# Operator Scheduling for Data Streams

Guna Prasaad

Microsoft Research India

## 1 Introduction

Big data is driving the technology industry to its limits. One of the most important dimension of big data is volume. The amount of data generated by social media, online behaviour analysis tools, is growing unbound. Research on handling big data has been majorly on two aspects : efficiently storing huge volumes of data as they arrive and processing them in batches later on. Apache Hadoop, Hive, Pig, are some examples of systems built on such computation workflows.

Competing with this trend, is the rate at which data is generated. Mobile phones and other hand-held devices such as tablets have added to this trend. With the emergence of Internet of Things (IoT), we can only expect this to grow further. Companies are demanding real time analysis of data, in place of store-then-process paradigms used in traditional big data systems, to improve customer experience. Requirement to process such fast arriving data in sub seconds is increasingly becoming a norm. Hence, the need to design and build systems that can handle such workloads has become more important than ever.

## 2 Stream Processing

An umbrella term used to refer to processing of data at such high rates is *stream processing.* The contents of the stream could be anything ranging from a JPEG file, audio signal, sensor readings to web-clickstream records or system logs. The programming model consists of a series of interconnected transformations on these streams culminating at one or more output streams, that are handled by external application interfaces. These transformations are referred to as *stream operations* - they take one or more streams as input, and produce one or more streams as output.

One of the defining feature of a streaming operator is the lifetime. Stream operators run indefinitely, which is in contrast to most operators which are expected to complete execution in some finite amount of time. Another interesting feature is in the incremental nature of the operators, especially in case of operators that are required to maintain a state. Such operators are called *stateful operators* and more often than not, this state is limited to a *window* of the recent past instead of the complete history. In this report, we will mainly focus on data streams in the form of tuples and limit our discussion to a set of operators that are relevant to such applications. Examples include `Map`, `Filter` among the stateless operators and stateful operators such as `Join`, `Aggregate`.

A streaming application is represented in the form of a *continuous query.* A continuous query is a directed acyclic graph, where each node is a stream operator and the edges between them represent flow of stream between operators or input, output streams. This encodes the logic of the transformations that each tuple in the stream must undergo.

## 3 Operator Scheduling

Building a stream processing engine requires engineering at multiple levels: programming model design to represent the transformations accurately and succinctly, underlying system architecture, runtime resource management and many more. As the system is expected to run indefinitely without compromising performance, run time resource management becomes all the more critical. In this report, we discuss one of the important aspects of resource management in a stream processing engine called *operator scheduling.* More broadly, the problem of operator scheduling is deciding *when, where* and *on what* to execute the transformation operators.

The complexity of the solution could range from a simple FIFO scheduling to a hierarchy of schedulers operating at different levels of the system, depending on the complexity of the continuous query itself and on the system architecture. Some commonly discussed CQ graphs are linear chains, fork join, series-parallel graphs and trees. The underlying system could be a single-core machine, multi-core machine or a shared-nothing cluster. The scheduling decisions in a single-core machine involves deciding when to execute which operator and for how much time, whereas in a shared-nothing cluster, the decision is more complex involving distribution and management of workload on different machines in the cluster. Streaming engines built on a shared-nothing cluster are generally called *distributed stream processing engines.*

In the following section, we summarize the various techniques and strategies prevalent in this problem space. The objective of this document is to identify and analyse possible opportunities of parallelization in multi-core systems and hence will emphasize more on how individual machines, either in the distributed cluster or otherwise, execute the job that it is allotted with.

## 4 Related work

There have been several attempts to design stream processing engines, each with its own set of advantages, disadvantages and system complexity. Here, we would discuss some of the important ones such as Aurora, Borealis, Stream, StreamIt and some newer systems both from industry and academia such as Apache Spark, S4, Samza, TRILL.

### 4.1 Aurora

Stream processing as a computational model has been around in the research community for quite some time in various forms such as synchronous dataflow, communicating sequential processes, Kahn process networks. Aurora project (2002) from Brandeis University, Brown University and MIT, first introduced this as a new class of data management problem representing database-active human-passive mode of computation, contrasting it with the human-active database-passive mode then prevalent in traditional RDBMS. Aurora is a single-site stream processing engine and supports two forms of queries: continuous queries and ad-hoc queries. Ad-hoc queries are queries on the complete history of output at specific vertices, called connection points, in the query DAG.

