Distributed Stream Processing with Query Compilation

Ivan Ilic, Till Lehmann, Tobias Niedling, Youri Kaminsky

26.04.2021

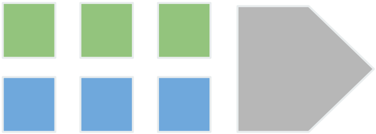
+ Usability

+ Efficiency

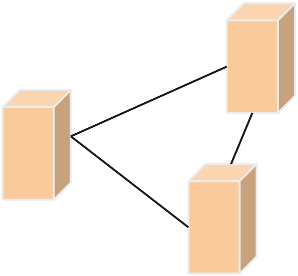
Outline

1. Technical Background
2. Workload Design
3. Implementation
4. Evaluation

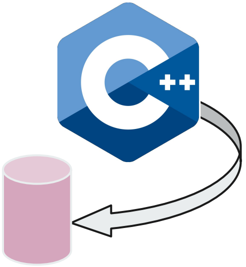
Technical Background



Stream Processing



Distributed Computing



Query Compilation

Assumptions

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.

Workload Design

Sample Query

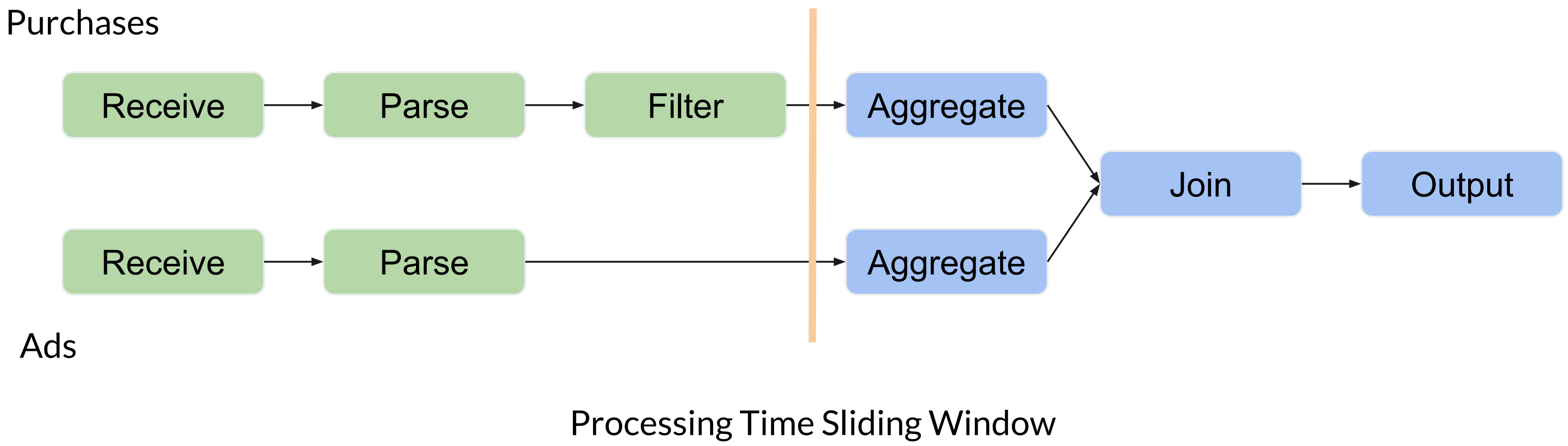
Stream 1:

```
ADS(
      ad_id: Int,
      user_id: Int,
      cost: Double
)
```

Stream 2:

