S

Stream Query Optimization

Martin Hirzel[1], Robert Soulé[2], Buğra Gedik[3], and Scott Schneider[1]
[1]IBM Research, Yorktown Heights, NY, USA
[2]Università della Svizzera Italiana (USI), Lugano, Switzerland
[3]Department of Computer Engineering, Bilkent University, Ankara, Turkey

Synonyms

Continuous query optimization; Stream processing optimization

Definition

Stream query optimization is the process of modifying a stream processing query, often by changing its graph topology and/or operators, with the aim of achieving better performance (such as higher throughput, lower latency, or reduced resource usage), while preserving the semantics of the original query.

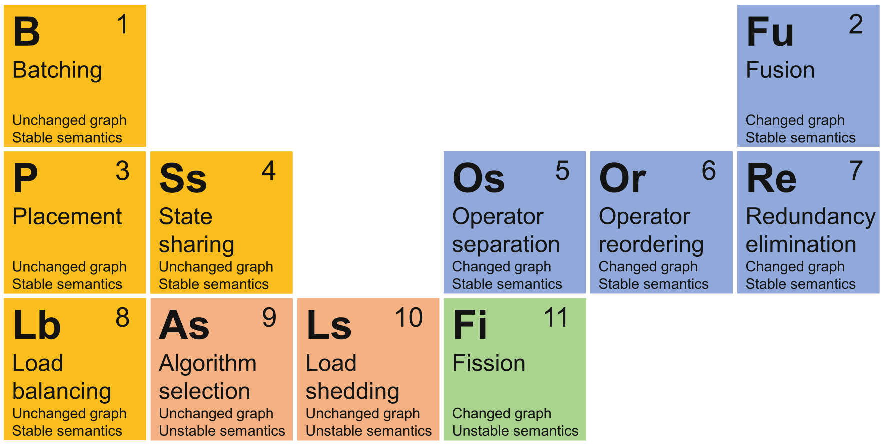
Overview

A stream query optimization modifies a stream query to make it faster. Users want stream queries to be fast for several reasons. They want to grasp opportunities or avert risks observable on the input streams before it is too late. They want any views derived from the input streams to be up-to-date and not stale. And they want their system to keep up with the rate of input streams without falling behind, which would require shedding load or saving data to disk for later processing.

Knowing about stream query optimizations helps developers at all layers. Application developers who know about stream query optimizations can get the most out of the optimizations built into their streaming platform and can supplement them by hand-optimizing their application where necessary. Streaming platform developers can use knowledge about stream query optimizations to make their platform faster by implementing additional optimizations or by generalizing their existing optimizations to apply in more situations. Finally, researchers who invent new optimizations need to know the state-of-the-art optimizations to channel their efforts into the most innovative and impactful direction.

The rest of this section introduces some basic concepts and gives a high-level overview and categorization of the most common stream query optimizations.

An optimization should be both safe and profitable. An optimization is *safe* if it can be applied to a stream query without changing what it computes, as determined by the user's requirements. An optimization is *profitable* if it makes the stream query faster, as measured by metrics that matter to the user, such as



Stream Query Optimization, Fig. 1 Overview of stream query optimizations discussed in this entry

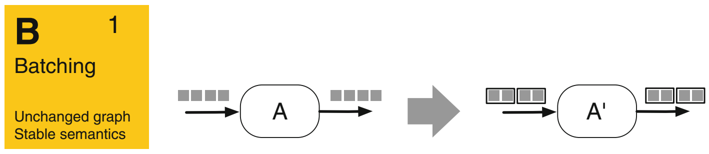
throughput, latency, or resource efficiency. There is a substantial literature on different stream query optimizations, with different safety and profitability characteristics. This entry lists the most common optimizations along with short descriptions. More in-depth descriptions can be found in our survey paper and tutorial on stream processing optimizations (Hirzel et al. 2014; Schneider et al. 2013).

