ARTICLE IN PRESS

J. Parallel Distrib. Comput. ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Processing high data rate streams in System S

H. Andrade [a,1], B. Gedik [a,*], K.-L. Wu [a], P.S. Yu [b]

[a] IBM Thomas J. Watson Research Center, Hawthorne, NY, 10532, United States

[b] Department of Computer Science, University of Illinois, Chicago, IL, 60607, United States

ARTICLE INFO

Article history:
Received 18 January 2010
Received in revised form
6 June 2010
Accepted 16 August 2010
Available online xxxx

Keywords:
Data stream processing
Scale-up strategies
Workload balancing
Split/aggregate/join architectural pattern

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 System S middleware and SPADE programming model. We demonstrate considerable scalability behavior coupled with low processing latency in a real-world financial trading application.

1. Introduction

Large-scale *sense-and-respond* systems [20] continuously receive external signals in the form of one or more streams from multiple sources and employ *analytics* aimed at detecting critical conditions and, ideally, responding in a proactive fashion. Examples of such systems abound, ranging from SCADA (Supervisory Control And Data Acquisition) systems deployed for monitoring and controlling manufacturing, power distribution, and telecommunication networks, to environmental monitoring systems, all the way to algorithmic trading platforms. All these sense-and-respond systems share the need for (1) calculating baselines for multiple samples of incoming signals (e.g., instantaneous electricity production levels, the fair price of a security, among others) 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 over different granularities by the establishment of a window over the incoming data. We refer to this first step as the *sensing* portion of a system. On the other hand, the correlation operation is typically the result of a *join* operation, where two signals are paired, generally using a window over the incoming data streams, and the result is used to drive an automated response whereby, for example, a request for the generation of extra power is made or a block of securities is sold or bought. This operation corresponds to the *responding* portion of a sense-and-respond system.

In many situations the number of signals to be independently aggregated and correlated is very high. For example, stock market feeds can contain information about trading for thousands of different securities — a financial firm processing and acting on information gleaned from the US equity market must track more than 3000 different stocks and an even larger number of derivatives on these stocks. Similarly, there are around 3000 power plants in the US [10] and millions of consumers. Streaming sense-and-respond systems must be able to cope with such a large influx of data.

In both examples, we argue that 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 distinct substreams. In our experience, in many cases, 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 [19]). Consequently, expressing such queries in relational stream processing algebra is often not possible, or is very costly, due to the overhead created by the large number of resulting independent queries, as well as the need for updating the set of queries as substreams dynamically arrive and depart.

In this paper, we focus on the problem of optimizing the split/aggregate/join architectural pattern (defined in Section 3). The

* Corresponding author.
E-mail addresses: hcma@us.ibm.com (H. Andrade), bgedik@us.ibm.com (B. Gedik), klwu@us.ibm.com (K.-L. Wu), psyu@cs.uic.edu (P.S. Yu).

[1] The author is currently with Goldman Sachs.

0743-7315/$ – see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2010.08.007

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. Among the contributions of this work, we highlight the following.

The architectural pattern characterization: We characterize an important streaming application architectural pattern. The definition of this architectural pattern has enabled us to design a programming framework, including a language, an optimization framework, and runtime support that enables application writers to focus on the application analytics as opposed to parallelization and distributed computing plumbing.

Language and code generation support: We identified a streaming application decomposition methodology as well as the compile-time knobs required for efficiently mapping a logical application onto physical resources.[2] The methodology hinges on the ways we provided the architecture in the language for partitioning the live data ingestion and processing workload into granular pieces (e.g., stream splitting, and what we call *per-group* aggregation and join operations) such that we can map the logical application as well as possible onto the underlying computational environment (e.g., by using code generation and operator-fusing techniques).

Comprehensive performance characterization: We implemented a realistic stock market trading application reliant on the split/aggregate/join pattern and provided a thorough experimental characterization of the per-group split/aggregate/join architectural pattern aimed at scaling up the application so that it can process half a million tuples per second on 16 cluster nodes.

In summary, this paper introduces the *split/aggregate/join* architectural pattern for stream processing systems, demonstrates how it is applied to *sense-and-respond* applications, and describes techniques to implement it efficiently using *per-group* processing. The overall approach is illustrated using a financial application, and an evaluation is performed on a distributed stream processing middleware using real-world workloads.

