Scale-up Strategies for Processing High-Rate Data Streams in System S

Henrique Andrade† Buğra Gedik† Kun-Lung Wu† Philip S. Yu†
† IBM Research
T. J. Watson Research Center
University of Illinois, Chicago
{hcma,bgedik,klwu}@us.ibm.com psyu@cs.uic.edu

Abstract—High performance stream processing is critical in many sense-and-respond application domains – from environmental monitoring to algorithmic trading. In this paper, we focus on language and runtime support for improving the performance of sense-and-respond applications in processing data from high-rate live streams. The central tenets of this work are the programming model, the workload splitting mechanisms, the code generation framework, and the underlying IBM Research’s System S middleware and SPADE programming model. We demonstrate considerable scalability behavior coupled with low processing latency in a real-world financial trading application.

I. INTRODUCTION

Large scale sense-and-respond systems [1] continuously receive external streams from multiple sources and employ analytics aimed at detecting critical conditions and, ideally, responding in a proactive fashion. All these sense-and-respond systems share the need for (1) calculating baselines for multiple samples of incoming signals (e.g., the fair price of a security traded in the stock markets) as well as (2) the correlation of the computed value for a signal with other signals (e.g., instantaneous electricity consumption levels, the ask (or offer) price of a security, among others). The computation of baselines is typically performed by aggregating multiple samples based on a group-by aggregation predicate. Such an aggregation can be executed in different ways with different granularities by establishing a window over the incoming data – the sensing portion of a system. The correlation operation is typically the result of a join operation, followed by acting on the information (e.g., buying an underpriced security) – the responding portion of a sense-and-respond system.

The underlying architectural pattern representing these sense-and-respond streaming systems consists of a large number of window-based aggregation operations coupled in some fashion with a large number of window-based join operations operating on a collection of substreams. In our experience, the number of distinct substreams might not even be known a priori (e.g., securities may be added/removed from the market) and the logical substreams might be multiplexed in a single physical stream feed (e.g., a Reuters Stock Market Data Feed). Consequently, expressing such queries in relational stream processing algebra is often not possible, or very costly, due to the overhead created by the large number of resulting independent queries and the need for updating the set of queries as streams dynamically arrive and depart.

In this paper, we focus on the problem of optimizing the split/aggregation/join architectural pattern (defined in Section III). The essential challenge is in splitting the workload – one or more primal streams – and the actual processing (aggregation/join) carried out by the application to scale up as more computational resources are employed.

II. PROCESSING MULTIPLEXED INDEPENDENT SUBSTREAMS

The initial operation typically performed by stream processing systems is data ingestion. This operation relies on an edge adapter that converts a data feed of incoming packets into stream data objects (or tuples) for processing. Usually, a limited amount of data cleaning, data conversion, and data transformation is also performed during data ingestion.

An edge adapter may create one or more data streams as it may employ a channelization method [2], whereby a fat physical stream can be split into a collection of thinner streams, for example, using multiple UDP multicast groups. The tuples flowing on each of these streams are usually logically related (e.g., trade transactions of IBM stock).

In most cases, physical as well as logical channels carry messages/tuples that are associated with different groups. For example, in processing trading market data, a financial firm must acquire a market feed such as Bloomberg B-Pipe1. Assuming that a stock trading firm is interested only in trades from the NASDAQ stock exchange, one or more channels will be created and each will contain independent transactions. In this case, channels will be created for splitting the incoming traffic for load balancing (e.g., ticker symbols starting with A, B, and so on) or for categorically partitioning the traffic (e.g., biotech companies, optics company, etc). The important point here is that each of these channels (we shall refer to them from this point on as streams) contain data belonging to different groups. For example, a stream carrying transactions related to “ticker symbol starting with the letter A” will include trading data on a group of companies such as Agilent (A), Alcoa (AA), among others.

1Bloomberg B-Pipe is a real-time data distribution service providing access to more than 200 stock exchanges.
III. The Split/Aggregation/Join Architectural Pattern

We can now define the split/aggregation/join architectural pattern. Given a particular stream where data belonging to multiple groups is multiplexed together, a sense-and-respond system will initially demultiplex the incoming data into a collection of physical streams, then aggregate data from multiple groups while, at the same time, correlate (by joining) the aggregates with other data coming from the same or other groups. In the example above, we used different company stocks as groups, but this was arbitrary (albeit realistic). As we previously stated, the number of groups is not necessarily known beforehand – for example, a newly listed company may become part of the mining sector in the stock market or a particular stock may be traded only sporadically. This is an important aspect of this architectural pattern as we will see.

In terms of relational algebra, the implementation of these operations requires (1) filtering to be carried out by a selection operator to perform the demultiplexing, creating the substream for a particular group, (2) independent aggregation to be carried out by an aggregate function, and, finally, (3) joining substreams to be carried out by a join operator. Because we are performing the split/aggregation/join for different groups, we will have a collection of chains, each one for a different group. We emphasized the term collection, because a complication arises when the number of groups is not known beforehand. In this case, it is not possible to create the collection of independent query networks a priori.