Aurora implements a QoS-aware scheduler which first optimizes for performance and then prioritizes them based on latency requirements in the QoS specifications. The first optimization is by grouping boxes into super boxes, which are then executed as a single unit, in order to save buffer space, avoiding spilling to disk scenarios and save the amount of time spent on queueing and enqueueing in the buffers. Further, they process the tuples in batches to reduce the number of context switches between boxes and also to leverage optimizations in implementation of the actual operator itself. In essence, the Aurora scheduler tells each when to execute and how many tuples to consume from the input buffer queue. Because of the inherent algorithmic complexity of such problems, they employ several different heuristics to solve the problem as efficiently as possible in most realistic scenarios. These scheduling decisions are taken based on runtime statistics, for example average processing cost per tuple, selectivity of an operator and hence fall into the paradigm of dynamic scheduling. The number of tuples to consume on invocation is decided by constraining specific quantities such as latency, throughput, cost per tuple.

To sum up, Aurora groups operators into super operators and dynamically schedules them based on run-time statistics on a single-thread. They do not exploit opportunities of parallelization in multi-core environments. They briefly consider a multi-threaded approach in which each operator runs on a thread of its own, but however discard it later as it is not scalable for huge query graphs.

## 4.2 Borealis

Borealis is the distributed version of the Aurora project, and it borrows the distribution functionality from Medusa. It offers many more interesting features such as dynamic revision of query results, handling partial data, dynamic modification of queries and time travel based on experience of deploying such streaming systems in the real world. A Borealis application, which is a single connected diagram of operators is deployed on a network of $N$ servers, called *sites.*

Scheduling in Borealis can be divided into two phases - the initial diagram distribution followed by dynamic optimization during run time. The primary goal of initial diagram distribution is to produce a feasible allocation of boxes and tables and to collocate them to minimize IO cost. The distribution here mainly focusses on reducing access times to stored data such as tables unlike other schedulers that focus on the properties of the query graph and operators itself. This happens in two phases: it first identifies potential groupings of operators using bounding box computations of operations on each table, based on overlaps and capacity of individual sites and allots the most demanding box to the site with greatest capacity; the second phase completes the process by appropriately assigning the remaining boxes using slack computations.

The dynamic optimization in Borealis is based on a hierarchy of monitors and optimizers. Each site has an associated monitor called local monitor and sites that produce outputs have end point monitors. local sites communicate statistics to end-points monitors. There are three levels of optimizers that handle overload and redistribution: local, neighbourhood and global. Every level escalates issues that are not solvable at that level. Local optimizers decide on load shedding, neighbourhood optimizers on load balancing between neighbouring sites and the global optimizer uses the end point monitor information for global decision in the system.

## 4.3 Stream

Stream is a data stream management system which was built at Stanford. Stream processing engines represent a regular pattern of memory allocation and de-allocation and without proper memory management such workloads can push the system into unacceptable durations of inactivity. Stream presents an algorithm for operator scheduling for minimizing memory requirements in the streaming system. Such requirements become quintessential in scenarios with high volume, irregular and bursty arrival of data.

The common practice during such bursty arrivals is to buffer the tuples at the input endpoint. However, such requirements could lead to a huge memory requirement while the least memory required to handle the workload might be much lower. Consider a simple example with two operators $o_1$ and $o_2$ with selectivities 0.02 and 5 respectively. It is better to buffer the extra data just after $o_1$ than before $o_1$ as the memory requirement drastically reduces, thanks to the very small selectivity of operator $o_1$. The idea is to schedule that operators that leads to the maximum reduction in memory requirement. However, this simple greedy approach does not yield in scenarios where a highly selective operator immediately follows a less selective operator and hence both these operators have to be scheduled together. The algorithm presented identifies such chains of operators that when run leads to maximum reduction in memory requirements. For purposes of this document, we could summarize this strategy as follows: the scheduler dynamically schedules the operators in a single thread based on a heuristic.