The rest of the paper is organized as follows. In Section 2, we discuss the nature of the high-performance processing of multiplexed independent substream problems. Section 3 discusses an approach for workload distribution based on the split/aggregate/join pattern. In Sections 4 and 5, we describe the stream processing platform used for carrying out the experimental evaluation and the SPADE stream processing programming language, respectively. In Section 6, we describe our case study application as an example of the split/aggregate/join streaming architectural pattern. Section 7 contains an extensive experimental analysis. In Section 8, we discuss the related work and contrast it with the ideas we present in this paper. And, finally, Section 9 summarizes this paper and discusses future planned extensions.

2. 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 *chanellization* method [27], 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). Another common approach is to employ a pub/sub system or *enterprise service bus* [8] to encapsulate and route the data from the physical feed to the downstream data consumers [6].

From the standpoint of stream processing, the difference between these two approaches lies in how much the original stream feed is split. With channelization technologies, the channels are in most cases created statically and one is typically limited to the number of channels that can be simultaneously supported by the middleware. Pub/sub technologies are much more dynamic as the existing subscriptions determine what really amounts to *logical* channels, as only messages matching the subscription predicates are routed to subscribers. Trade-offs between these approaches and hybrid ones that lie somewhere in between exist, but a longer discussion is outside the scope of this work.

Regardless of the approach used for ingesting physical streams, in most cases, physical as well as logical channels carry messages/tuples that are associated with different *groups*. To make this statement more concrete, let us again look at an example. In processing trading market data, a financial firm must acquire a market feed such as Bloomberg B-Pipe.[3] The market feed will then be ingested using one of the approaches that were delineated above. Assuming that the 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, logical or physical channels will be created for splitting the incoming traffic for load balancing (e.g., ticker symbols starting with A, B, and so on) or 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) contains 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 multiple companies such as Agilent (A), Alcoa (AA), among others. In this example, we refer to each company as a *group*, because trading analytics and activities will take place on stocks belonging to a particular company.

3. The split/aggregate/join architectural pattern

We can now define the *split*[4]/*aggregation*[5]/*join*[6] *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, correlating (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). Groups can contain collections, such as all companies that operate in the mining sector. 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

[2] Note that automatic optimization work is ongoing, but outside the scope of this paper. Nevertheless, the automatic optimization approach *does* use the fundamental processing decomposition instruments described in this work.

[3] Bloomberg B-Pipe is a real-time data distribution service providing access to more than 200 stock exchanges.

[4] A split operation is used to divide a stream into multiple ones to distribute the incoming traffic based on an application-specific predicate.

[5] An aggregation operation is used for grouping and summarization of incoming tuples over windows. For example, one can compute the average price of stock over the last 30 s.

[6] A join operation is used for correlating two streams. For example, a newsfeed with news items for a company can be correlated with streams carrying stock prices for that company. Streams can be paired up in several ways and the join predicate, i.e., the expression determining when tuples from the two streams are joined can be arbitrarily complex.

Please cite this article in press as: H. Andrade, et al., Processing high data rate streams in System S, J. Parallel Distrib. Comput. (2010), doi:10.1016/j.jpdc.2010.08.007

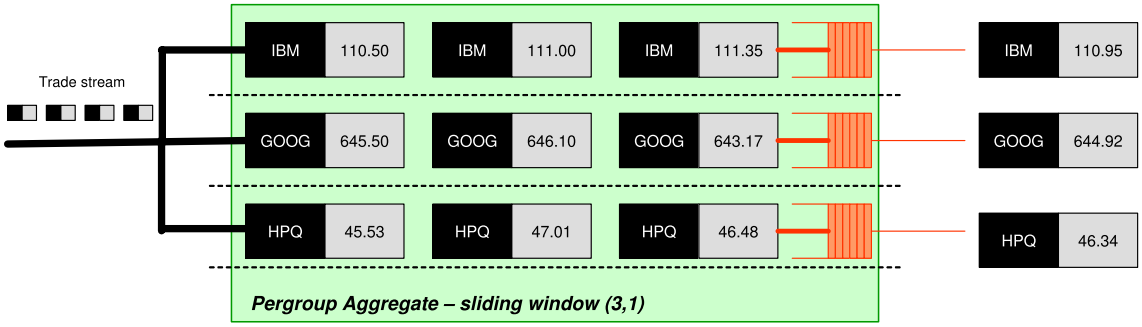

Fig. 1. A *per-group* aggregation operation. Moving averages are independently computed for different stock symbols inside the same Aggregation operator.