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

Sample Query



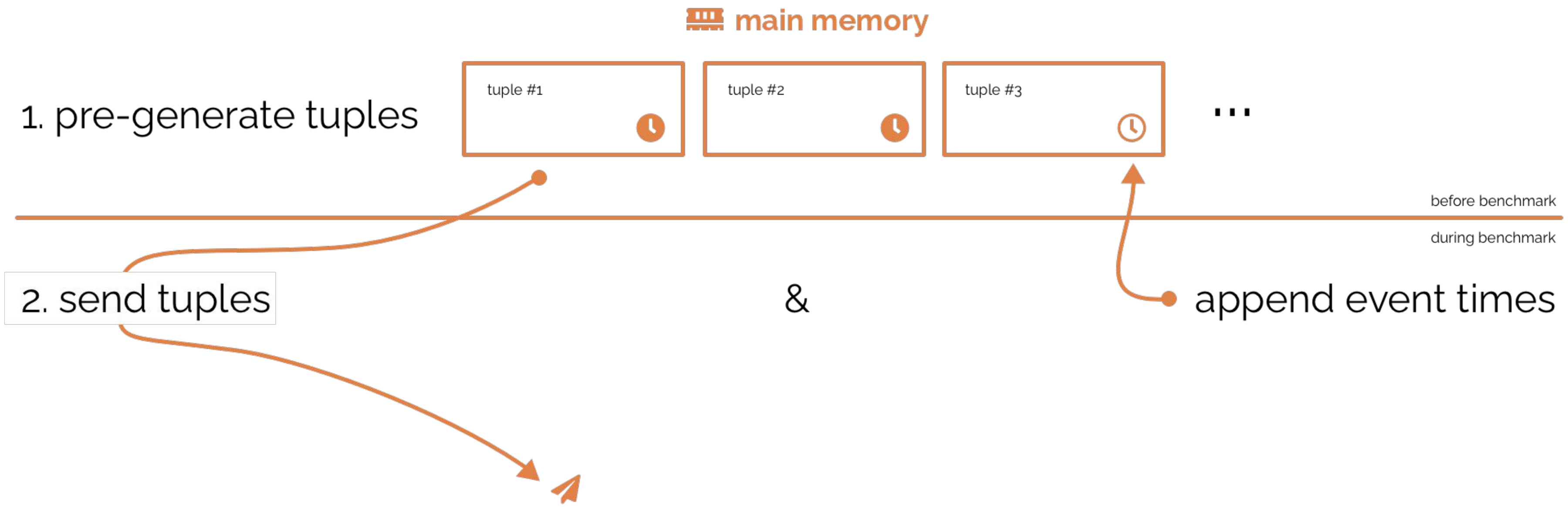
Sample Query

```
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;
```

Implementation

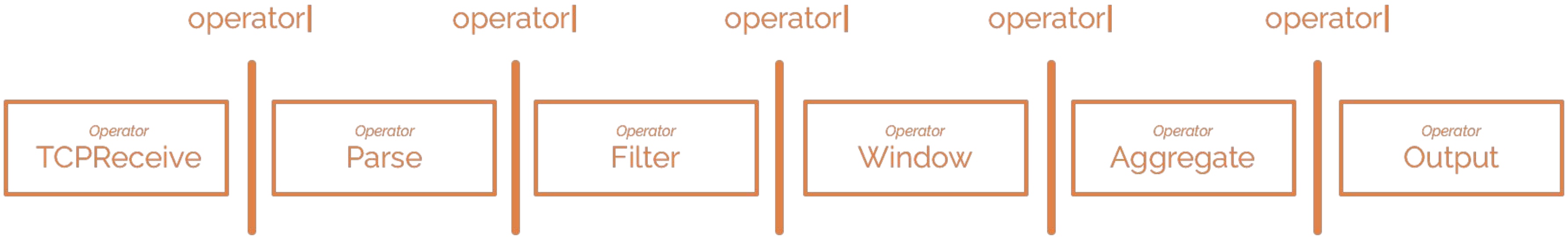
Data Generator



Streaming Engines

#	engine	purpose	expectation
1	query implemented into Apache Flink (JVM)	baseline	lower bound
2	query implemented into an iterator style C++ engine	baseline	lower bound
3	highly optimized hardcoded C++ query implementation	baseline	upper bound
4	C++ engine generating a distributed, compiled C++ query	*evaluation*	(best engine ever!)