The approach we delineated above based on run-of-the-mill relational operators suffers from two major shortcomings. First, as pointed out, one must know the number of groups a priori. The second, and a more fundamental flaw, is the fact that the query network grows with the number of groups. In other words, supporting a new group requires adding new selection operators, new aggregators, and new joins.

Given this situation, it is clear that for many applications, the scaling up costs can be steep. Interestingly, however, due to the independent processing of the different chains, one can see that the problem is embarrassingly parallel. Also it can be seen that the filtering that precedes the processing is performed on the group-by attribute. On the other hand, the windowing characteristics of both the Aggregation and the Join operator apply independently to each group. For example, the computation of the moving average trading price for the IBM stock over the last 20 transactions is independently triggered by the arrival of a new trade transaction of the IBM stock. Therefore, while a single Aggregate operator can be used for making the same computation for different groups in a typical relational query, aggregation (and join) operations in streaming scenarios typically rely on a window for determining when the operation is complete (e.g., the tuples received in the last 5 minutes for a time-based window, or the last 10 trade transaction tuples for a count-based window). Specifically, if a new trade on the IBM stock arrives, it triggers a change in the moving average for the IBM stock alone. The same reasoning applies to processing isolation necessary for stream Join operators.

That means that we need a mechanism for having the Aggregate and Join operators simultaneously operate on different groups in a compartmentalized fashion. In this scenario, the filtering can be done efficiently by simply hashing on the Aggregate group-by attribute and the Aggregate operator can independently compute the aggregations for the different groups as the windowing boundaries apply independently to the different groups. We achieved this by adding what we call a per group modifier to the windowing support in both the Aggregate and Join operators to SPADE (see Section ?? for additional detail).

Note that employing the per group modifier greatly reduces the number of operators that must be deployed, simplifying both the application physical deploying planning phase as well as the application execution management.

IV. System Infrastructure

We implemented the split/aggregation/join architectural pattern by extending the SPADE programming language and runtime support provided by System S, which we now describe in more detail.

System S [3], [4], [5] is a large-scale, distributed data stream processing middleware under development at the IBM T. J. Watson Research Center. It supports structured as well as unstructured data stream processing and can be scaled to a large number of compute nodes. The System S runtime can execute a large number of long-running jobs (queries) that take the form of data-flow graphs. A data-flow graph consists of a set of Processing Elements (PEs) connected by streams, where each stream carries a series of Stream Data Objects (SDOs). The PEs implement data stream analytics and are basic execution containers that are distributed over the compute nodes. The compute nodes are organized as a shared-nothing cluster of workstations (COW) or as a large supercomputer (e.g., Blue Gene). The PEs communicate with each other via their input and output ports, connected by streams. The PE ports as well as streams connecting them are typed.

SPADE [6] (Stream Processing Application Declarative Engine) is the declarative stream processing engine of System S. It is also the name of the declarative language used to program SPADE applications. SPADE provides a rapid application development (RAD) front-end for System S. Concretely, SPADE offers:

1. An intermediate language for flexible composition of parallel and distributed data-flow graphs. This language sits in-between higher level programming tools and languages such as the System S IDE or stream SQL and the lower level System S programming APIs.
2. A toolkit of type-generic built-in stream processing operators. SPADE supports all basic stream-relational operators with rich windowing semantics. It also seamlessly integrates built-in operators with user-defined ones.

3. A broad range of stream adapters. These adapters are used to ingest data from outside sources and publish data to outside destinations, such as network sockets, relational and XML databases, file systems, as well as proprietary platforms such as IBM Websphere Front Office, and IBM DB2 Data Stream Engine, etc.

SPADE uses a code generation to fuse operators into PEs. The PE code generator produces code that (1) fetches tuples from the PE input buffers and relays them to the operators within, (2) receives tuples from operators within and inserts them into the PE output buffers, and (3) for all the intra-PE connections between the operators, it fuses the outputs of operators with the inputs of downstream ones using function calls. In other words, when going from a SPADE program to the actual deployable distributed program, the logical streams may be implemented as simple function calls (for fused operators) to pointer exchanges (across PEs in the same computational node) to network communication (for PEs sitting on different computational nodes). This code generation approach is extremely powerful because through simple recompilation one can go from a fully fused application to a fully distributed one, adapting to different ratios of processing to I/O provided by different computational architectures (e.g., blade centers versus Blue Gene).

V. A CASE-STUDY APPLICATION: BARGAIN DISCOVERY

Many financial market data processing applications can be described based on the split/aggregation/join architectural pattern as they fit a mold where one must first build predictive models for asset pricing or risk management and, later, correlate model results with incoming, live data and, thus, drive a trading platform to execute sell or buy orders.

Our aim in defining a case-study application is to capture this pattern rather than accurately and closely mimic algorithm trading strategies. We focus on showing how one can make use of systems-oriented optimizations (e.g., workload partitioning and effective distributed processing placement strategies) to increase data processing rates. Two key metrics are of interest: (1) data ingestion throughput, measured at market feed ingestion point and (2) latency, which we measure to capture the delay imposed by the central computing portion in an application (e.g., how long does it take to update the moving average for IBM stock trades).