## 4.4 PIPES

Most of the multicore scheduling strategies encountered till now can be classified into one of the two classes: operator-per-thread (OTS) or graph-per-thread (GTS). In OTS, each operator runs on a different thread and GTS, all operators are run on the same thread and is controlled by a scheduler. Some techniques group operators to decrease the overheads, however not many have tried to pipeline these grouped operators and this is exactly what PIPES proposes. They group operators into super operators, and run each of these super operators in a thread of its own. This grouping of operators could be based on any objective: minimize latency, increase throughput, etc.

## 4.5 StreamIt

StreamIt is a language designed at MIT to help programmers annotate potential venues for parallelization in a streaming system. Such carefully designed constructs help analyse the program statically and optimize it during compile time. StreamIt is a more general purpose language and designed especially for streaming signal processing workloads. However, the optimizations and scheduling strategies discussed are generally similar to much of the data processing workloads we focus on.

StreamIt aims at exploiting both data and pipelined parallelism. Every operator is modeled as a filter with specific selectivity, known at compile time. These filters could be either stateless or stateful. The scheduling in StreamIt is based on fusion, replication and pipelining. Stateless filters are fused into a single filter and executed as a single unit. Many instances of stateless filters are invoked taking advantage of data parallelism in the computation. Finally, these filters are pipelined with the stateful filters, which are neither replicated nor partitioned.

As an improvement to the above mentioned optimization, they propose two other strategies : *cache-aware fusion* and *cache-aware scaling*. Fusion offers many benefits including reduced method call overheads and improved producer consumer locality. However, excessive fusion might mean that the combined instruction and data size of the filters might overflow the caches and hence hamper the performance. To avoid this, filters are fused only until they reach the overflow limit of the cache. This strategy provides a mean 1.4x improvement over the complete fusion strategy. Tuples are processed in batches in order to amortize the cache miss overheads. While batching in itself provides amortizing benefits, excessive batching might lead to overflow of data buffers of the operators, again leading to cache overflow. So, they developed a cache-aware scaling heuristic that is effective in selecting an appropriate intermediate scaling factor and this improvement provides 1.9x improvement over cache-aware fusion alone.

## 4.6 Apache Storm

Apache Storm is a real-time stream processing framework built by Twitter with the key goal of guaranteed message processing. A message cannot be lost due to node failures and at least once processing is guaranteed. The details of how Storm is designed to achieve such robustness is beyond the scope of this report. However, we would like to briefly discuss the operator scheduling implemented in Storm. The primary scheduler in Storm is called Nimbus, which is the main server where user code is uploaded. Nimbus distributes this code among the worker nodes for execution. Nimbus is responsible for keeping track of progress of workers and handling node failures.

Each storm application is a continuous query and is called a topology. When the topology is created it can specify how many instances of the operator (called bolts) to run, and each instance is called a task. The tasks are executed in parallel in different nodes. When two bolts with multiple instances are connected to each other, the input from the first bolt flows to the task belonging to the receiving bolt according to some message grouping rules. This grouping could be based on a key for stateful operator or simply some heuristic to balance load for stateless operators.

Every node in the Storm cluster has a fixed number of worker slots, equal to the number of cores in the machine. These slots are filled by Java processes called workers. Every worker owns a set of threads each running an executor. Tasks allotted by the Nimbus scheduler are executed in these executors. The deployment plan is essentially an assignment of executors to workers and workers to worker slots in the nodes. The distribution strategy involves two components: topology-aware distribution which groups executors based on locality and traffic aware scheduling that reduces inter-node and inter-slot traffic on the basis of communication patterns at runtime. They do not currently support movement of stateful components across workers.