Stream query optimizations are best understood with respect to stream graphs. A *stream graph* is a directed graph whose edges are streams and whose nodes are operators. Root and leaf nodes are called sources and sinks, respectively. This entry uses terminology that makes only few assumptions so as not to unnecessarily restrict its scope. For instance, this entry does not assume restrictions on the shape of the stream graph: unless specified otherwise, it does not assume that stream graphs are acyclic, or are single-source-single-sink, or are trees. A *stream* is an ordered sequence of data items, which are values that can range from simple numbers to flat tuples to more elaborate structured data that may be deeply nested and have variable size. Streams are conceptually infinite, in the sense that as the streaming computation unfolds over time, the sequence of data items is unbounded in length. *Operators* are primarily stream transformers but can also have state and side effects beyond the output streams they produce. Indeed, sources and sinks are operators that typically have the side effect of continuously consuming input from and producing output to the external world outside of the stream graph.

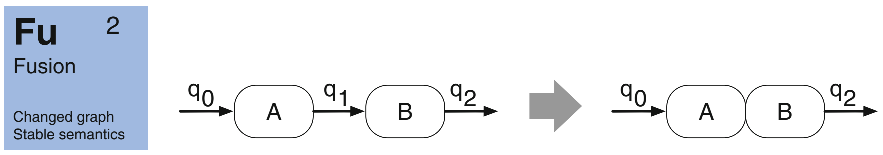
Figure 1 lists the most popular stream query optimizations. Each optimization has a symbol (e.g., **B**), a name (e.g., Batching), and two annotations indicating its effect on the graph and the semantics. The optimizations to the left (shown in orange and red) leave the stream graph unchanged. This means the graph still contains the same nodes and edges, and any changes are limited to the behavior of individual operators. The optimizations on the right (shown in blue and green) change the stream graph. The optimizations on the top (shown in orange and blue) keep the semantics stable. This means that as long as their safety preconditions are satisfied, the observable behavior of the stream query is the same before and after applying the optimization. These optimizations are safe in the sense discussed above. The extreme case is batching and fusion at the very top, which are not only safe, but have safety preconditions that are trivial to satisfy. Only the three optimizations at the very bottom have unstable semantics: load shedding, which always changes the semantics, and algorithm selection and fission, which sometimes change semantics if the algorithm is approximate or if the fission perturbs the order of data items. These forms of semantic changes are sometimes tolerable depending on application requirements.

The following section will elaborate on each of the optimizations from Fig. 1 with illustrations, definitions, and literature references.

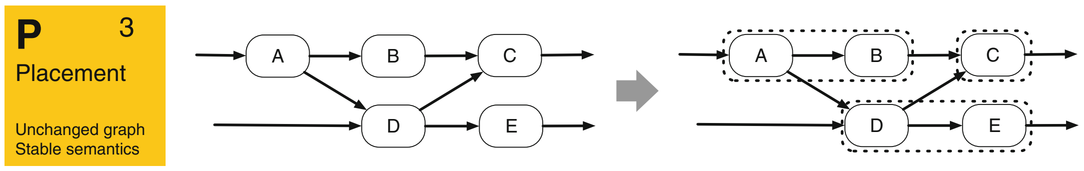
Key Research Findings



Batching *Definition:* Batching reduces overhead by processing multiple data items together. This optimization improves throughput by amortizing the cost of operator-firing and communication over several data items. However, this throughput gain is usually at the expense of additional latency, as an operator cannot fire until it has received a batch-size number of data items. *References:* In the literature, batching is also called train scheduling (Carney et al. 2003) and execution scaling (Gordon et al. 2006). Batching can be dynamic, as in SEDA (Welsh et al. 2001), or static, as in StreamIt (Gordon et al. 2006).