Query compilation



Query compilation

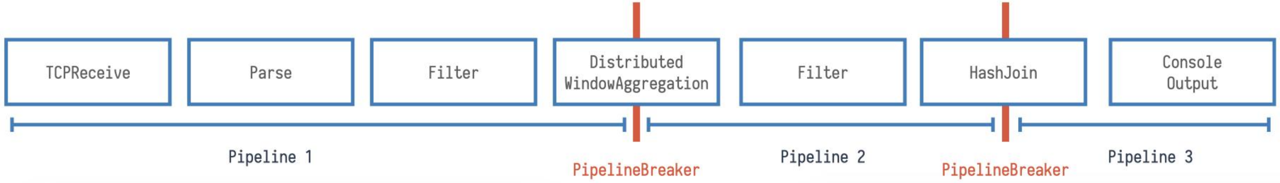
```
struct CheckoutTuple {
  std::shared_ptr<_Datatype> purchaseId,
    userId,
    adId,
    value,
    eventTime,
    processingTime;
};

CheckoutTuple checkoutTuple{
    POD("int", "purchaseId"),
    POD("int", "userId"),
    POD("int", "adId"),
    POD("double", "value"),
    POD("uint64_t", "eventTime"),
    POD("uint64_t", "processingTime")
  };

auto checkoutPipeline =
    TCPReceiver(12346)
      | Parse(Schema(std::vector{
            checkoutTuple.purchaseId,
            checkoutTuple.userId,
            checkoutTuple.adId,
            checkoutTuple.value,
            checkoutTuple.eventTime}))
      | Filter(checkoutTuple.adId, " == 0")
      | AppendTimestamp(checkoutTuple.processingTime)
      | DistributedWindowAggregation(1000, 5, aggregatedCheckoutTuple);
```

Hasso Plattner Institut
Digital Engineering · Universität Potsdam

Task parallelism



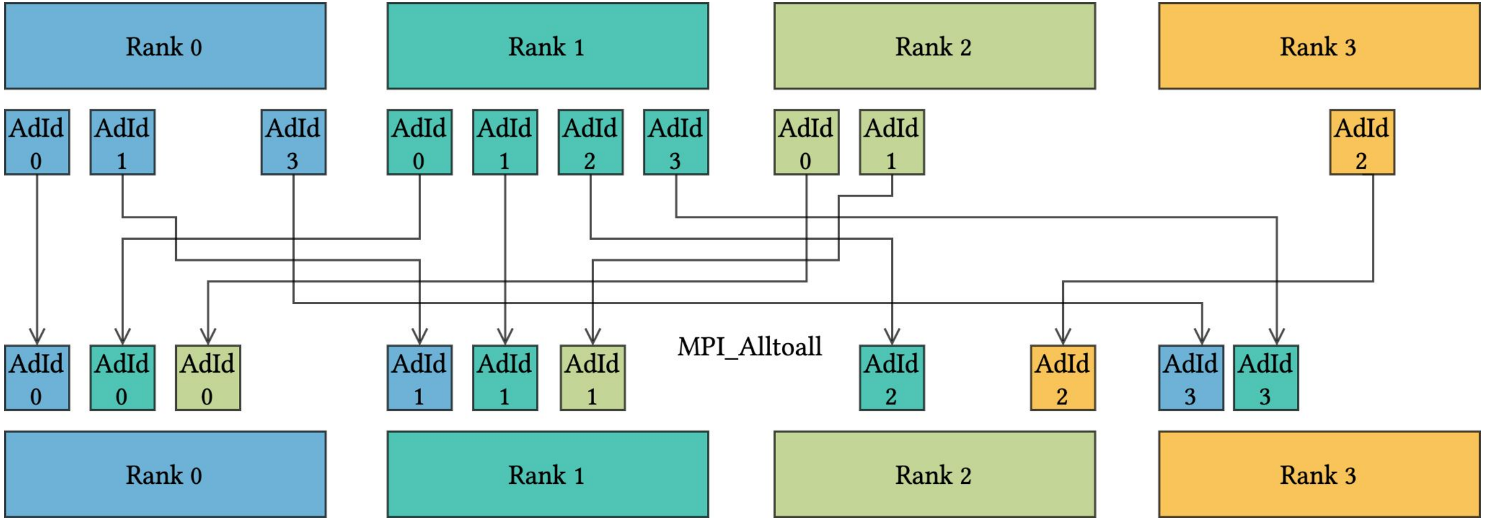

```
thread pipeline1([]{
    while (true) {
        TCPReceive();
        Parse();
        Filter();
        if (DistributedWindowAggregation::preprocess())
            semaphore1.signal();
    }
});
```

```
thread pipeline2([]{
    while (true) {
        semaphore1.wait();
        DistributedWindowAggregation();
        Filter();
        if (HashJoin::preprocess())
            semaphore2.signal();
    }
});
```

```
thread pipeline3([]{
    while (true) {
        semaphore2.wait();
        semaphore3.wait(); // from other Stream
        HashJoin();
        ConsoleOutput();
    }
});
```