be traded only sporadically. This is an important aspect of this architectural pattern, as we shall soon 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* operation. Because we are performing the split/aggregate/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*.

In the following section we shall demonstrate how we address both the issue of coping with a very high number of independent chains as well as the issue of not knowing *a priori* how many chains there are.

3.1. The per-group modifier

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*, although it is possible to incrementally modify the query network as new groups are detected. 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 operators 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 min for a time-based window, or the last ten trade transaction tuples for a count-based window). A window is, however, intimately related to a group as it triggers additional processing or termination of processing based on the tuples that have been accumulated for one particular group. 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.

What is needed is the means 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 can achieve this by adding a *per-group* modifier to the windowing support in both the Aggregate and Join operators. Fig. 1 shows a *per-group* Aggregate operator. In this example, only three stock symbols are shown and a 3-trade transaction sliding window is depicted. These independent windows (maintained by the *per-group* version of the Aggregate operator) carries out a moving average computation for the stock price.

Effectively, this modifier logically creates multiple independent windows, one per group. The actual implementation employs the group-by attribute to segment the processing in the Aggregate operator as seen in Fig. 1 and the equijoin attribute to segment the processing for the Join operator. Note that this approach is applicable to both tumbling and sliding windows.[7]

In general, the problem of deploying an application query network depends on how to effectively distribute the individual processing chains on a stream data processing platform. Namely, the following questions must be solved at planning time: how many different stream engine containers[8] to employ, how many operators to run in each stream engine, how to perform internal operator scheduling within each engine, and how many nodes to employ for performing the computation. These questions are subjected to the underlying computational architecture hosting the stream processing system and are very critical as far as the overall application performance and scalability are concerned. Indeed, this is one of the critical problems facing the system infrastructure of trading firms. In Section 5, we will discuss how the code generation approach employed by the System S's SPADE compiler will help with these issues. As we will see, the *per-group* support is only as effective as how well one can distribute the processing load across processors and computational nodes.

Note that employing the *per-group* modifier greatly reduces the number of operators that must be deployed, reducing the

[7] Tumbling windows are operated on and then flushed when they become full (e.g., after having accumulated 20 trading transactions). Sliding windows on the other hand have two components: an expiration policy and a trigger mechanism. The expiration policy defines when accumulated tuples are ejected and, therefore, are no longer part of the internal state carried by an Aggregate operator (e.g., we keep the last 100 most recent tuples around). The trigger mechanism flags when the aggregation operation should take place (e.g., an aggregation should be made and output every time a new tuple is received by the operator).

[8] Most of the distributed stream engines currently available rely on containers on each of the nodes the system runs on for distributing the processing load.

Please cite this article in press as: H. Andrade, et al., Processing high data rate streams in System S, J. Parallel Distrib. Comput. (2010), doi:10.1016/j.jpdc.2010.08.007

overhead[9] of going across a different set of operator instances for each logical stream, reducing the memory footprint, and also simplifying both the planning phase and the query execution management. On the other hand, taking *per-group* processing to the extreme, where a single operator does all the aggregation (or join), has several disadvantages as well. First, such an operator becomes monolithic and may not be able to take advantage of multiple nodes. Second, multi-threading capabilities have to be built in manually as part of the operator implementation, in order to take advantage of multiple cores. The *per-group* modifier, on the other hand, enables one to adjust the number of parallel chains, where each chain has *per-group* processing taking place over its subset of logical streams. This enables us to achieve the best of both worlds: take advantage of intra-node and inter-node parallelism, while keeping the operator's internal implementation free of multi-threaded code.

In Section 7, we empirically show the improvements that can be obtained by employing the *per-group* modifier to create chains with different processing granularities.