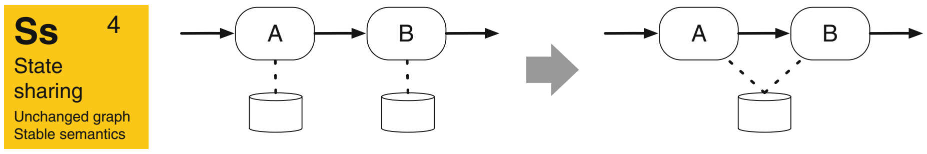


Fusion *Definition:* Fusion combines smaller operators into a larger one, to avoid the overhead of data serialization and transport. Operators may be fused in many ways, for example, by placing the operators into the same thread or by keeping operators in separate threads that share a common address space. Fusion may come at the cost of decreased pipeline parallelism. *References:* StreamIt (Gordon et al. 2002) aggressively fuses fine-grained operators, followed by fission. In Aurora, fusion is referred to as superbox scheduling (Tatbul et al. 2003). System S uses the COLA fusion optimizer (Khandekar et al. 2009), which balances safety and profitability constraints.

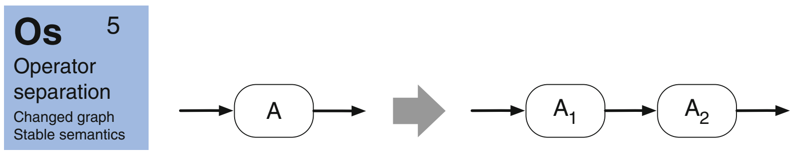


Placement *Definition:* Placement assigns operators to hosts and cores to reduce communication costs or better utilize available resources. Frequently, these two goals are at odds. When multiple operators are placed on the same host, they communicate at lower cost, but they compete for common resources, such as disk, memory, or CPU. On the other hand, when operators are placed on different hosts, that reduces contention but incurs higher communication costs.

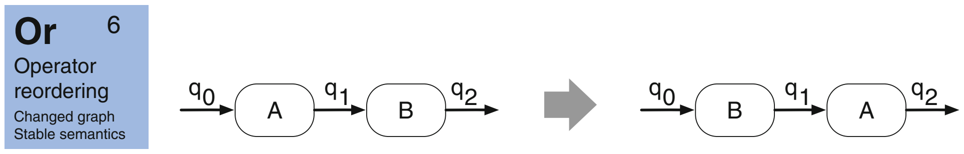
References: StreamIt uses placement to optimize streaming applications deployed on multi-core machines with nonuniform memory access (Gordon et al. 2002). Pietzuch et al. use metrics gathered from network conditions to place operators in a distributed setting (Pietzuch et al. 2006). SODA incorporates job admission with the placement decisions (Wolf et al. 2008).



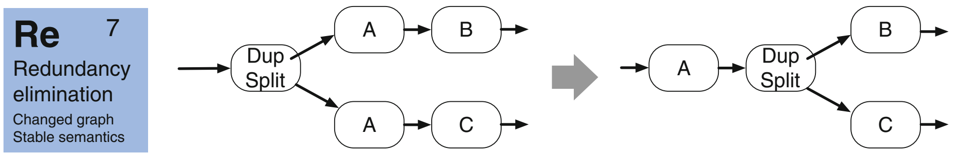
State sharing *Definition:* State sharing attempts to avoid unnecessary copies of data. While the main goal of the optimization is to reduce the memory footprint of a streaming application, it can also impact performance by reducing stalls due to cache misses or disk I/O. *References:* State sharing can be applied generally between streaming operators, such as in the work of Brito et al. (2008). Or, it can be applied in more restricted forms, such as in CQL, which shares only window state (Arasu et al. 2006), or in the work of Sermulins et al., which shares queue state between two pipelined operators (Sermulins et al. 2005).



