Distritbuted Stream Processing With Query Compilation

Master Project Report

Ivan Illic, Youri Kaminsky, Till Lehmann, Tobias Niedling

Hasso-Plattner-Institute for Digital Engineering
Data Engineering Systems Group
Prof.-Dr.-Helmert-Str. 2-3
14482 Potsdam, Deutschland

Term	Winter 2020/2021
Chair	Data Engineering Systems Group
Supervision	Prof. Dr. Tilmann Rabl & Dr. Pedro Silva

Potsdam, March 31st 2021

Contents

1 Introduction

As the value of new data generally decreases over time, analyzing it as early as possible becomes increasingly important. This is where stream processing frameworks come into play. In contrast to traditional batch processing, stream processing engines (SPEs) continuously process data as soon as it arrives from an unbounded data source (*stream*). Thereby stream processing queries have no finite runtime and steadily generate results.

To cope with ever-growing amounts of data, state-of-the-art stream processing engines like Apache Flink choose to scale-out and distribute the workload across multiple nodes. Thus, they achieve higher performance and very good scalability. On the downside, however, recent analyses expose their deficiency in sufficiently utilizing modern hardware capabilities (i.e. multi-core processors), as they often rely on the JVM as an overhead introducing abstraction layer [Zeuch et al., 2019].

Latest research explores increasingly sophisticated approaches to enhance query performance on single processing units. Compiling queries into strongly optimized, hardware-conscious executables has proven to surpass traditional approaches in terms of throughput and latency by orders of magnitude [Neumann, 2011, Grulich et al., 2020]. The benefit applies especially to long-running streaming queries, where the constant compilation overhead becomes negligible.

Although distribution as well as query compilation have been proven to increase the maximum throughput of SPEs, no research exists that analyses how these distinct optimization techniques perform in combination. Thus, to get the best of both worlds, it is necessary to evaluate how well the latest research achievement of compiled queries can be harnessed and integrated into the industry-standard approach of distributing streaming workloads on multi-node clusters.

In this work, we present a novel streaming engine prototype that generates compiled queries for streaming data and distributes the workload across multiple nodes. Our experimental evaluation shows that our proposed approach achieves 12.6× higher throughput than Apache Flink and scales super-linearly when distributing across multiple nodes on a modern 16-nodes high-performance cluster.

Our contributions are summarized as follows:

- We demonstrate the performance optimization potential that can be realized with hardware-aware executables by implementing a hardcoded C++ query, that is later used as an evaluation baseline.
- We add another engine to our set of baselines by implementing an iterator-style C++ streaming engine that is similar in concept to Apache Flink.
- We implement a novel streaming engine prototype that combines a) distribution over multiple nodes with b) query compilation for highly optimized stream processing.
- We implement a multi-threaded data generator capable of streaming data at specified rates and use it to evaluate our distributed, query-compiled engine thoroughly.
- We perform an extensive performance evaluation of our engine prototype that includes (amongst others) comparing its performance to Apache Flink and analysing its behaviour when scaling across an increasing number of nodes.

The rest of this report is structured as follows: In Chapter 2, we introduce relevant background information about stream processing, distributed computing and query compilation. In Chapter 3, we clarify the query workload design that we use for our experiments as well as the explicitly made design decisions and assumptions that are relevant with regards to our streaming engine implementations. In Chapter 4, we present the relevant implementation concepts and performance optimizations of our presented engine prototype. Following that, we present experimental results in Chapter 5. Finally, we discuss and conclude the viability of our presented SPE prototype in Chapter 6 and 7.

2 Background

2.1 Stream Processing

Stream processing refers to data processing on unbounded/infinite data sources (aka. data streams). Stream processing engines (SPEs) (or streaming engines) are processing engines designed explicitly for such input data. When processing data streams, a precise notion of time becomes necessary as a streaming engine's output depends on the employed time domain. Typically, stream processing distinguishes between two time domains: event time and processing time.

When utilizing event times, SPEs process tuples (or records) in the order that certain events occurred (i.e., the point in time of the event that led to the tuple's creation). Processing time is defined as the point in time at which a tuple *enters* the processing engine.

Employing the processing time-domain, therefore, entails less semantic meaning. Which one of these time domains is used solely depends on the specific use case at hand. While a stateless operator like filter and map would not need any time domain, stateful operators (i.e., aggregations, joins, etc.) are implemented by discretizing unbound data into finite windows (either time or count-based).

Common window types are fixed windows, sliding windows, and session windows. Fixed windows classify incoming tuples in separate windows of fixed temporal length.

Sliding windows do the same while allowing for overlap between multiple fixed windows: A sliding window of 5 seconds duration and a sliding interval of 1 second will result in a fixed window of 5 seconds duration, created each second. Thus, tuples are processed multiple times as part of different windows. Naturally, this allows for optimized implementations of a shared state.

Session windows have varying temporal lengths. They are typically used with the event time domain in mind and are defined by a timeout gap. The SPE terminates open windows as soon as no relevant records arrive for a given timeout period. In this way, records are grouped based on how related they are in terms of time.

2.2 Distributed Computing

Handling high-velocity data streams efficiently and with low latency poses a particular challenge. Therefore, many stream processing engines, namely Apache Spark, Apache Storm, and Apache Flink, utilize a scale-out strategy. By distributing the workload across many computing nodes (cluster), the query execution is parallelized [Zaharia et al., 2016, Toshniwal et al., 2014, Carbone et al., 2015]. Scale-out parallelism benefits from increased computing power, higher availability (fault tolerance), and better scalability (achievable with commodity hardware). However, the limited bandwidth of the node's network connection emerges as the main bottleneck.

When employing distribution, the SPE commonly partitions the data stream among multiple nodes based on a user-specified key. Each node processes its associated records and therefore stores its relevant state in the main memory while the nodes communicate with each other if necessary (shared-nothing-architecture). Nevertheless, the SPE has to handle the arrival of foreign-key-tuples.

One possible solution is a central load balancing unit, responsible for distributing the tuples by their key towards each node. However, this introduces the load balancing unit as a potential bottleneck.

Alternatively, decentralized communication patterns are suitable. For instance, all nodes can re-partition tuples to other nodes using an all-to-all communication pattern.

Efficient distributed streaming engines need to achieve even load balancing. Otherwise, the slowest node may restrain the system's overall performance. Consequently, a fair distribution of the key-ranges is crucial. Additionally, today's state-of-the-art SPEs come with the ability to re-shuffle key groups when re-scaling the number of nodes. This requires migration of relevant local state between nodes (partial aggregates, hash tables, etc.).

2.3 Query Compilation

Most state-of-the-art streaming engines often rely on the Java Virtual Machine (JVM) to achieve hardware abstraction. Unfortunately, recent research exposes their deficiency in properly utilizing modern hardware architectures (many-core and multicore CPUs, Caches, SIMD, etc.) to its fullest extent. The introduced processing overhead of JVM-based streaming engines includes scattered data object representations in main memory, lack of spatial locality, virtual function calls, frequent instruction mispredictions, and garbage collection [Zeuch et al., 2019, Neumann, 2011].