4. The System S platform

System S [2,15,29] 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 tuples. 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 the streams connecting them are typed. PEs can be explicitly connected using hard-coded links or through implicit links that rely on type compatibility. The latter type of connections is dynamic and allows System S to support incremental application development and deployment. Besides these fundamental functionalities, System S provides several other services, such as fault tolerance, scheduling and placement optimization, distributed job management, storage services, and security, to name a few.

5. SPADE

SPADE [13] (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. SPADE offers the following.

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 and punctuation 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, and file systems, as well as proprietary systems such as IBM Websphere Front Office, and IBM DB2, etc.

5.1. Programming model

The SPADE language provides a stream-centric, operator-level programming model. The stream-centric design implies building a programming language where the basic building block is a *stream*. In other words, an application writer can quickly translate the flows of data he/she anticipates from a back-of-the-envelope prototype into the application skeleton, by simply listing the stream data flows. The second aspect, i.e., operator-level programming, is focused on designing the application by reasoning about the *smallest* possible building blocks that are necessary to deliver the computation an application is supposed to perform. The SPADE operators are organized in terms of domain-specific toolkits (e.g., signal processing, data mining, etc.). In most application domains, application engineers typically have a good understanding about the collection of operators they intend to use. For example, database engineers typically design their applications in terms of the operators provided by the (stream) relational algebra [4,24].

5.2. Compiler and runtime support

SPADE leverages the existing stream processing infrastructure offered by the Stream Processing Core (SPC) [2] component of System S. Given an application specification in SPADE's intermediate language, the SPADE compiler generates optimized code that will run on SPC as a native System S application, as seen in Fig. 2. SPADE's code generation and optimization framework enables it to fully exploit the performance and scalability of System S. The reliance on code generation provides the means for the creation of highly optimized platform-specific and application-specific code. In contrast to traditional database query compilers, the SPADE compiler outputs code that is very tailored to the application at hand as well as system-specific aspects such as the underlying network topology, the distributed processing topology for the application (i.e., where each piece will run), and the computational environment. In most cases, applications created with SPADE are long-running queries. Hence the long execution times amortize the build costs. Nevertheless, the SPADE compiler has numerous features to support incremental builds, reducing the build costs as well.

As we stated, SPADE uses 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*. This fusion of operators with function calls results in a depth-first traversal of the operator subgraph that corresponds to the partition associated with the PE, with no queuing involved in between. In other words, when going from a SPADE program to the actual deployable distributed program (seen in Fig. 2), the logical streams we see in Fig. 2 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

[9] While operators can be grouped together within the same process for performance considerations, there is still a performance hit due to the abstraction provided by an operator. For instance, SPADE provides a flexible framework to fuse operators. Even under fusion, every time a tuple travels through an operator port, the middleware performs certain bookkeeping, such as updating counters to track port statistics. Ports also serve as hook points for additional runtime services, such as profiling and debugging. While going from one operator to the next is much cheaper than going across the transport layer to a remote operator (on a different process or node), it is still not free.

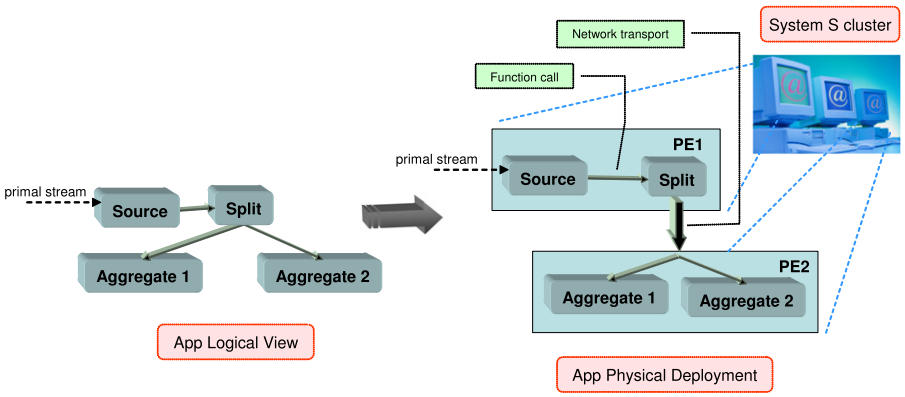

Fig. 2. A logical SPADE application (on the left) and its physical deployment (on the right). Application developers focus on the logical view and the SPADE compiler generates code appropriate to physical configuration parameters such as inter-connectivity infrastructure (e.g., cluster of workstations, SMP, etc.), number of nodes, type of nodes (e.g., x86, PowerPC, or Cell processors), among others.