Distribution

Distribution



Evaluation

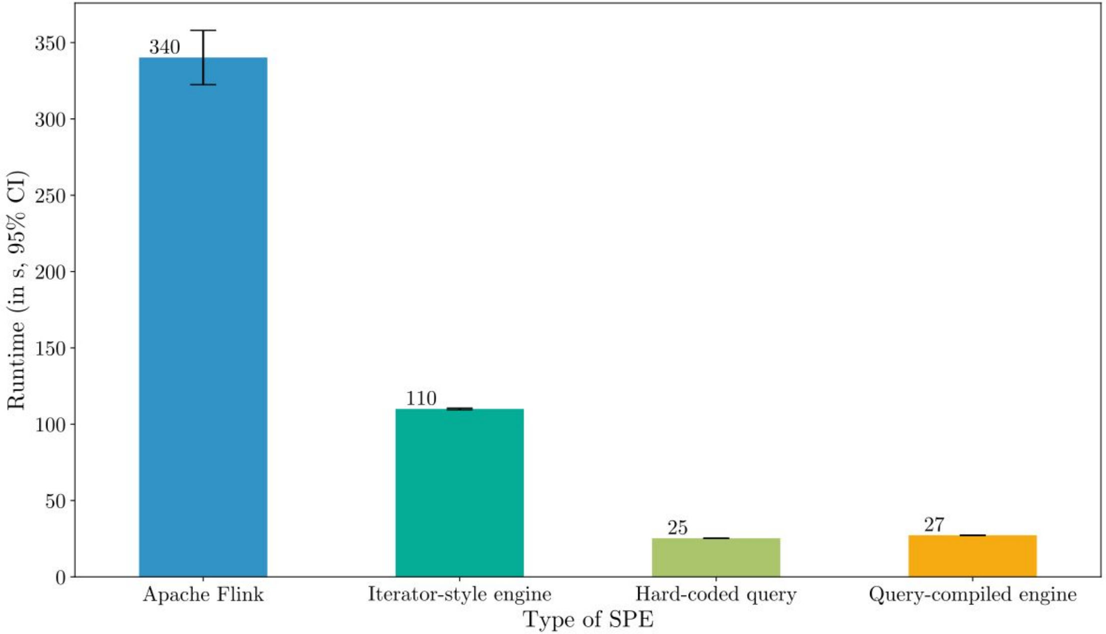
Experimental Setup

- 16 nodes cluster of Score Lab
  - 2x Intel Xeon Gold 5220S CPU
  - 95 GB RAM
  - 25 Gbit/s Ethernet networking