Hard-coded implementations achieve a speedup of multiple orders of magnitude, as they

specialize in a single query. Query-awareness allows optimizations like fewer boilerplate code, reduced function calls, tight loops, and increased data and instruction cache utilization. Unfortunately, creating a hand-written query inherently introduces increased implementation involvement.

To cope with this, Neumann, 2011 first introduced query compilation in databases - a technique of automatically generating highly optimized code that, once compiled, rivals the runtime performance of hand-written queries. Generating the code for a query and compiling it into an executable imposes a constant time overhead. Thus, the query compilation approach is most efficient for long-running queries, typically found in stream processing. Therefore, the query compilation approach is promising. In fact, Grulich et al., 2020 proved significant performance improvements compared to the traditional interpretation-based query execution used in state-of-the-art SPEs.

3 Design Decision

3.1 Workload Design

We define one query for all of our benchmarking experiments to compare the different streaming engine prototypes implemented. The workload is inspired by a typical in-app advertisements and purchases scenario in which one might want to assess the profit a particular type of ad achieved. We define two data streams: One contains monitored data about in-app advertisements. The other one consists of information about in-app purchases generated through users clicking on specific advertisements.

```
ADS(ad_id: Int, user_id: Int, cost: Double)
PURCHASES(purchase_id: Int, user_id: Int, ad_id: Int, value: Double)
```

We define the following query as our benchmarking workload. It essentially describes each advertisement's profit as the difference between the sum of costs and the total generated revenue through in-app purchases resulting from the same ad. Additionally, we filter out purchases with an id of zero to include a filter operator in our query. Semantically, this could indicate purchases that have been completed independently of any advertisements.

```
SELECT a.ad_id, p.sum_purchases - p.sum_costs
FROM
    (SELECT ad_id, SUM(value) as sum_purchases
        FROM PURCHASES GROUP BY ad_id) as p,
    (SELECT ad_id, SUM(cost) as sum_costs
        FROM ADS GROUP BY ad_id) as a
WHERE p.ad_id == a.ad_id AND p.ad_id != 0;
```

3.2 Assumptions

In the following, we explicitly state the assumptions and limitations of our streaming engine implementations within the context of our work. This project aims to analyze and evaluate

the performance benefits of combining query compilation and distribution in the stream processing domain. Thus, some functionalities of a production-ready system are omitted.

1. We focus on processing time windowing only.
2. We solely implement sliding windows.
3. We assume a fixed number of nodes to distribute across for the entire query runtime. Thus, we don't support re-scaling while executing a query.
4. We only support decomposable aggregation operations.
5. We do not ensure fault tolerance.

4 Implementation

4.1 Data Generator

Exactly benchmarking stream processing engines is a sophisticated task. Common mistakes result from an insufficient separation of the data generator and the system under test. Traditionally, message brokers like Apache Kafka are used in production-ready systems [Jay Kreps, 2011]. Message brokers store data of different sources (sensors, IoT, etc.) on disk and then allow streaming engines to pull the data. However, such setups can quickly induce a bottleneck for the whole streaming pipeline. Consequently, they threaten to limit the peak performance of the system under test (SUT). [Karimov et al., 2018].

To achieve fast data generation rates that can saturate any of our evaluated streaming engines, we generate data on the fly using a (distributed) in-memory approach similar to what Karimov et al., 2018 proposes. This allows us to perform benchmarks in which the bottleneck is either the network bandwidth between stream processing nodes or the streaming engine's capability to process the incoming data. We run a dedicated instance of our data generator on the same node for each node of the streaming engine. That way, we simulate a setting in which the network bandwidth between data sources and the streaming engine is never saturated. Nonetheless, since we use the TCP protocol for actual data transfers, we could run the data generation on separate machines.

To avoid any disk reads/writes, we keep every generated tuple that the streaming engine will eventually process in the main memory. Initial tests revealed that generating complete tuples on the fly during the benchmark did not yield sufficient throughput.

To overcome this issue, we adopted pre-generation of *all* n tuples in advance of the actual benchmarking experiment. Pre-generated tuples contain all values except for the event time. As soon as all n tuples are created, multiple threads iterate over the tuples and append event times to each tuple in a configurable data generation rate. This procedure simulates a natural occurrence of tuples at a fixed rate. Enabling multiple threads to calculate and attach event times independently improves the data generation throughput drastically (up to 700MB/s).

When the streaming engine has successfully parsed and processed its latest TCP receive

buffer of tuples, it pulls a new batch of data from the generator. It is crucial to append event times at a fixed rate, independently from the streaming engine's processing speed. Otherwise, we can not observe the backpressure of unprocessed tuples - a typical characteristic of oversaturated streaming engines.

To achieve the desired behaviour, we fill a separate TCP sending buffer of fixed size (up to 2MB) with complete tuples as soon as their event times are appended. Simultaneously, the data generator continues to append event times to the remaining tuples residing in the primary buffer. In terms of optimizing the TCP communication for high throughput, messages need to be sizable, so we set the upper limit for the TCP send- and receive-buffers to 2MB.

If the streaming engine processes tuples faster than the data generator produces them, the TCP sending buffer will be incomplete. However, the generator sends a buffer of tuples as soon as the engine requests it to keep the streaming engine's idle time as low as possible.

If, on the other hand, data is generated faster than the streaming engine can process it, the difference between the event timestamp and the point in time of a tuple being sent will steadily increase over time. We can observe this behaviour in the form of increased latency. The independence of the generated event times and the streaming engine's execution allows us to perform correct latency measurements.

4.2 Streaming Engines

Achieving meaningful experimental results requires comparing against relevant baselines. In our benchmark experiments, we use ① Apache Flink, a state-of-the-art SPE, as our first baseline. Additionally, we implement ② an iterator style query processing SPE in C++ as a second baseline. Contrary to Flink, which is written for the JVM, we implement our proposed SPE prototype in C++. Thus, we achieve higher comparability by including a second baseline that is written in the same programming language and that contains equivalent implementations of certain operator logic (e.g. the same hash-join algorithm) as our proposed, optimized approach. Furthermore, we test the upper-performance limits of a single node by implementing ③ a highly optimized, hard-coded solution that processes only our defined query from Section 3.1.

This fine-grained collection of baselines allows us to precisely evaluate the impact of the two optimization techniques (query compilation and distribution) employed in our prototype engine. Our proposed SPE prototype ④ generates nearly optimal code for a given user-defined query and can also distribute the streaming workload across multiple nodes.

The following table shows an overview of all the SPEs under evaluation:

#	engine & technique	purpose
①	Apache Flink (JVM)	baseline
②	Iterator style C++ engine	baseline
③	Hard-coded C++ query	baseline
④	Distributed, query-compiled C++ engine	evaluation

As depicted in the table above, our evaluation mainly focuses on the performance of engine ④ which we compare with our set of baselines. In the following subsections, we explain relevant implementation details and undertaken performance optimizations with regards to our proposed streaming engine prototype.