Table 1
Trade and Quote (TAQ) data excerpt — showing trade and ask quote transactions.

Ticker symbol	Date	Timestamp	Transaction type	Price	Volume	Ask price	Ask size
MWG	30-DEC-2005	14:30:06.280	Trade	24.27	500	–	–
TEO	30-DEC-2005	14:30:06.283	Quote	–	–	12.85	1
UIS	30-DEC-2005	14:30:06.286	Quote	–	–	5.85	6
NP	30-DEC-2005	14:30:06.298	Trade	28.00	5700	–	–
TEO	30-DEC-2005	14:30:06.389	Trade	12.79	700	–	–

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). Currently, fusing is controlled through compiler directives and primitives in the SPADE code, but we are actively working on automatic planning techniques.

6. 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 points 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. A sample snippet of this data can be seen in Table 1, and the distribution of transactions per stock symbol is depicted in Fig. 4. 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). On the other hand, 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. However, a very small fraction of these stock symbols account for the bulk of the trading as one can easily see in the cumulative distribution function (CDF) plots. The implication here is that a substantial amount of imbalance can take place depending on how the processing is split. In Section 7, it will become clear that our approach to optimizing the split/aggregate/join architectural pattern is almost immune to imbalances.

Our sample application emulates a scenario where a securities trading firm makes money by quickly spotting *bargains* in the market. To identify a bargain, the firm first needs to acquire data to build a model for pricing all (or some of) the securities that are being traded. Once a security is priced (let us call it a *fair* price), the firm can assess whether the ask quotes are *mispriced*. That is, it must verify whether a seller is willing to sell that security (or a bundle of those) by a price that is lower than the *fair* price as predicted by the pricing model. The incoming primal stream carries all the information necessary for performing such algorithm.

Fig. 5 depicts a simplified view of the application in terms of SPADE's stream-relational operators. We omit the operators we included for instrumentation purposes for clarity. The primal stream is first split into two substreams — a Trade stream and a Quote stream, originating two separate processing chains. The Trade stream is used to feed the pricing model with recent trades. The pricing model employs a simple moving average approach for assessing the *fair* price for a security. This moving average, commonly referred to as VWAP (Volume-Weighted Average Price), is calculated by adding up the dollars traded for every transaction (price multiplied by number of shares traded) and then dividing by the total shares traded for a specific trading window. Typically, trading windows of different sizes are employed to capture long-term to short-term changes in pricing. In our particular implementation, we simultaneously compute the VWAP *for every single security* using three different window sizes — the last 5 Trade

```
for_begin @J 1 to NUMAGG

      stream VWAPAggregator@K_@L_@J(
        tickersymbol : String,
        svwap        : Double,
        svolume      : Double)
          := Aggregate(TradeFilter@K_@L <count(WSIZE*@J), count(1), pergroup>)[tickersymbol]
               { Any(tickersymbol), Sum(myvwap), Sum(volume) }
        -> partition["P@K_@L"], node(pool,@K)

      stream VWAP@K_@L_@J(
        tickersymbol : String,
        cvwap        : Double)
          := Functor(VWAPAggregator@K_@L_@J)[]
               { tickersymbol, (svwap/svolume)*100.0d }
        -> partition["P@K_@L"], node(pool,@K)

      stream BargainIndex@K_@L_@J(
        tickersymbol : String,
        bargainindex : Double)
          := Join( VWAP@K_@L_@J <count(1), pergroup>; QuoteFilter@K_@L <count(0)> )
               [{tickersymbol}={tickersymbol}, cvwap > askptickersymbole*100.0d]
               { tickersymbol := $1.tickersymbol,
                 bargainindex := exp(cvwap-askptickersymbole*100.0d)*asksize }
        -> partition["P@K_@L"], node(pool,@K)

for_end
```