The specific application we designed ingests trade and quote (TAQ) data from a stock exchange. In particular, the data is a sequence of trade and quote transactions, where trade transactions are characterized by the price of an individual security and the number of securities that were acquired/sold (i.e., volume). Quote transactions can either be a bid or an ask quote. A bid quote refers to the price a market maker (i.e., a firm that trades securities) will pay to purchase a number of securities and an ask quote refers to the price a market maker will sell a number of securities for. We based our experiments on trade and quote transactions that took place in December 2005. In the dataset, quote transactions are around 8 times more common than Trade transactions. Second, there are around 3000 stock symbols for which there is market trading activities.

VI. EXPERIMENTAL EVALUATION

We evaluated the performance of the SPADE-based implementation (using the split/aggregation/join architectural pattern and per group windows) of the Bargain Discovery application seen in Figure 1, with respect to throughput and latency. The experiments presented in this section were performed on a subset of the System S cluster at Watson, using up to 16 nodes, where each node has two hyperthreaded 3GHz Intel Xeon processors and are connected together with a Gigabit Ethernet network. All of the values reported in our results represent the steady state runtime behavior of the application and are deduced from raw data collected via reservoir sampling [8] with a default buffer size of 5000 samples.

To achieve scalability, we increase the number of nodes used to execute the Bargain Discovery application. Note that all this is possible with a simple recompilation of the SPADE program! We report results for both distributed primal sources, that is, sources that contain non-overlapping content, and replicated primal sources, that is, sources that contain the same content, but come from different physical channels.

Distributed Primal Sources: Under the distributed primal sources model, the stream processing graphs routed at different sources receive non-overlapping portions of the source data. As a result, scalability with increasing number of compute nodes is easily achieved when the number of distributed primal

\[ \text{peak rate typically observed in our 2005 TAQ dataset is around 100,000 quotes and trades per second [7], with the average rates well below half of that figure. More recently, estimates for 2008 for market data rates are in the neighborhood of 1 million transactions per second, although this number is for options data (not TAQ) as we are using, but it highlights the need for scalability in stream processing systems dealing with financial data.} \]
sources is large. When the number of primal sources is small, the scalability mainly depends on SPADE’s performance in executing each chain, as well as splitting the source stream among multiple chains and nodes. In the financial trading domain, distinct feeds typically represent trading activity from different markets. In other words, feeds from different stock exchanges (e.g., NYSE, CME, LSE, etc). As a result, distributed primal sources occur naturally in this domain.

The graphs in Figure 2 plot the throughput (in tuples/sec) as a function of the number of nodes used to execute the Bargain Discovery application, for different number of distributed primal sources. In this setup, each node hosts four chains, one per processor. As observed for the 8 and 16 sources cases, we achieve close to perfect scalability when the number of primal sources is large (≈ 35K tuples/sec to ≈ 540K tuples/sec going from 1 node/1 source to 16 nodes/16 sources, i.e. a 15.5 speedup). Nevertheless, even when the number of sources is 4, SPADE provides good scalability (a 7-time speedup with 8 nodes and an 11-time speedup with 16 nodes). When we only have a single source, the scalability drops after 8 nodes, mainly due to the inefficiency of splitting a stream into more than 8 substreams on 8 nodes.

**Replicated Primal Sources:** Under the replicated primal sources model, the aim is to scale up with as few sources as possible, since larger number of sources imply more expense for receiving the exact same content through multiple distinct channels. In other words, there is only one market feed and it is being both replicated and split (and cost is incurred in doing so) such that the processing can be spread across nodes. This model is relevant in practice because trading platforms usually employ redundancy as a way of coping with failures.

The graph in Figure 3 plots the throughput (in tuples/sec) as a function of the number of nodes used to execute the Bargain Discovery application, for different number of replicated primal sources. In this setup, again each node hosts four chains, one per processor. The results are similar to the distributed sources scenario in terms of the general trends, with one significant difference: Regardless of the number of sources available, the speedup achievable is limited compared to the ideal case of linear speedup. This is due to Amdahl’s law [9], i.e., in the pipelined processing chain, the initial steps of data ingestion are inherently sequential, bounding the speedup that can be obtained by the remaining processing chain.

**VII. Concluding Remarks**

The split/aggregation/join architectural pattern is a common template for implementing stream processing applications in different domains. In many cases, such as in the financial domain, scalable and high-performance business logic translates directly into actual financial returns – the first one to spot a trading opportunity has the advantage. Therefore, the optimization of this architectural pattern is critical. In this paper, we have shown how features of System S and the SPADE programming language and compiler features can be used to achieve scalability and low latency.

We are now working on porting System S to run on Blue Gene. We believe that the Blue Gene networking characteristics coupled with SPADE’s ability to generate architecture-specific code will be specially suitable to support extremely high-data rate streaming applications.

**References**