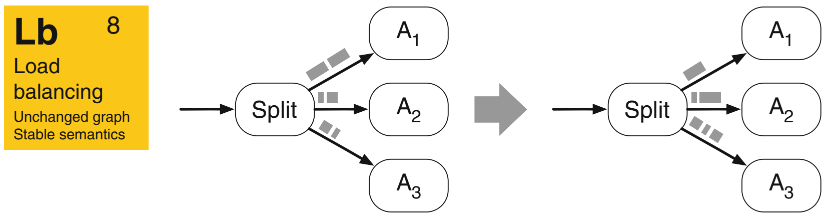
Operator separation *Definition:* Operator separation splits a large computation into smaller steps. In some cases, this optimization can result in reduced resource consumption. Often, this optimization is used to enable other optimizations, such as operator reordering. *References:* Using algebraic equivalences for separating operators is a common technique in database query execution planning (Garcia-Molina et al. 2008). Yu et al. (2009) present a stream query compiler that uses explicit annotations to determine when to separate aggregate operators. Decoupled software pipelining separates general code by analyzing data dependencies from first principles (Ottoni et al. 2005).



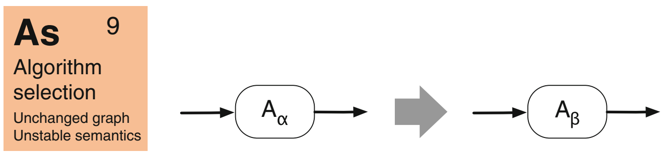
Operator reordering *Definition:* A reordering optimization moves more selective operators, which reduce the data volume, upstream. This has the benefit of reducing the data flowing into downstream computation, thus eliminating unnecessary work. However, care must be taken to preserve the desired semantics, and operators should only be re-ordered if the operations are commutative. *References:* Reordering based on the properties of relational algebra is common in database query planning (Garcia-Molina et al. 2008). The Volcano (Graefe 1990) system implements a particularly profitable case of reordering: swapping split and merge operators in a data-parallel pipeline to avoid choke-points. Eddies (Avnur et al. 2000) is a dynamic technique for finding the most profitable ordering of operators with independent selectivities.



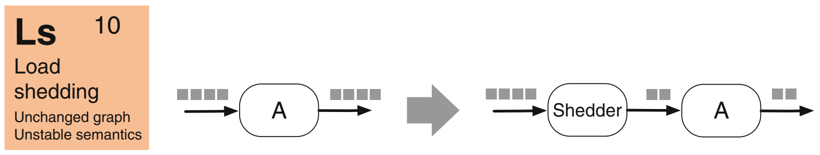
Redundancy elimination *Definition:* Redundancy elimination eliminates superfluous computations. This optimization must ensure that removing a given computation does not change the resulting output. While, in general, determining program equivalence is undecidable, in practice, streaming languages are often based on an algebra, which makes the optimization feasible. *References:* The Rete algorithm (Forgy 1982) is a highly-influential approach to detecting and eliminating redundancies in a multi-tenant system. NiagaraCQ (Chen et al. 2000) applied similar ideas to processing streaming XML. Pietzuch et al. (2006) eliminate redundancy at application launch time.



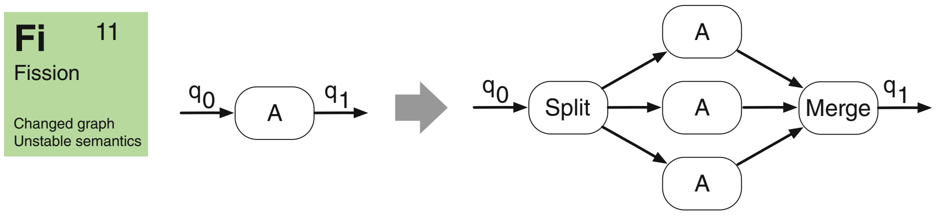
Load balancing *Definition:* Load balancing attempts to distribute workload evenly across resources. To be effective, a load-balancing optimization must adapt to workload skew, e.g., when there are many accesses to a popular data item. *References:* River uses intelligent routing for load balancing in a cluster (Arpaci-Dusseau et al. 1999). Caneill et al. use online adaptive routing, which considers downstream communication (Caneill et al. 2016). In contrast, Amini et al. use operator placement for load balancing in a cluster (Amini et al. 2006). StreamIt also uses placement for load balancing, but targets multi-core machines (Gordon et al. 2002).