4.2.1 Query Compilation

Our query compilation implementation is based on the following architecture: A main pipeline builder class is responsible for generating the final query code out of a set of operators that the user may arrange to form the intended query. Each operator defines its computations as a string containing valid C++ code, while a comprehensive system ensures the availability of global declarations and consistent, collision-free variable naming. To reduce the overhead of copying local data structures from one operator scope to another, operators declare data structures that are going to be accessed by following operators in the pipeline globally. To achieve performance as close to the hard-coded solution as possible, we avoid function calls and place each operator's code into one tight loop, if possible. Thus, we construct a predictable branch layout and achieve very high data locality.

4.2.2 Task Parallelism

We introduce *pipeline breakers* as operators that can not be applied on single tuples (e.g. filters or maps) but instead rely on the context of multiple tuples. Pipeline breakers divide the query's operator pipeline into multiple sub-pipelines, each of which is terminated by a pipeline breaker. The engine ensures, that each sub-pipeline runs in its own thread, thus achieving task parallelism. Figure 4.1 illustrates the behaviour.

In our engine implementation, the windowing operator is the only actual pipeline breaker. However, both, the aggregation operator and the hash join operator, require a preceding windowing operator. Our task-parallel approach results in the following performance benefit: A given initial sub-pipeline that receives a data stream via TCP, parses the tuples and

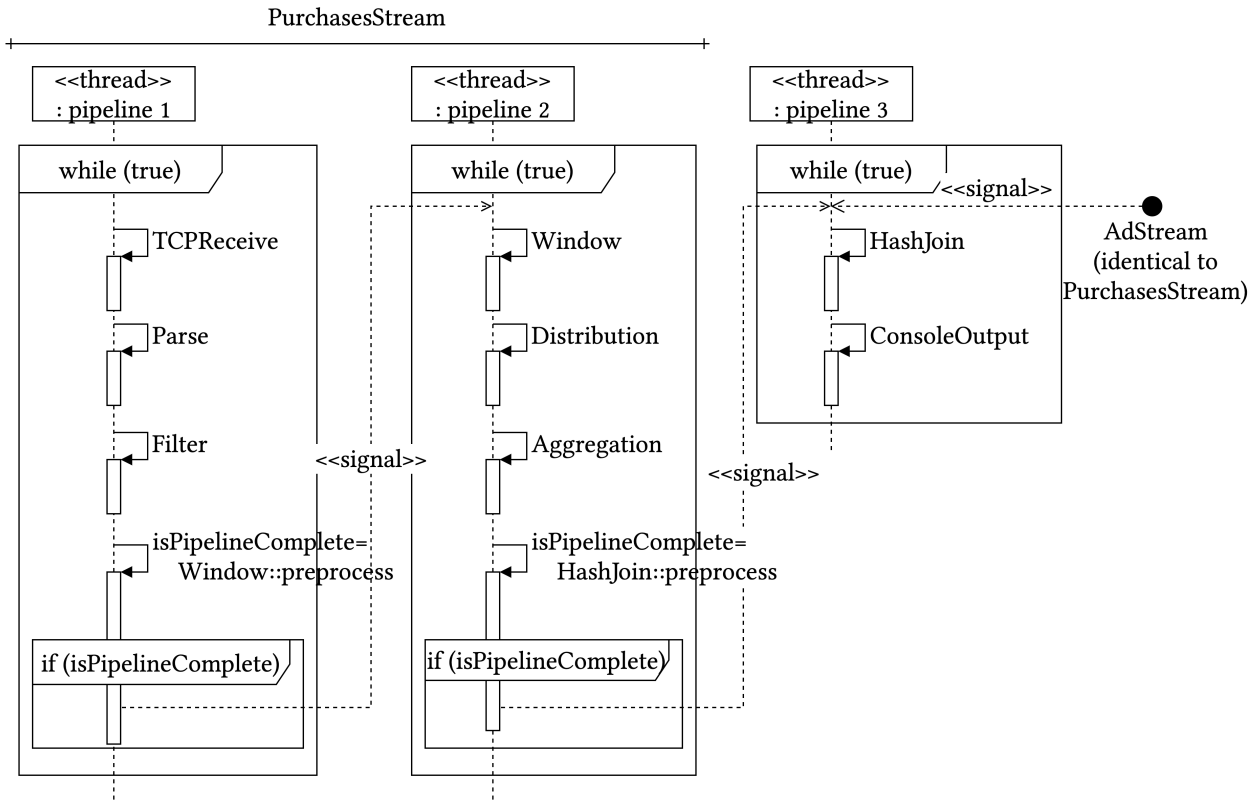

Figure 4.1: Visualization of the benchmark query execution behaviour as generated by the distributed, query-compiled engine.

performs a filter operation will continue to process without being stalled while the following second sub-pipeline can perform its tasks in parallel.

4.2.3 Distribution

As already mentioned, our query-compiled engine ④ generates queries that can be executed on multiple nodes. For this purpose, we assume that inter-node communication (transmitting tuples via network) is only necessary for windowing operations since all other operators are not pipeline-breakers. Non-pipeline breaking operators perform their computation tuple-wise, which can always be performed on an individual node only. However, within a windowing operator, tuples of every node are re-distributed to every other node with respect to a user-defined key. Given our defined query from Section 3.1, we distribute tuples based on the `ad_id` as follows: Tuple `t` belongs to node `n = t.ad_id modulo N` with `N` being the number of nodes that we distribute across. Figure 4.2 illustrates our partitioning scheme.

We implement network-based communication using the OpenMPI framework [OpenMPI, 2021]. We choose the `MPI_Alltoall` directive to perform the necessary exchange of tuples. Afterwards, each node stores all its associated tuples for the current window which enables

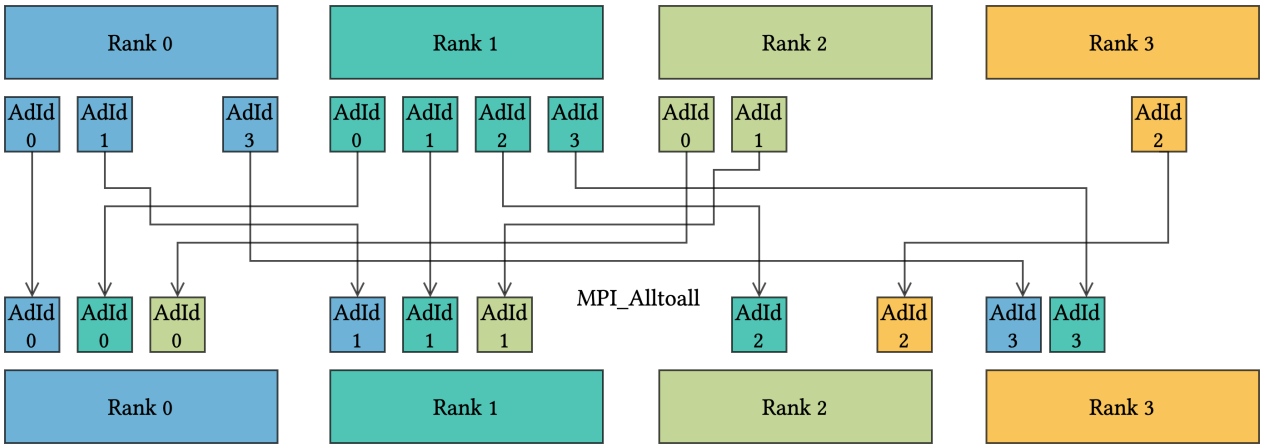

Figure 4.2: Partitioning scheme: Distributing tuples of our benchmark query based on the `ad_id` key.

following aggregation or join operations to be correct.