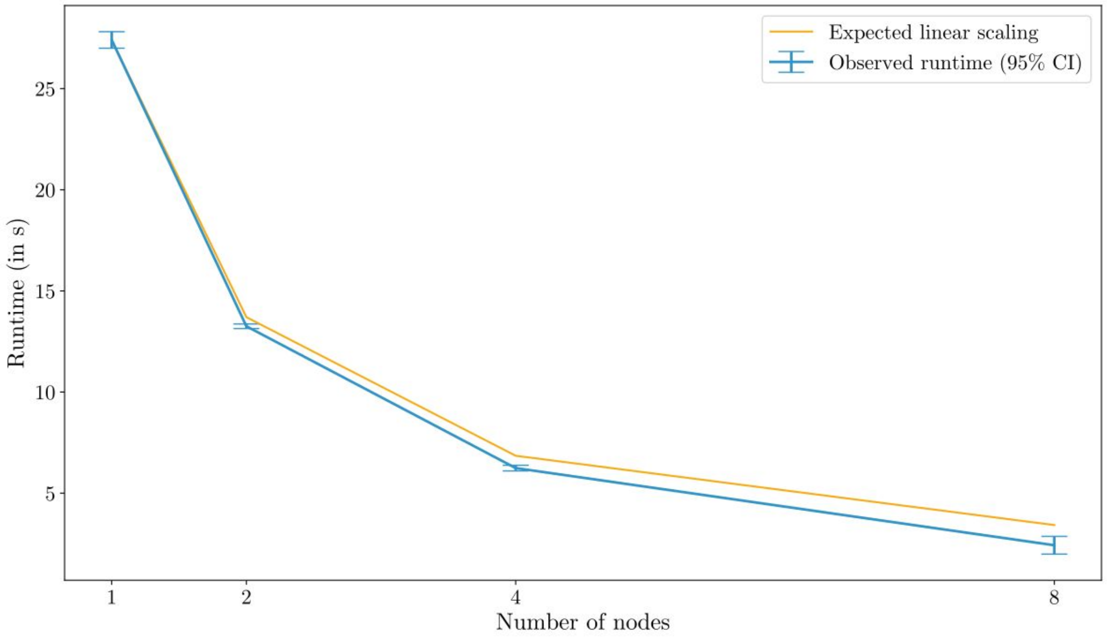
- numactl to bind process and memory allocation
  - Different NUMA nodes for generator and SPE