Algorithm selection *Definition:* Algorithm selection uses a different algorithm to implement an operator. There are various reasons to change the algorithm. For example, one algorithm may be more general than another. One algorithm may optimize for different criteria. Or, one algorithm may perform better than others under different circumstances. *References:* Changing the operator algorithm for particular workloads is common in database systems (e.g., using a hash join vs. a nested loop join Garcia-Molina et al. 2008). In streaming systems, SEDA allows an operator to pick a different algorithm to provide degraded service (Welsh et al. 2001). Borealis allows an operator to switch to a different algorithm based on a control input (Abadi et al. 2005). And, SODA offers algorithm selection at the granularity of entire jobs (Wolf et al. 2008).



Load shedding *Definition:* Load shedding copes with high load by dropping data items to process. Load shedding may change the expected results of a computation, and therefore the semantics of the streaming application. *References:* Aurora implements priority-based load shedding (Tatbul et al. 2003). Compact Shedding Filters perform load-shedding at data-generating sensors, rather than the server, to avoid unnecessary network communication (Gedik et al. 2008).



Fission *Definition:* Fission, often referred to as data parallelism, attempts to process multiple data items in parallel by replicating an operator. Note that when parallelizing operators, the optimizer must respect ordering or state constraints to ensure the correctness of the computation. *References:* The StreamIt compiler applies fission to stateless operators with static selectivity (Gordon et al. 2006). Schneider et al. explored how to make fission safe in the more general case of partitioned-stateful and selective operators (Schneider et al. 2015). Finally, Brito et al. propose using transactional memory to make fission safe in the case of arbitrary operator state (Brito et al. 2008).

Examples of Application

We illustrate the use of the three most popular optimizations, namely, fission, fusion, and batching, in the context of real-world streaming applications from the literature.

Fission One application that benefits a lot from fission is streaming radio astronomy, which forms evolving imaging maps of radio emission from the sky over a wide range of frequencies (Biem et al. 2010b). High-performance processing is a critical requirement in this application, due to the sheer volume of sensor data that flow in real-time from a very large array of antennas, which is typical of phased array radio telescopes (see SKA 2000). The application's flow graph is organized as a split-merge topology. An initial operator performs frequency mapping and blocking so that a subsequent operator splits the data into blocks of frequency channels. Each such block is then processed in parallel by performing indexing and convolution operations, forming a costly stateless parallel region that highly benefits from fission.

Finally, the results are merged via an aggregation to form the complete imaging map.

Fusion In Biem et al. (2010a), a streaming application is presented for maintaining live traffic status information using GPS sensor data. The application processes floating car data originating from public transportation vehicles to extract up-to-date traffic information, such as speed and traffic flow measurements at the level of streets within a city, traffic volume measurements by region, estimates of travel times between different points, etc. The application is organized as a graph of streams and operators, where different operators perform individual tasks, such as data parsing and cleaning, snapping GPS points to roads, aggregation and statistics maintenance, prediction of travel times, etc. The operators in the flow graph of the application are grouped into processing elements, which are then distributed across machines. The operators that are assigned to the same processing element are fused inside a shared address space, in order to reduce the data transfer latency.

Batching The batching optimization is particularly useful when interacting with external systems. A common use case for such interaction is managing state. For instance, LinkedIn, which is a business- and employment-oriented social networking service, runs several streaming applications that manage state, such as user profiles and aggregate counts. These applications include email digest generation, top-k relevant category detection, and profile update standardization, among others (Noghabi et al. 2017). In these applications, the state needs to be accessed from either the local disk-based or remote state management systems. Similarly, many of these streaming applications write their output to an external system, such as a message queue. When an external interaction is to be performed on a per-tuple basis, batching is an effective optimization that can amortize the overheads and significantly improve the performance.

Future Directions for Research

In an ideal world, programmers would code their streaming applications at the most natural level of abstraction without having to worry about runtime performance, and the streaming system would automatically execute the applications with consistently high performance. While stream query optimizations have made significant advances toward this goal, they still fall short on automation and predictability.