per group modifier

per group modifier

operator-fusing and placement directives

Fig. 3. A SPADE code excerpt showing the VWAP calculation and the Bargain Detection implementation, employing the *per-group* construct and the operator fusion and placement features. The main body of the code is placed in a loop (with index @J) to create parallel chains for different aggregation window sizes (up to NUMAGG). Partitioning directives are used to co-locate all the operators under the same processing element, and placement directives are used to locate them on the same node. There are also two outer loops (with indices @K and @L), whose bodies are not shown for brevity, but their indices show up in the figure. These are used to further distribute the processing to multiple processing elements on a given node (using index @L) and to multiple nodes (using index @K).

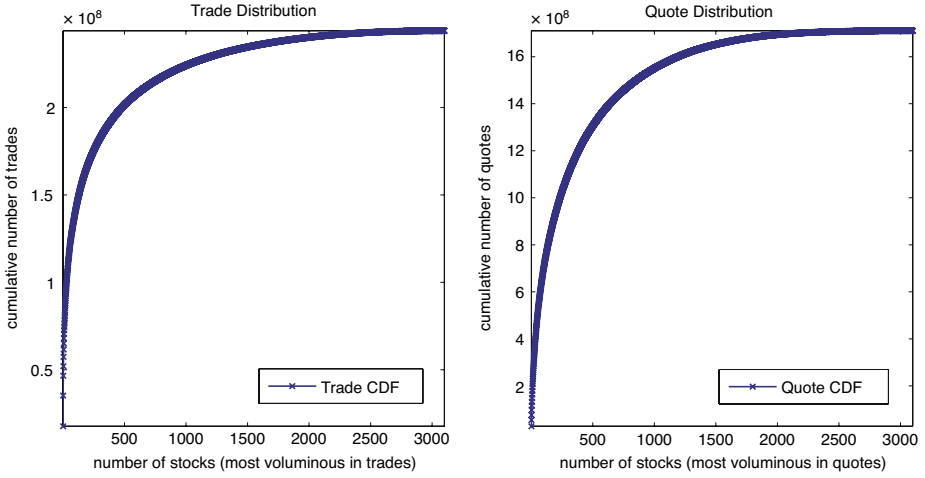

Fig. 4. Trade and Quote dataset characteristics. The cumulative distribution function plot shows that a small number of stock symbols account for a large volume of transactions.

transactions, the last 10, and the last 15. The SPADE code excerpt can be seen in Fig. 3. The actual VWAP computation is carried out using a sliding window, with a slide factor of 1, which means that three new VWAP computations are made every time a new Trade transaction comes in. During this portion of the computation, we have employed the SPADE *per-group* construct. As we have stated before, the application is running the pricing model for *all* securities that are being traded. In principle, a single *Aggregate* operator can carry out that computation as the *per-group* construct essentially creates different and isolated buckets for each different security, as we saw in Fig. 1. Later in this paper, we will also show that the *per-group* construct also enabled us to split the computation across different processing chains through simple hashing, replicating the processing chain for different groups of securities, achieving very good scalability.

The two processing chains are brought together by *Join* operators (one for each VWAP window size). The join operation is driven by the arrival of a new ask quote transaction. Its other input is fed by the most recently computed VWAP value. The SPADE Join operator can operate on windows and the windowing specification is unique to each input. Therefore, for a VWAP processing chain, a window of size 1 is employed (for keeping the last computed *fair* price for a security) and no windowing is employed for the Quote processing chain (i.e., a window of size 0) as we want to process the incoming Quote transactions as they arrive. Note that, again, the *per-group* construct is used in the *Join* operator as well to make sure that we can perform this correlation independently and simultaneously for every stock symbol, as also seen in Fig. 3. The *Join* operator also computes a *bargain index* which considers both the price difference between the quoted price and the *fair* price as well as the volume to be sold as specified by the Quote transaction. Therefore, a *good* bargain will either have a large price difference for a small volume or a small price difference for a large volume. Finally, the result of that computation is exponentially scaled up

