidelberg, 2011. Springer-Verlag.

[38] K. Tangwongsan, M. Hirzel, and S. Schneider. Low-latency sliding-window aggregation in worst-case constant time. In *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems*, DEBS '17, page 66–77, New York, NY, USA, 2017. Association for Computing Machinery.

[39] K. Tangwongsan, M. Hirzel, and S. Schneider. Optimal and general out-of-order sliding-window aggregation. *Proc. VLDB Endow.*, 12(10):1167–1180, jun 2019.

[40] K. Tangwongsan, M. Hirzel, and S. Schneider. Out-of-order sliding-window aggregation with efficient bulk evictions and insertions. *Proc. VLDB Endow.*, 16(11):3227–3239, 2023.

[41] K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu. General incremental sliding-window aggregation. *Proc. VLDB Endow.*, 8(7):702–713, feb 2015.

[42] G. Theodorakis, A. Koliousis, P. Pietzuch, and H. Pirk. Lightsaber: Efficient window aggregation on multi-core processors. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2505–2521, New York, NY, USA, 2020. Association for Computing Machinery.

[43] J. Traub, P. Grulich, A. Cuéllar, S. Breß, A. Katsifodimos, T. Rabl, and V. Markl. Efficient window aggregation with general stream slicing. In M. Herschel, H. Galhardas, C. Binnig, Z. Kaoudi, I. Fundulaki, and B. Reinwald, editors, *Advances in Database Technology - EDBT 2019*, pages 97–108. OpenProceedings.org, 2019. 22nd International Conference on Extending Database Technology, EDBT 2019 ; Conference date: 26-03-2019 Through 29-03-2019.

[44] J. Traub, P. M. Grulich, A. R. Cuéllar, S. Breß, A. Katsifodimos, T. Rabl, and V. Markl. Efficient window aggregation with general stream slicing. In *22th International Conference on Extending Database Technology (EDBT)*, 2019.

[45] G. Varghese and T. Lauck. Hashed and hierarchical timing wheels: Data structures for the efficient implementation of a timer facility. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, page 25–38, New York, NY, USA, 1987. Association for Computing Machinery.

[46] J. Verheijde, V. Karakoidas, M. Fragkoulis, and A. Katsifodimos. S-QUERY: opening the black box of internal stream processor state. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*, pages 1314–1327. IEEE, 2022.

[47] J. Verwiebe, P. M. Grulich, J. Traub, and V. Markl. Survey of window types for aggregation in stream processing systems. *The VLDB Journal*, pages 1–27, 2023.

[48] Y. Wang and Z. Liu. A sneak peek at risingwave: a cloud-native streaming database. In *Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems*, DEBS '22, page 190–193, New York, NY, USA, 2022. Association for Computing Machinery.

[49] W. Yue, L. Benson, and T. Rabl. Desis: Efficient window aggregation in decentralized networks. 2023.

[50] S. Zeuch, A. Chaudhary, B. D. Monte, H. Gavriilidis, D. Giouroukis, P. M. Grulich, S. Breß, J. Traub, and V. Markl. The nebulastream platform for data and application management in the internet of things. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020.

[51] C. Zhang, R. Akbarinia, and F. Toumani. Efficient incremental computation of aggregations over sliding windows. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, KDD '21, page 2136–2144, New York, NY, USA, 2021. Association for Computing Machinery.