## 4.7 Apache S4

S4 is another streaming system which stands for Simple Scalable Streaming System, developed by Yahoo. It employs the popular actors model for computations. The processing model is inspired by Map Reduce and uses key-based programming model. S4 creates a dynamic network of processing elements (PE) that are arranged in a DAG at runtime. PEs are basic computational elements in S4 and it is the user-defined code specifying how to process events. A new instance of a PE is created for different values of the key attribute and PEs can output events to other PEs in the system. One of the biggest challenges in PE architecture is in key attributes with large value domains and hence the system removes/discards unused PEs after a certain time using validation time mechanisms such as Time-To-Live(TTL). Processing nodes host processing elements and events in the S4 are routed to the same processing nodes by maintaining a hash on the key attributes. Processing nodes are responsible for listening to events, executing operations on the incoming events using the PEs, dispatching events with the assistance of the communication layer and emitting output events. The operations of the processing nodes could be easily parallelized in a multicore environment, however no clear indication of multi-threaded or multi-process execution of the processing node functionalities have been mentioned in the documentation.

## 4.8 Apache Samza

Apache Samza is another near-realtime asynchronous computational framework for stream processing originally developed at LinkedIn. Storm and Samza are very similar - both support partitioned stream model, distributed execution environment, etc. However, there are subtle differences in features such as ordering and guarantees, state management. The biggest difference in the execution environment is that Storm uses one thread per task by default whereas Samza uses single-threaded processes called containers. A Samza container may contain multiple tasks but there is only one thread that invokes each of the tasks in turn. This means each container is mapped exactly to one CPU core which simplifies the resource model and reduces interference from other tasks. Storm's multi-threaded model has the advantage of taking better advantage of excess capacity on an idle machine, at the cost of a less predictable resource model.

## 4.9 Spark Streaming

Spark streaming is the streaming component in Apache Spark developed at Amplab, U.C.Berkeley. The fundamental unit of data in Spark is called Resilient Distributed Dataset (RDD). The streaming equivalent of RDDs is called DStreams, which is essentially a stream of RDDs. It implements streaming as a continuous processing variant of Spark jobs. It implements a pull based streaming model, in which each stage waits for the previous stage to complete the processing of the RDD before it moves onto processing the next RDD in the stream.

Spark implements two levels of scheduling : the DAG scheduler and the Task Scheduler. DAG scheduler implements stage oriented scheduling - computes a DAG of stages for each Spark job. RDD operations with narrow dependencies such as map, filter are pipelined together into a set of tasks in each stage. But operations with shuffle boundaries are broken down into multiple stages. The DAG scheduler submits the stages as a task set to the underlying task scheduler that runs them on the cluster. In addition, the DAG scheduler also comes up with preferred locations for tasks sets and passes them to the lower level task scheduler. A task represents a unit of work on a partition of a distributed dataset. Each node in the cluster hosts an executor that can execute multiple tasks in parallel. Each application has its own executor processes and if a node has more than one core, it could potentially host multiple executors. To conclude, Spark Streaming tries to exploit data parallelism by partitioning stages into tasks and executing multiple tasks in parallel in the executors.

## 4.10 TRILL

TRILL (stands for trillion events per day) is a batched-columnar stream processing engine developed at Microsoft Research. Trill processes data in batches whose size could be controlled by punctuations and hence claim to support processing of data in the entire latency spectrum. One of the novel contributions of TRILL is columnar data organization of batches. Although it has been quite prevalent in modern DBMS, this is the first attempt to deploy the strategy for streaming systems. The data batches are stored in columnar fashion and they use a novel HLL code generation technique to construct and compile HLL source code on-the-fly to operate on the columnar batches.

Another important novelty is in the multi core execution of the TRILL operators and is heavily inspired by the map reduce model of computation. TRILL uses a streaming variant of the traditional map reduce architecture where in each computation unit runs in a different thread. TRILL performs stage execution of the operators and uses as much degree of parallelism during execution of each stage using the map-shuffle-reduce paradigm.

## 4.11 Conclusion of Related Work

We briefly discussed operator scheduling in various stream processing systems developed both by academia and industry. A major takeaway would be that some streaming systems, both batched and realtime, indeed try to exploit data parallelism on multicore architecture. For instance, Apache Storm's Nimbus scheduler creates multiple bolt instances called tasks, where task operates on a partition of the data, and these tasks run in parallel both among different nodes and among different executors in the same node. To put it simply, the decision of static partitioning to exploit data parallelism is taken at the global scheduler and deployed on different cores and threads on the nodes.