Letter from the Special Issue Editors

The precursors of data-stream systems began to show up in the late 1980s and early 1990s in the form of "reactive" extensions to data management systems. With such extensions, there was a reversal of sorts between the roles of data and query. Database requests – in the form of continuous queries, materialized views, event-condition-action rules, subscriptions, and so forth – became persistent entities that responded to newly arriving data.

The initial generation of purpose-built stream systems addressed many issues: appropriate languages, dealing with unbounded input, handling delay and disorder, dealing with high data rates, load balancing and shedding, resiliency, and, to some extent, distribution and parallelism. However, integration with other system components, such as persistent storage and messaging middleware, was often rudimentary or left to the application programmer.

The most recent generation of stream systems have the benefit of a better understanding of application requirements and execution platforms, by virtue of lessons learned through experimentation with earlier systems. Scaling, in cloud, fog, and cluster environments, has been at the forefront of design considerations. Systems need to scale not just in terms of stream rate and number of streams, but also to large numbers of queries. Application tuning, operation, and maintenance have also come to the forefront. Support for tradeoffs among throughput, latency, accuracy, and availability is important for application requirements, such as meeting service-level agreements. Resource management at run time is needed to enable elasticity of applications as well as for managing multi-tenancy both with other stream tasks and other application components. Many stream applications require long-term deployment, possibly on the order of years. Thus, the ability to maintain the underlying stream systems as well as evolve applications that run on them is critical. State management is also a concern, both within stream operators and in interactions with other state managers, such as transactional storage. There has also been a focus on broadening the use of stream-processing systems, but through programming models for non-specialists and by supporting more complex analyses over streams, such as machine-learning techniques.

This issue is devoted to this next generation of stream-processing, looking at particular systems, specific optimization and evaluation techniques, and programming models.

The first three papers discuss frameworks that support composing reliable and distributed stream (and batch) processing networks out of individual operators, but are somewhat agnostic about what the particular operators are. Samza (Kleppman, et al.) is a stream-processing framework developed initially at LinkedIn that supports stream operators loosely coupled using the Kafka message broker. The use of Kafka reduces dependencies between stream stages, and provides replicated logs that support multiple consumers running at different rates. The next paper (Fu, et al.) introduces Heron, whose API is compatible with Twitters early streaming platform, Storm. Heron features support sustained deployment and maintenance, such as resource reservations and task isolation. The paper discusses alternative back-pressure mechanisms, and how Heron supports at-least-once and at-most-once messaging semantics. Apache Flink (Carbone, et al.) is a framework that supports a general pipelined dataflow architecture that handles both live stream and historical batch data (and combinations) for simple queries as well as complex iterative scripts as found in machine-learning. The paper discusses mechanisms for trading latency with throughput; the use of in-stream control events to help checkpointing, track progress and coordinate iterations; and low-interference fault-tolerance taking consistent snapshots across operators without pausing execution.

The next three papers deal with complete systems that include specific query languages. In Connected Streaming Analytics (CSA) from Cisco (Shen, et al.), stream-processing components can be embedded in network elements such as routers and switches to support Internet-of-Things applications. Given this execution environment, it is important that stream queries not interfere with high-priority network tasks. CSA uses a container mechanism to constrain resources and promote portability. The language is SQL with window extensions. CSA supports different kinds of window joins: best-effort join combines data immediately on receipt, whereas coordinated join matches items based on application time, which may require buffering. Trill (Chandramouli, et

al.) shares goals with Flink in seeking a single engine that can work for online, incremental and offline processing, and supports latency-throughput tradeoffs as appropriate for different contexts. It takes a library approach that allows mutual embedding with applications written in high-level languages. Trill queries are written in a LINQ-based language that supports tempo-relational operations, along with timestamp manipulation capabilities. For performance, it uses a columnar in-memory representation of data batches. The subsequent paper looks at language runtime support for the IBM Stream Processing Language (SPL) (Schneider, et al.). The SPL runtime provides certain execution guarantees, such as isolation of operator state and in-order delivery, and satisfies performance goals such as long-term query execution without degradation and efficient parallel execution. Performance optimizations include both "fusion" (combining operators into a single Processing Element) and "fission" (replicating a portion of the query graph).

The next three papers consider stream-processing optimizations and guarantees. While several of the systems in the foregoing papers provide a means to make performance tradeoffs, in practice it can be difficult for a user to determine the best way to adjust the control knobs. The FUGU stream-processing system (Heinze, et al.) employs strategies that automate the adjustment of these parameters, based on on-line profiling of query execution and user-provided latency specifications. The paper from Worchester Polytechnic Institute (Rundensteiner, et al.) looks at several methods to improve performance of pattern-matching queries, using a variety of sharing strategies. Examples are Event-Sequence Pattern Sharing, which determines temporal correlations between sub-patterns in order to decide whether sharing is beneficial, and Shared Event-Pattern Aggregation, which looks for shared aggregation opportunities at the sub-pattern level. Several early stream systems had the ability to access stored data in some form, for example, to augment stream events with information from a look-up table. However, these systems gave limited consistency guarantees, either between the stream and the stored data, or between shared access to stored data across stream operators. The S-Store system (Tatbul, et al.) develops a stream-processing model that provides several correctness guarantees, such as traditional ACID semantics, order-of-execution conditions and exactly-once semantics.

The last two papers are oriented towards application development. Most stream systems require queries to be written in a special request language or a general-purpose programming language, either of which is a hurdle for non-CS experts. The Event Model (TEM) (Etzion, et al.) allows a user to specify an event-driven application by concentrating on application logic, expressed in diagrams and associated condition tables. The TEM environment can fill in low-level details and manage the conversion to a particular stream-processing system. "Live" analytics are a major driver of next-generation stream systems. Our final paper looks at mining for events in a text stream (Grossniklaus, et al.). It adopts a tool-kit approach that allows easy implementation of many of the published approaches in this domain. In addition, it describes an evaluation platform for comparing alternative event-detection techniques.

David Maier, Badrish Chandramouli
Portland State University (Maier), Microsoft Corporation (Chandramouli)