Please cite this article in press as: H. Andrade, et al., Processing high data rate streams in System S, J. Parallel Distrib. Comput. (2010), doi:10.1016/j.jpdc.2010.08.007

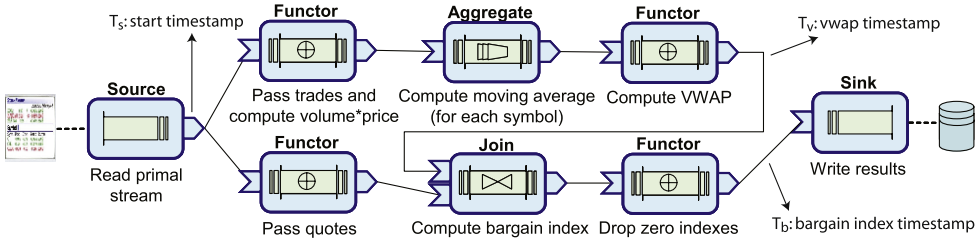

Fig. 5. The Bargain Discovery application described in terms of a topological graph of operators. Source and Sink are edge adapters (for receiving/exporting data from System S), Functors are used for filtering and data transformation, Aggregate operators are used for group-by operations, and Join operators are used for correlating data from different streams.

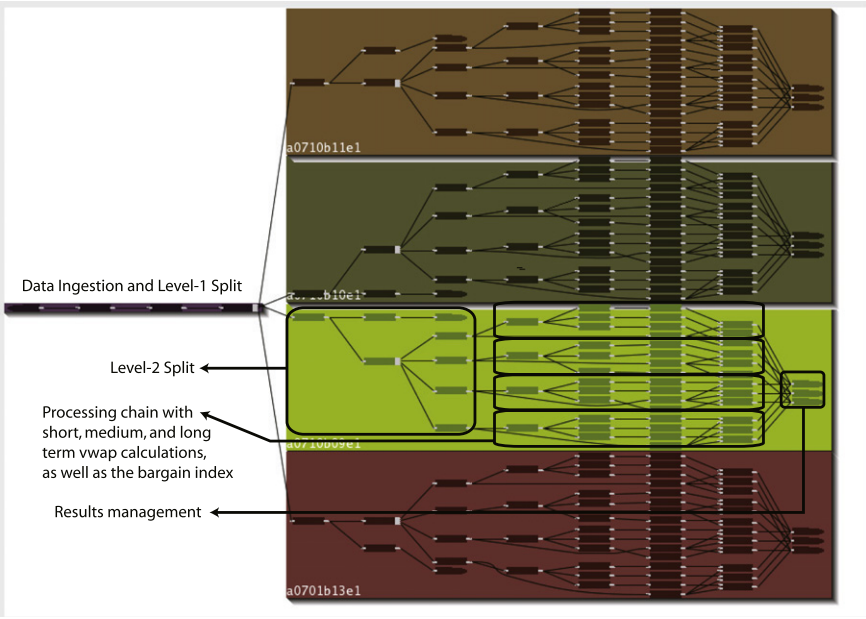

Fig. 6. The SPADE operator processing graph. The operator processing graph corresponds to the application writer's logical view – each small box corresponds to an operator and they are connected by logical streams.

to allow for ranking the bargains as they are dispatched to an automated trading platform. In our example, the application deposits the computed bargains in a file.[10]

In terms of performance characterization, the Bargain Discovery application can be employed to study two important metrics, which will be carefully dissected in Section 7. The first one, *rate of ingest*, measured at the *Source* tells us how many transactions can be ingested from a market feed.[11]

In general, data stream management systems employ load shedding techniques as one of the ways to keep up with possibly overloading rates of ingest [23]. However, for financial market applications it is important to consider the case where no load is shed, both for business, but primarily for regulatory reasons. Therefore, in this work, we consider the rate of ingest metric under a no load shedding policy. The second important metric is *latency* — how long a given message (or transaction) takes to traverse a query processing chain. In financial applications, latency is critical because it affects the core of the business — how long it takes to go from detecting a market opportunity to acting on it determines who is profitable. In our application, there are two important latencies to keep track of: VWAP computation and bargain detection. In Fig. 5, we show three instrumentation points T_s, marking the initial time stamp for a Trade or Quote transaction, T_v, marking when a Trade tuple originates a new VWAP computation, and T_b when a Quote tuple originates a new bargain index computation. The VWAP computation ($T_v - T_s$) latency is influenced by how large the aggregation window is, in addition to the actual computation that must take place. The bargain detection latency ($T_b - T_s$), arguably the most important one here, is influenced by the join algorithm only.