As a further performance optimization, we fuse the windowing functionality with the aggregation of tuples into one operator. This allows for computing preliminary partial aggregates. We thereby substantially reduce the network communication load as we only transfer one tuple aggregate per `ad_id` instead of many individual tuples. Because we implement the network communication within our task parallelism framework, the `MPI_Alltoall` calls run in a non-blocking fashion. As a result, our engine will continue to receive incoming tuples, parse, filter and pre-aggregate them, independent of the re-distribution.

5 Evaluation

5.1 Experimental Setup

We perform our benchmarks on the Fujitsu RX2530 M5 server, a 16-nodes cluster designed for high-performance computing tasks. Each node is equipped with 2x Intel Xeon Gold 5220S CPUs with 18 cores @2.70GHz and a total of 95 GB of RAM. The nodes are interconnected with 25 Gbit/s network bandwidth. The system runs Linux kernel version 5.4.0 and an Ubuntu 20.04 LTS. We use `numactl` to bind the process and memory allocation of our streaming engine to one NUMA node while the data generator runs on a different NUMA node. Thus, we avoid unwanted NUMA effects.

We measure the event time latency by adding an *event timestamp* in the data generator and adding a second timestamp at the end of the tuple processing, called *processed timestamp*. After every experiment, we compare these timestamps to determine the duration that the processing of a specific tuple took. Therefore, we can calculate an exact event time latency for each tuple and present the minimal event time latency for each window. The presented runtime is also determined by these artificial timestamps. For every run, we compare the lowest *event timestamp* with the highest *processed timestamp*. To calculate the average throughput, we divide the number of processed tuples by the measured runtime. Additionally, we count the number of tuples in each window to present a more fine grained throughput per window.

5.2 Experiments

In this section, we present the results of our experimental evaluation. Our performed experiments cover a wide range of performance aspects and benchmark metrics. First, we consider the runtime for all stream processing engines presented in section 4.2 for a fixed amount of data. Next, we focus on the query-compiled engine (④) since this is our main contribution. Therefore we evaluate the engine's achieved throughput and latency when scaling a) the workload and b) the number of nodes. Moreover, we examine the maximum sustainable throughput. These experiments are run with our benchmark query workload, defined in sec-

tion 3.1. Finally, we measure the engine's performance for varying key ranges to analyze how tuple ids' distribution impacts the runtime.

5.2.1 Comparing Stream Processing Engines

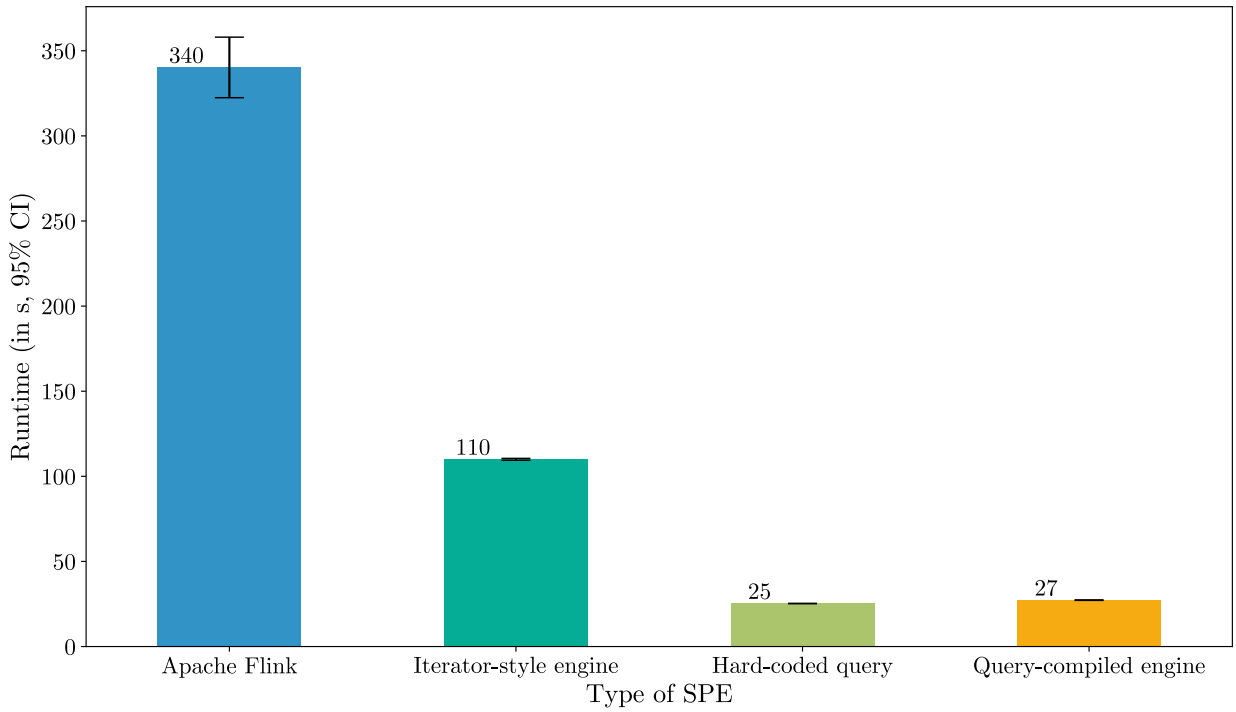

Figure 5.1: Runtime comparison of streaming engines 1–4 on a single node for 180M tuples

For this experiment, we run all stream processing engines discussed in section 4.2 on a single node. Each node processes the same query from section 3.1. The data is ingested through the TCP socket connection with the data generator, running on the same node. We run every experiment five times. As a result, we present the average and the 95% confidence interval for the runtime of these five runs. All engines terminate their execution as soon as they receive the last tuple from the data generator. Thus, we ignore the *ramp-off* phase as we do not continue processing the data still present in the last remaining window slides. Cutting off the ramp-off phase does not render our benchmark less conclusive as we are only interested in the actual *live* performance of our streaming engines for long-running queries.