It is still difficult to fully automate streaming optimizations, because automatic optimizers may not find the most profitable setting. Finding the most profitable setting ahead-of-time, before executing the streaming application, is tricky because the performance model may be complicated (e.g., when multiple optimizations interact) or some information required by the performance model may be missing or hard to predict. An alternative to ahead-of-time optimization is online feedback-directed optimization, which is subject to the trade-offs inherent in the SASO properties of feedback control (stability, accuracy, settling, and no overshoot) (Hellerstein et al. 2004). Work such as that of De Matteis and Mencagli, which uses a control-theoretic approach to perform online adaptations to optimize for latency (De Matteis et al. 2016), is a promising direction.

When streaming applications fall short of their expected peak performance, programmers often optimize them by hand. This has the advantage of giving programmers more control over squeezing the last bit of performance out of their application. Unfortunately, it can also clutter their code, can make the performance brittle by over-fitting to the current workload and execution environment, and may even inhibit automated optimizations. One approach to address these issues is to decouple the optimizer hints and directives from the core application logic (Hirzel et al. 2017).

Overall, there is still much work to be done in developing more effective optimizations, more reliable performance models, and more unintrusive language features for giving optimizer hints. This entry represents a snapshot of the state of the art, and we encourage the reader to venture beyond it.

Cross-References

- ▸ Continuous Queries
- ▸ Introduction to Stream Processing Algorithms
- ▸ Sliding-Window Aggregation Algorithms

References

Abadi DJ, Ahmad Y, Balazinska M, Çetintemel U, Cherniack M, Hwang JH, Lindner W, Maskey AS, Rasin A, Ryvkina E, Tatbul N, Xing Y, Zdonik S (2005) The design of the Borealis stream processing engine. In: Conference on innovative data systems research (CIDR), pp 277–289

Amini L, Jain N, Sehgal A, Silber J, Verscheure O (2006) Adaptive control of extreme-scale stream processing systems. In: International conference on distributed computing systems (ICDCS)

Arasu A, Babu S, Widom J (2006) The CQL continuous query language: semantic foundations and query execution. J Very Large Data Bases (VLDB J) 15(2): 121–142

Arpaci-Dusseau RH, Anderson E, Treuhaft N, Culler DE, Hellerstein JM, Patterson D, Yelick K (1999) Cluster I/O with river: making the fast case common. In: Workshop on I/O in parallel and distributed systems (IOPADS), pp 10–22

Avnur R, Hellerstein JM (2000) Eddies: continuously adaptive query processing. In: International conference on management of data (SIGMOD), pp 261–272

Biem A, Bouillet E, Feng H, Ranganathan A, Riabov A, Verscheure O, Koutsopoulos HN, Rahmani M, Guc B (2010a) Real-time traffic information management using stream computing. IEEE Data Eng Bull 33(2): 64–68

Biem A, Elmegreen B, Verscheure O, Turaga D, Andrade H, Cornwell T (2010b) A streaming approach to radio astronomy imaging. In: Conference on acoustics, speech, and signal processing (ICASSP), pp 1654–1657

Brito A, Fetzer C, Sturzrehm H, Felber P (2008) Speculative out-of-order event processing with software transaction memory. In: Conference on distributed event-based systems (DEBS), pp 265–275

Caneill M, El Rheddane A, Leroy V, De Palma N (2016) Locality-aware routing in stateful streaming applications. In: International conference on middleware, pp 4:1–4:13

Carney D, Cetintemel U, Rasin A, Zdonik S, Cherniack M, Stonebraker M (2003) Operator scheduling in a data stream manager. In: Conference on very large data bases (VLDB), pp 309–320

Chen J, DeWitt DJ, Tian F, Wang Y (2000) NiagaraCQ: a scalable continuous query system for internet databases. In: International conference on management of data (SIGMOD), pp 379–390