- Each experiment conducted 5 times

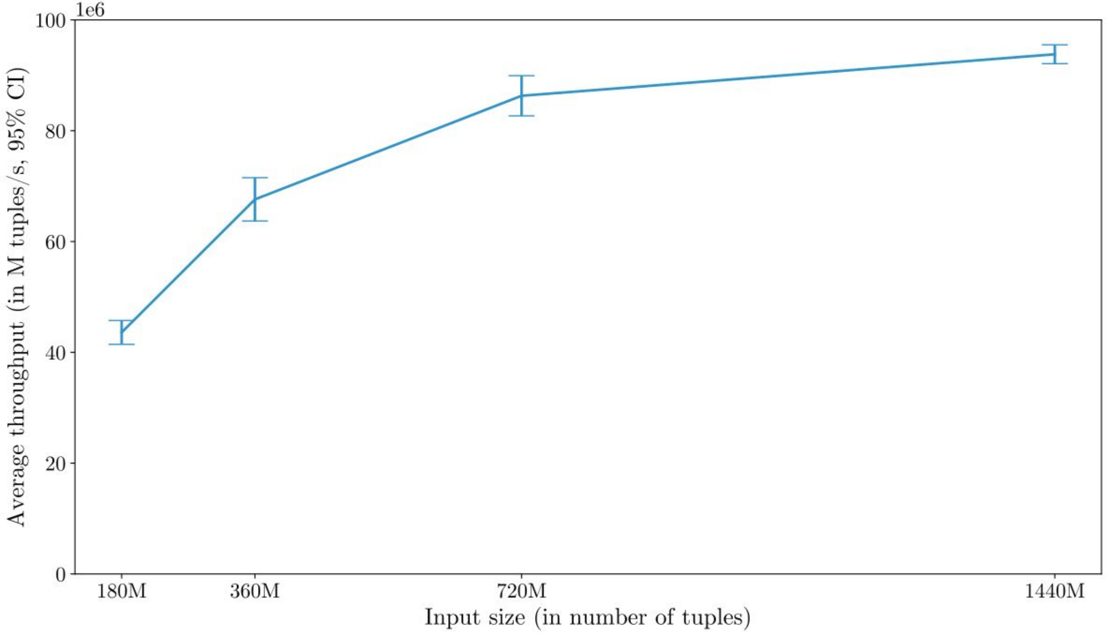
Experiment: Comparing SPEs



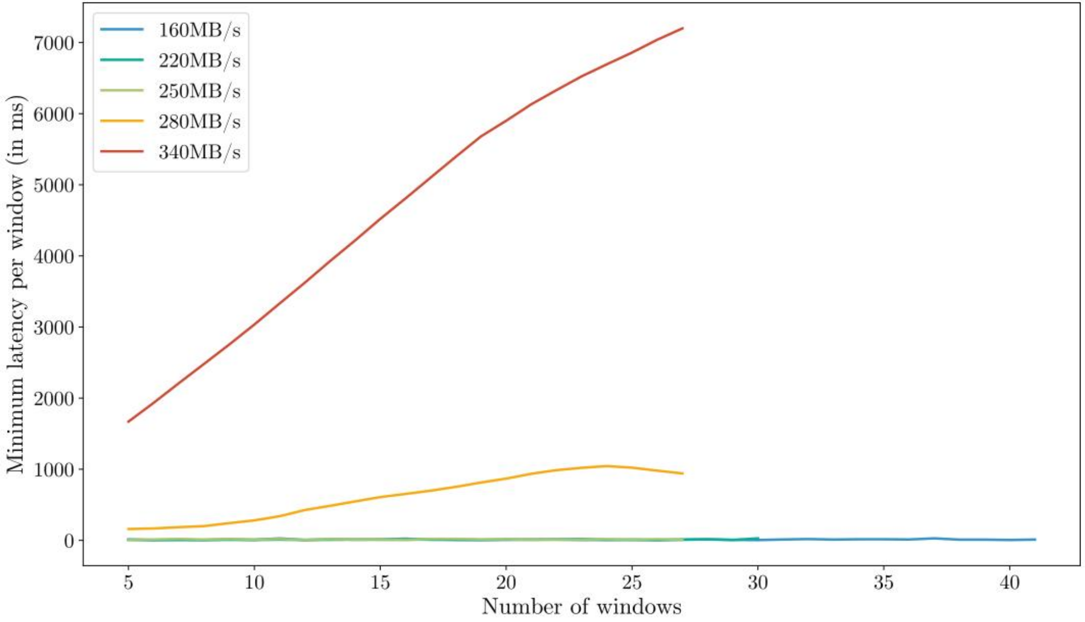
Experiment: Scaling number of nodes



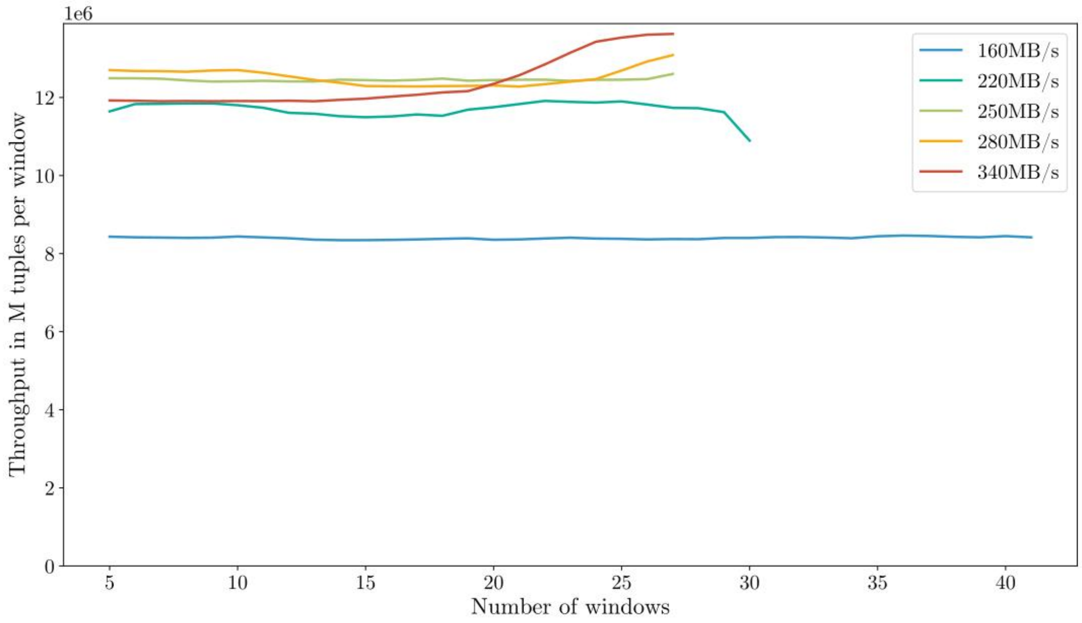
Experiment: Scaling data sizes



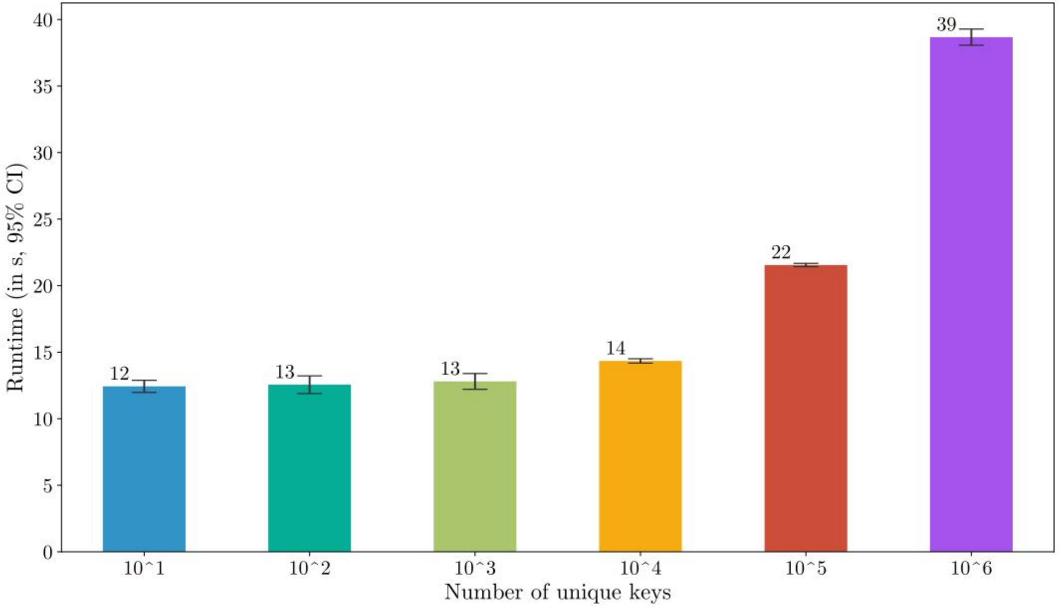
Experiment: Sustainable Throughput (1)



Experiment: Sustainable Throughput (2)



Experiment: Varying Key Ranges



Discussion

✔ Included: Performance comparison with Flink and baseline approaches on a single node

Scale-Out experiments show real-world behaviour (multi-node cluster and unbounded stream)

Performance impact of data rate and key range

✖ To Do: Distribute Apache Flink and compare to our engine prototype on *n* nodes

Evaluate other workloads (i.e., different user-defined queries)

Conclusion

Conclusion

- Combining **query compilation** and **distribution** …
  - … is practically feasible
  - … shows significant performance improvements
- Our prototype achieves 12.6× higher throughput than Flink, and scales well when distributing

Future Work:

- Further extend our evaluation as discussed previously
- Move away from a prototype towards a more complete streaming engine