In total, the data generator produces 180 million tuples for each of the two source streams. As described in section 4.1, the data generator produces and stores all required tuples upfront in main memory for higher throughput. Considering the additional memory footprint of the running streaming engine itself, we hit the main memory limit of a single node with this data amount. Since we want to conduct experiments running as long as possible, we choose 180

million tuples as the standard workload size.

Figure 5.1 shows the results of the experiment. We observe a significant difference between Apache Flink and all other stream processing systems that we developed in this project. Flink runs more than 3× longer to process the given workload compared to the second slowest system ②, and 13.6× longer than our highly optimized hardcoded query ③. There are obvious reasons for that difference. Flink is a fully-fledged streaming engine that implements a lot of features that we omit in our prototypes. These features include fault tolerance, monitoring, and many more. Additionally, Flink runs within the Java Virtual Machine. This additional abstraction layer introduces overhead compared to a hardware-aware compiled C++ program.

We observe further significant performance differences concerning the comparison of our C++ engines. Our hard-coded query and our query-compiled engine perform roughly 4× better than the iterator-style approach. As shown in previous research, avoiding function calls, implementing tight loops, a predictable branch layout, and extremely high data locality results in significantly higher throughput which explains why our engines ③ and ④ achieve the shown speedups.

Furthermore, we notice that the hardcoded query and the query-compiled engine's query have approximately the same performance (within 93%). We achieve this desirable result through two optimizations. Firstly, we reduce the overhead of copying local data structures through global declarations, which benefits all pipeline operators. Secondly, we place the code of each operator into one tight loop whenever possible. Tight loops mitigate overhead that function calls would otherwise introduce.

To keep up with its hardcoded counterpart, the query code generated by the query-compilation engine has to be as similar as possible to the hardcoded implementation. Therefore it is crucial to fine-tune the string concatenation mechanism of the query compilation without losing generality. Within our defined assumptions, we still ensure generality regarding the queries that the user can implement. Nevertheless, we show that, for our specified workload, it is possible to reach nearly the same performance as hardcoded queries achieve by employing query compilation.

5.2.2 Scaling Behaviour

In the following, we investigate the behaviour of our distributed, query-compiled engine ④ when scaling out in terms of a) processing nodes and b) data size.

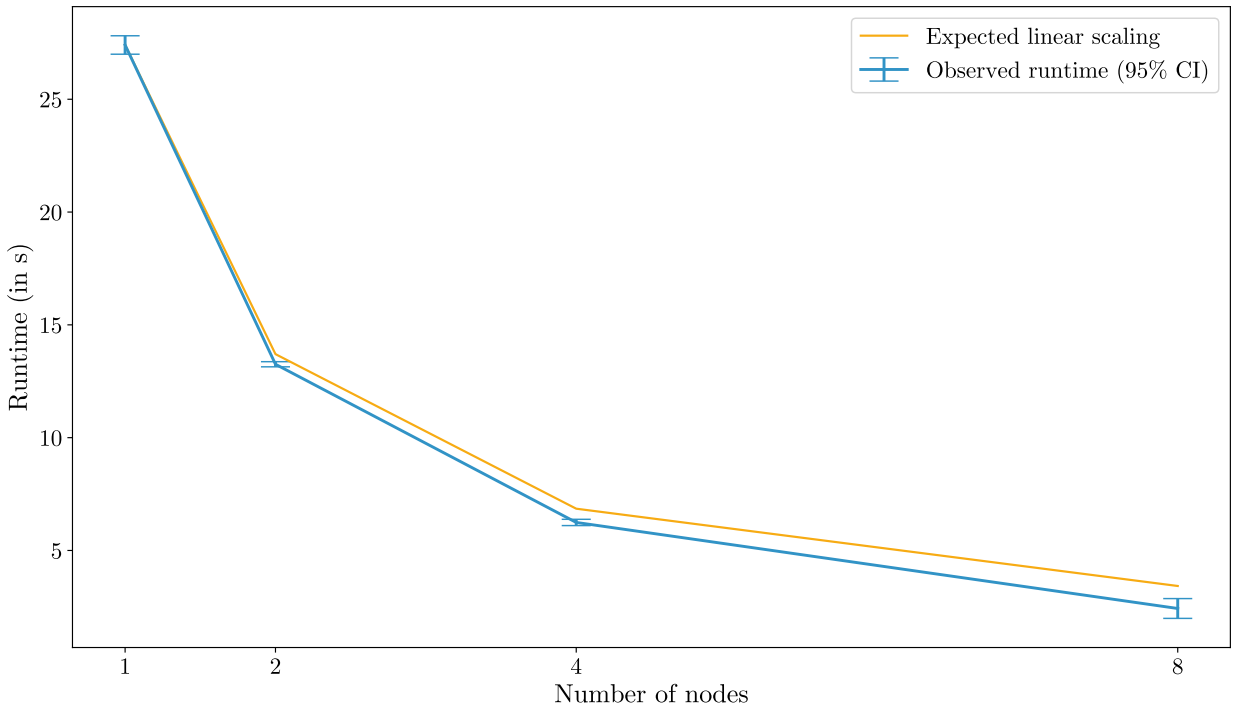

Figure 5.2: Runtime of our distributed, query-compiled engine for increasing numbers of nodes, with a constant total data size of 180M tuples

Scaling Number of Nodes: The first experiment examines the runtime development for an increasing number of computing nodes. For each number of nodes, we repeat the experiment five times. Each query execution processes a fixed total input of 180 million tuples per stream (independent of the number of nodes). We present the average as well as the 95% confidence interval of all runtimes. The data generator runs on each node and thus serves each local instance of the engine covering the whole range of `ad_id`s. However, as described in section 4.2.3, each node is only responsible for processing a segment of the key range. Tuples outside of this segment are redistributed to their respective node. In other words, with `N` nodes, each instance of the generator outputs `180/N` million tuples. Each node outputs its processed, final tuple results locally.

We expect the runtime to decrease approximately linearly depending on the number of nodes used. On the one hand, additional network transfer of data might introduce some overhead. On the other hand, splitting and thus shrinking the key range among nodes can reduce the complexity of aggregation and join operations. Smaller key ranges allow for tighter hash tables and, thus, increased data locality. Figure 5.2 presents the results of the experiment. We can observe a slighly super-linear speedup, that is the speedup is even higher than the linear reference (blue line).

We conclude that the reduced computational complexity overshadows the overhead introduced by network transfer in this experiment. The network traffic implied by our query workload is quite low. Since we transmit partial aggregates, each node only needs to transmit one tuple per `ad_id` per stream to another node. In this experiment, the `ad_id` values range from 0 to 1000. A tuple is not larger than 50 Byte. For two streams, this sums up to less than 100 KiB of data to transfer per second for each node. However, the much more important reason is that distribution reduces the processing overhead of the final aggregation and join of a single node since each node performs these operations only on a `1/n`-th of the whole key range.