De Matteis T, Mencagli G (2016) Keep calm and react with foresight: strategies for low-latency and energy-efficient elastic data stream processing. In: Principles and practice of parallel programming (PPoPP), pp 13:1–13:12

Forgy CL (1982) Rete: a fast algorithm for the many pattern/many object pattern match problem. Artif Intell 19:17–37

Garcia-Molina H, Ullman JD, Widom J (2008) Database systems: the complete book, 2nd edn. Prentice Hall, Upper Saddle River

Gedik B, Wu KL, Yu PS (2008) Efficient construction of compact shedding filters for data stream processing. In: International conference on data engineering (ICDE), pp 396–405

Gordon MI, Thies W, Karczmarek M, Lin J, Meli AS, Lamb AA, Leger C, Wong J, Hoffmann H, Maze D, Amarasinghe S (2002) A stream compiler for communication-exposed architectures. In: Conference on architectural support for programming languages and operating systems (ASPLOS), pp 291–303

Gordon MI, Thies W, Amarasinghe S (2006) Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In: Conference on architectural support for programming languages and operating systems (ASPLOS), pp 151–162

Graefe G (1990) Encapsulation of parallelism in the Volcano query processing system. In: International conference on management of data (SIGMOD), pp 102–111

Hellerstein JL, Diao Y, Parekh S, Tilbury DM (2004) Feedback control of computing systems. Wiley, Hoboken

Hirzel M, Soulé R, Schneider S, Gedik B (2014) A catalog of stream processing optimizations. ACM Comput Surv (CSUR) 46(4):1–34

Hirzel M, Schneider S, Gedik B (2017) SPL: an extensible language for distributed stream processing. Trans Program Lang Syst (TOPLAS) 39(1):5:1–5:39

Khandekar R, Hildrum I, Parekh S, Rajan D, Wolf J, Wu KL, Andrade H, Gedik B (2009) COLA: optimizing stream processing applications via graph partitioning. In: International conference on middleware, pp 308–327

Noghabi SA, Paramasivam K, Pan Y, Ramesh N, Bringhurst J, Gupta I, Campbell RH (2017) Samza: stateful scalable stream processing at LinkedIn. In: Conference on very large data bases (VLDB), pp 1634–1645

Ottoni G, Rangan R, Stoler A, August DI (2005) Automatic thread extraction with decoupled software pipelining. In: International symposium on microarchitecture (MICRO), pp 105–118

Pietzuch P, Ledlie J, Schneidman J, Roussopoulos M, Welsh M, Seltzer M (2006) Network-aware operator placement for stream-processing systems. In: International conference on data engineering (ICDE), pp 49–61

Schneider S, Gedik B, Hirzel M (2013) Tutorial: stream processing optimizations. In: Conference on distributed event-based systems (DEBS), pp 249–258

Schneider S, Hirzel M, Gedik B, Wu KL (2015) Safe data parallelism for general streaming. IEEE Trans Comput (TC) 64(2):504–517

Sermulins J, Thies W, Rabbah R, Amarasinghe S (2005) Cache aware optimization of stream programs. In: Conference on languages, compiler, and tool support for embedded systems (LCTES), pp 115–126

SKA Telescope (2000) Square kilometre array telescope. https://skatelescope.org. Retrieved Nov 2017

Tatbul N, Cetintemel U, Zdonik S, Cherniack M, Stonebraker M (2003) Load shedding in a data stream manager. In: Conference on very large data bases (VLDB), pp 309–320

Welsh M, Culler D, Brewer E (2001) SEDA An architecture for well-conditioned, scalable Internet services. In: Symposium on operating systems principles (SOSP), pp 230–243

Wolf J, Bansal N, Hildrum K, Parekh S, Rajan D, Wagle R, Wu KL, Fleischer L (2008) SODA: an optimizing scheduler for large-scale stream-based distributed computer systems. In: International conference on middleware, pp 306–325

Yu Y, Gunda PK, Isard M (2009) Distributed aggregation for data-parallel computing: interfaces and implementations. In: Symposium on operating systems principles (SOSP), pp 247–260