Today’s real-time data analytics applications are built using a range of software platforms for distributed stream processing. Popular stream processing platforms include Apache Storm [1], Spark [2, 3], and Flink [4]; Google Cloud Dataflow [5]; Microsoft Trill [6]; and emerging frameworks such as Timely [7, 8] and Differential Dataflow [9]. However, engineering and performance advances over the last two decades have not been met by adequate attention to software correctness. Correctness is especially important in this context because the amount of data, distributed deployment, and real-time nature of these applications makes them difficult to understand and to debug [10–12]. Moreover, errors are catastrophic: whereas an error in an offline application might go unnoticed if it is diagnosed and fixed in a timely manner, an error in a streaming application immediately results in either wrong results, delays, or service outages for downstream consumers. To ensure the highest level of safety for present and future applications, we advocate for formal methods work in the rigorous formalization and verification of stream processing programs and systems.

**Challenges.** Before we can achieve verified streaming applications, researchers and practitioners must agree on what it means for stream processing programs to be correct. Unfortunately, this remains an outstanding challenge: there is no unifying language standard, specification, or semantics that is understood across systems. For example, a stream processing program is typically taken to be a dataflow graph of operators, but systems disagree on whether edges in the graph can be ordered streams, or whether all data may be out-of-order. The details of how streams are partitioned between graph operators is also system-dependent. To further complicate matters, modern stream processing platforms often support complex features including user-defined stateful operators [1, 13, 14], communication across partitions [15], querying or interfacing with external services [16, 17], and iterative computation [8].

**Broader context.** In contrast to today’s stream processing applications, database query engines and batch processing applications often benefit from formal semantics built on relational algebra that is well-understood and agreed upon, leading to fruitful research on semantically predictable query languages, optimization, and distributed evaluation. In distributed systems, formal specifications can be exploited in order to prove systems correct under faults, to prove safety through model-checking or to test correctness at runtime using traces. Formalization in these areas has enabled verification, testing, optimization, and synthesis.

**Opportunities.** We identify four correctness dimensions which are common to all stream processing platforms, regardless of specific system choices and features, and represent important opportunities in this space. First, stream processing programs process both out-of-order and in-order data. Data cannot be treated naively as an unordered relation, because some time-series constructs (windowing, streaming aggregation, and interpolation) are implemented in an order-dependent way, but it is also not solely ordered, because distribution and network delays often cause out-of-order arrivals. This raises the need to encode precisely what ordering requirements are made on physical events [8, 18–22]. Second, stream processing systems perform program transformations to achieve distribution: the (generally sequential or declarative) user query is parallelized and distributed across nodes, which requires making choices about how streams are partitioned and how operators are replicated. This raises the need to ensure safe distribution: the distributed code should be semantically equivalent to the original program in some sense [20, 21, 23]. Third, we observe that the performance of operators in a stream processing program is actually critical for correctness, and not just a matter of efficiency. This is because if a program receives more input items than it can handle, it will crash and the fault is likely unrecoverable. This raises the need to infer performance guarantees on operators to ensure predictable execution, ideally at compile time [24–29]. Finally, due to distributed deployment, stream processing applications should be fault-tolerant. This dimension is well-studied by existing fault tolerance work in the streaming context [30–34].

**Outlook.** If successful, formalization could shape design and tool support for the future of stream processing systems. Figure 1 shows how formal models could fit in a unified architecture for stream processing. The system interface offers well-defined formal semantics and supports formal analyses (checking whether certain assumptions are met), which inform the compiler in generating an efficient and correct implementation. Ordering requirements are encoded in the formal execution semantics, and can be exploited by formal analyses and tools. Safe distribution semantics are exploited by the distributed implementation and compiler/optimizer. Performance guarantees are provided by a formal analysis of the user query, and preserved by the compiler/optimizer.
REFERENCES