Scaling Data Size:

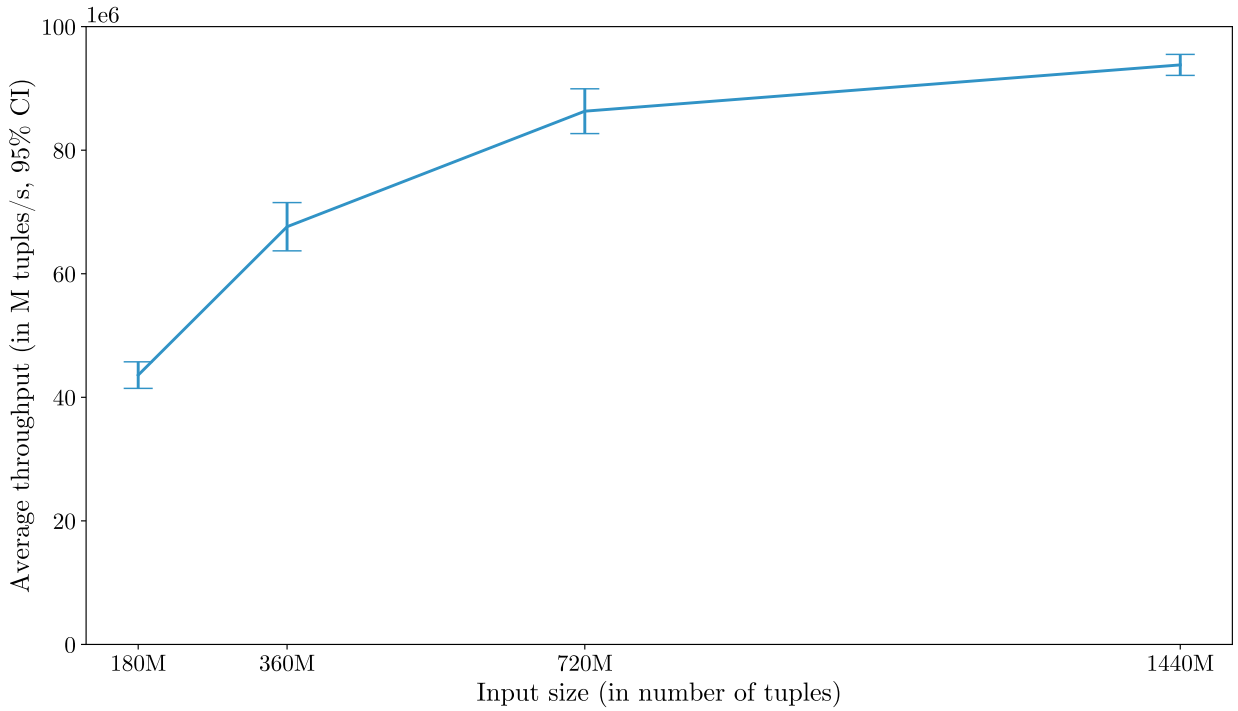

Figure 5.3: Throughput of our distributed, query-compiled engine for increasing input sizes on 8 nodes

In the second experiment, shown in Figure 5.3, we analyze the achieved throughput of engine ④ when scaling the number of tuples. Here, we always process using 8 nodes. At maximum, the total number of tuples reaches 1.44 billion tuples per stream.

Interestingly, we observe that processing a total of 180M tuples on eight nodes happens so fast that our streaming engine does not yet reach the maximum possible throughput. Our engine finishes execution before the *warm-up* phase of filling the first five window slides is fully completed. As soon as the fifth window is completed, each parsed tuple is counted

five times - once for all five windows that contain the tuple. Thus the overhead introduced through parsing is reduced in relation to the generated output.

With increasing numbers of tuples, we see a steady increase in throughput as the proportion of the *warm-up* period shrinks compared to the total execution time. For the most extensive data set, the average throughput peaks at over 90 million tuples per window, indicating which throughput our engine achieves for an actually unbounded data source.

5.2.3 Sustainable Throughput

We run all previous experiments with the maximum possible data generation rate, i.e., the data generator was not throttled in any way. Of course, this is not a realistic use case for a streaming engine since it leads to steadily increasing event-time latencies and overflowing buffers for long-running queries. Therefore, in the following experiment, we determine the maximum sustainable throughput empirically using different fixed data generation rates. The throughput is considered to be sustainable if the event time latency stays constant and relatively low during execution.

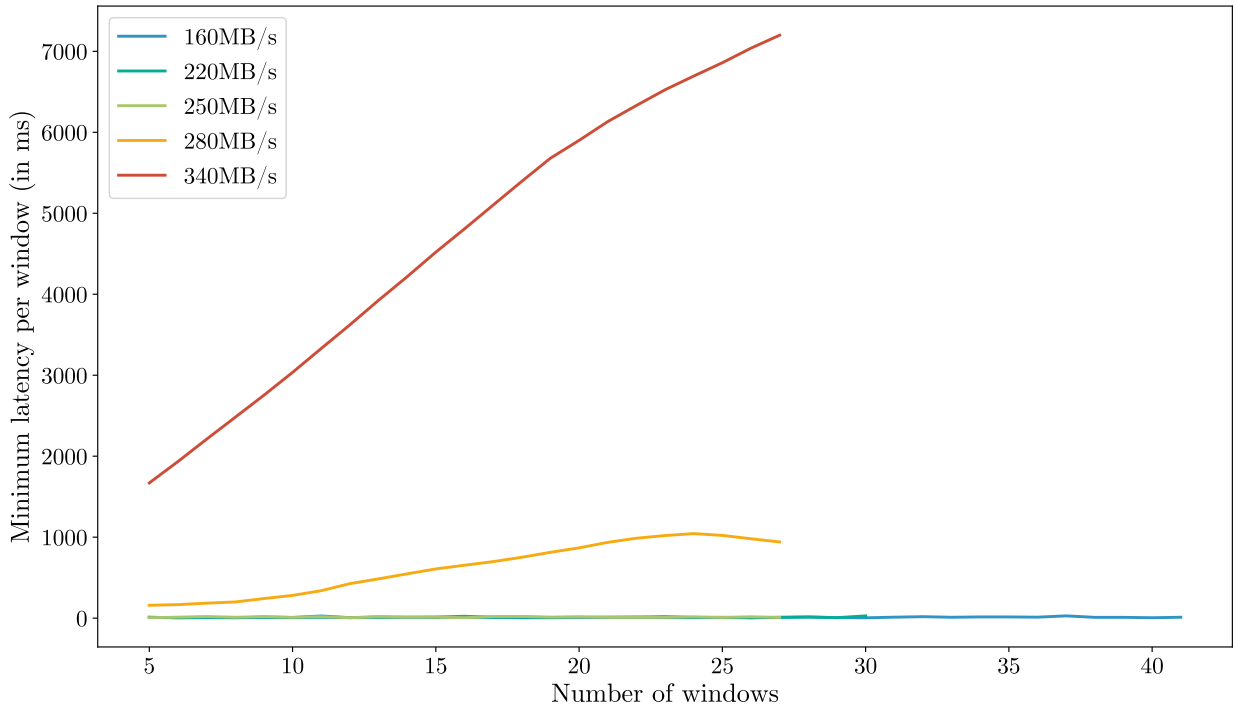

Figure 5.4: Event time latency of our distributed, query-compiled engine for varying data generation rates on a single node

Figure 5.4 visualizes the event time latency development as processing time progresses.

We calculate the event time latency of a window from the last tuple included in the window, i.e., the one with the highest event time. We observe that a data rate of 340 MB/s is not sustainable as the event time increases drastically over time. In this experiment, we neglected the first five windows, as they represent the *warm-up* phase but not the engine's long-time behaviour. Although the query execution finishes first for 340 MB/s, (after 20 seconds, processing 20 windows), the latency grows up to more than 6 seconds, which is too high for most stream processing applications.

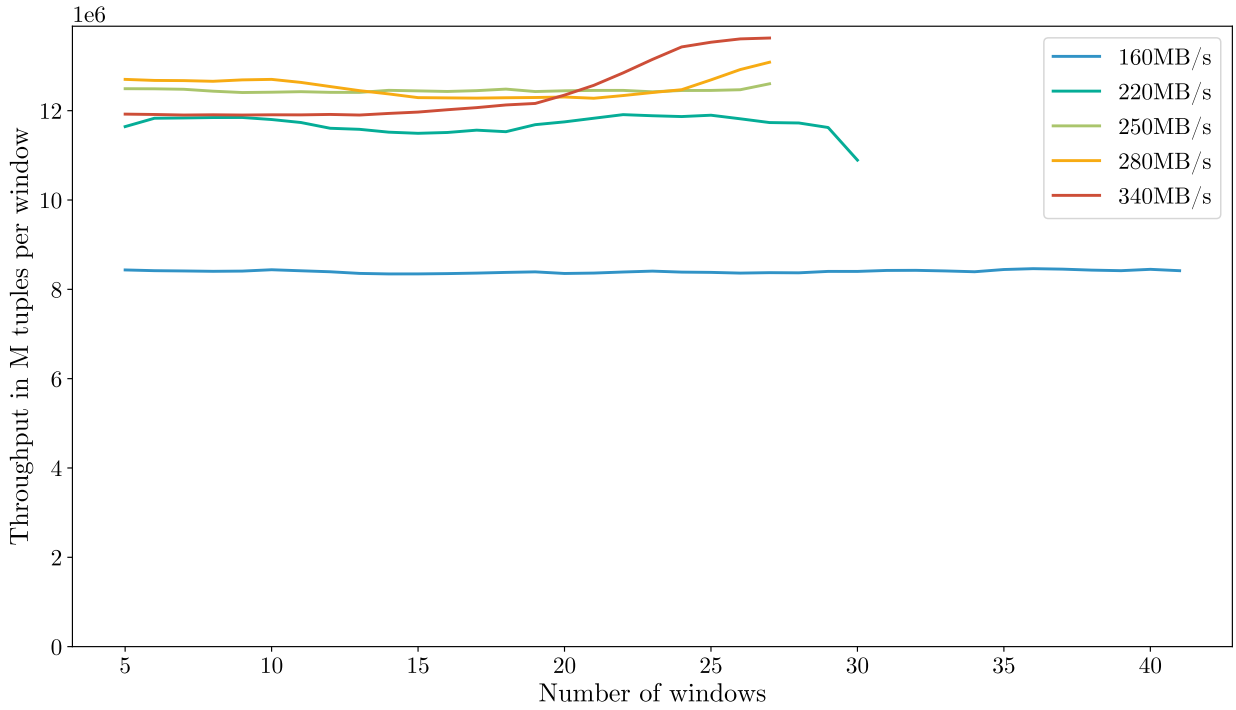

Figure 5.5: Throughput of our distributed, query-compiled engine for varying data generation rates on a single node

All other runs show a more promising event time latency behaviour. While a data generation rate of 280 MB/s still appears unsustainable, we achieve sustainable throughput for data rates between 220 MB/s and 250 MB/s. For all the data generation rates above 220 MB/s, we observe that the total query runtime only varies in an insignificantly small range. In all of these cases, the engine processes at near maximum throughput. This indicates that these processing rates hit the processing limit of our distributed, query-compiled engine ④. In contrast, the much longer query execution time for a data rate of 160 MB/s shows that the engine is not utilized at full capacity as it processes its current tuples faster than the new tuples arrive.

This observation gets even more apparent when comparing the maximum achieved throughput for all data generation rates, shown in Figure 5.5. The sustainable runs with data rates of 220 MB/s to 250 MB/s achieve a relatively constant throughput of nearly 12 million tuples per window. On the other hand, the non-sustainable runs with up to 340 MB/s do not achieve significantly higher throughput values.

5.2.4 Increasing Key Range

In all previous experiments, we use a default key range of 1000 `ad_id`s. In this final experiment, we test our engine's capability to handle more *difficult* data sets by including wider key ranges. The key range has a significant impact on the processing load of the query because of the following reasons:

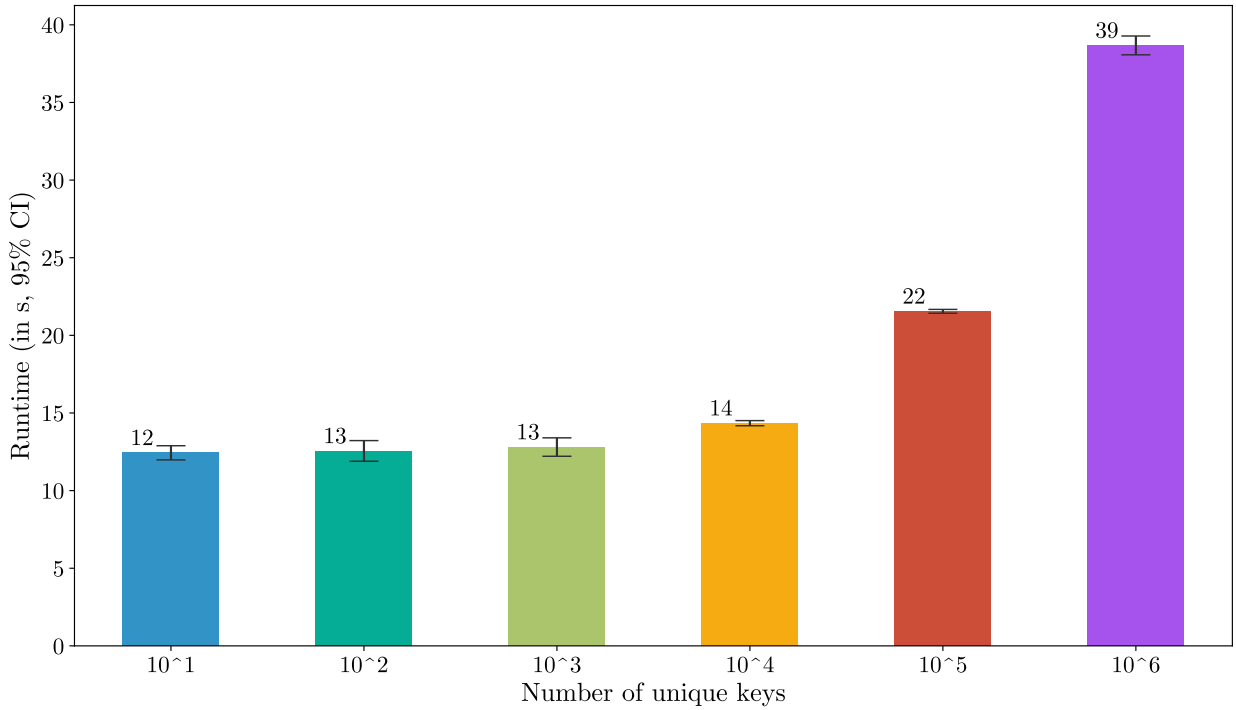

Figure 5.6: Runtime of our distributed, query-compiled engine for varying key ranges on 8 nodes

- The partial aggregation in advance to the tuple re-partitioning over network is done by key, resulting in one map entry per key. A higher key range results in more data entries in our map data structure.

- When a window closes, each node needs to transfer `(n-1)/n` of all tuples to other nodes, resulting in much more network traffic for larger key ranges.
- The final join operation between the two streams requires more hash table entries for larger key ranges.
- There is one line of output for each key in a window. Thus, a larger key range results in more disk I/O.

Figure 5.6 shows the runtime for 90 million tuples per node per stream, for a total of 720 million tuples per stream. As expected, the runtime increases for larger key ranges with 3.25× longer average runtimes for a key range of 10^6 compared to the lowest key range of only 10 different key values. Therefore, we conclude that finding an adequate distribution key is a crucial requirement for the efficient execution of distributed streaming queries.

6 Discussion

In this section, we conclude and discuss the results of our experimental evaluation from Chapter 5. In line with related research, we observe significant performance improvements when comparing our query-compiled engine with Apache Flink: On a single node, we achieve 12.6× higher throughput for a typical stream processing workload that includes filter, aggregation, and join operations in one query. Furthermore, we observe that our engine's generated and compiled query code achieves near-optimal performance as the evaluated runtime is very close to the performance of our carefully optimized, hardcoded baseline query (within 93%). Thus, our stream processing engine's compiled query substantially utilizes the hardware capabilities of multi-core processors.

Furthermore, we evaluate how our engine scales when distributing the workload across an increasing number of compute nodes. We observe very competitive results in this regard as the throughput scales even super-linearly due to our small network communication overhead and the reduced processing load that each node needs to handle locally. We keep the network overhead small by locally computing partial aggregates before performing the re-partitioning via the network. Additionally, by implementing task parallelism and utilizing the multi-core architecture of the compute nodes, each running instance continues processing incoming tuples locally, independently of the network communication of finished windows.

However, we do not compare the performance of Apache Flink with our engine's execution when distributing the query across multiple nodes. Thus, we can not definitively conclude about how the scaling behaviour of our engine differs from that of Flink. Such measurement results would further strengthen the case that our approach outperforms state-of-the-art SPEs by orders of magnitude.

Nonetheless, we notice that evaluating further workloads (i.e., different user-defined queries) becomes necessary to test the performance for use cases in which certain optimizations might not be possible. For example, computing partial aggregates before re-partitioning to reduce the network transfer cost is not possible when the user does not specify any aggregation operators. It remains interesting to see how our prototype performs for a broader set of queries and which optimizations can be implemented in those cases.

7 Conclusion

In this work, we presented a novel stream processing engine prototype that combines query compilation and distribution as two effective performance optimizations. Alongside our proposed engine prototype, we implemented a hardcoded C++ query and an iterator-style C++ streaming engine. We used both of these engines, combined with the state-of-the-art SPE Apache Flink, as our profound set of baselines that allowed us to perform an extensive and comparable evaluation of our distributed, query-compiled engine prototype. Our evaluation is additionally deepened by contributing a data generator implementation capable of streaming data at a configurable frequency. This allowed us to benchmark our engine's performance under different stream loads (e.g., reaching sustainable throughput as well as oversaturating our engine for high-speed data rates).

We conclude that firstly, combining query compilation and distribution is practically feasible and, secondly, the performance benefits from a well-optimized query-compiled *and* distributed SPE are significant. Our engine prototype outperforms state-of-the-art SPE Apache Flink on a single node and shows competitive results when scaling across multiple nodes. Even though we present an SPE prototype rather than a fully-fledged application in this work, our observed evaluation results indicate that a combination of query compilation and distribution is a highly promising approach for real-world workloads.

Regarding future work, we find that evaluating our engine's performance for a broader range of user-defined queries becomes a necessary next step as discussed in Chapter 6. Another possible extension could tackle reducing our assumptions by adding more features, e.g. implementing event time windowing, and thus moving away from a prototype implementation towards a more complete streaming engine.

Bibliography

Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., & Tzoumas, K. (2015). Apache flink: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin*, *38*. http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-198940

Grulich, P. M., Sebastian, B., Zeuch, S., Traub, J., Bleichert, J. v., Chen, Z., Rabl, T., & Markl, V. (2020). Grizzly: Efficient stream processing through adaptive query compilation, 2487–2503. https://doi.org/10.1145/3318464.3389739

Jay Kreps, J. R., Neha Narkhede. (2011). Kafka : A distributed messaging system for log processing. *NetDB*, 1–7.

Karimov, J., Rabl, T., Katsifodimos, A., Samarev, R., Heiskanen, H., & Markl, V. (2018). Benchmarking distributed stream data processing systems, 1507–1518. https://doi.org/10.1109/ICDE.2018.00169

Neumann, T. (2011). Efficiently compiling efficient query plans for modern hardware. *4*(9), 539–550. https://doi.org/10.14778/2002938.2002940

OpenMPI. (2021). Openmpi: Open source high performance computing. https://www.open-mpi.org/

Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J. M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., Bhagat, N., Mittal, S., & Ryaboy, D. (2014). Storm@twitter. *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, 147–156. https://doi.org/10.1145/2588555.2595641

Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., Ghodsi, A., Gonzalez, J., Shenker, S., & Stoica, I. (2016). Apache spark: A unified engine for big data processing. *Commun. ACM*, *59*(11), 56–65. https://doi.org/10.1145/2934664

Zeuch, S., Monte, B. D., Karimov, J., Lutz, C., Renz, M., Traub, J., Breß, S., Rabl, T., & Markl, V. (2019). Analyzing efficient stream processing on modern hardware. *Proc. VLDB Endow.*, *12*(5), 516–530. https://doi.org/10.14778/3303753.3303758