The actual implementation of the application in SPADE was slightly more complex than we described because we also added additional operators for instrumentation and scalability purposes (e.g., primal stream and second-level and third-level stream splitting) for multi-processor and multi-node workload and processing distribution. Figs. 6 and 7 show two depictions of the application processing graphs that were actually deployed. In both cases, we extracted the pictures using System S's visualization front end, called StreamSight [9]. StreamSight allows different visualization perspectives. We chose to show the application in terms of SPADE operators (Fig. 6) as well as in terms of actual processing elements (i.e., the runtime execution containers) (Fig. 7) — the logical and physical view as conceptually depicted in Fig. 2. Contrasting the two images allows one to observe the effects of operator fusion carried out by the SPADE compiler. In both cases,

[10] SPADE has a large number of edge adapters to stream the result out to other platforms. The *file* edge adapter is one of them.

[11] The *peak* rate typically observed in our 2005 TAQ dataset is around 100,000 quotes and trades per second [30], with the average rates well below half of that figure. More recently, estimates for 2008 for market data rates [11] 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.

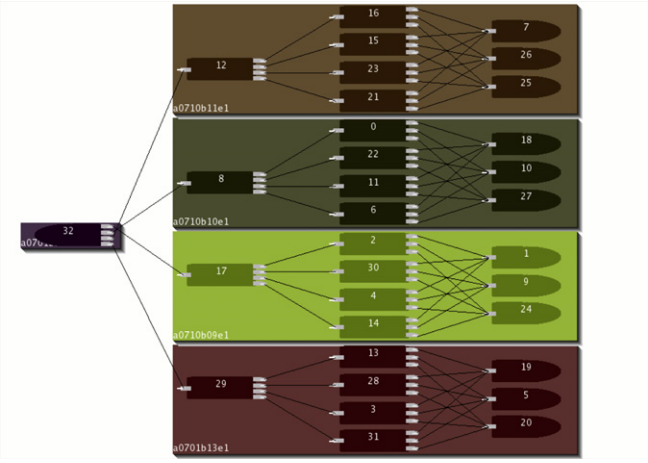

Fig. 7. The Processing Element (PE) processing graph. The PE processing graph corresponds to the physical deployment plan. In this case, several operators are fused inside a single processing element – each small box corresponds to a processing element and they are connected by physical streams. After fusion, many logical streams become function calls.

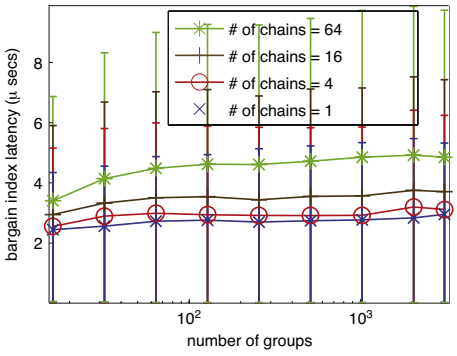

Fig. 8. Latency with 95% prediction interval as a function of the number of groups (stock symbols).

we showed a configuration where the primal stream is split four ways to feed independent processing chains residing on four distinct nodes.

7. Experimental evaluation

In this section, we evaluate the performance of the SPADE-based implementation (using the split/aggregate/join architectural pattern and per-group windows) of the Bargain Discovery application, 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 3 GHz Intel Xeon processors and is connected with a Gigabit Ethernet network. All of the values reported in the results represent the steady-state runtime behavior of the application and are deduced from raw data collected via reservoir sampling [26] with a default buffer size of 5000 samples.

7.1. Latency results

All latency results presented are from runs on a single node, using only a single processor. This is aimed at showcasing the advantage of using per-group modifier in terms of reducing the middleware overhead, even in the extreme cases where there are no parallelization opportunities.

The graphs in Fig. 8 plot the mean bargain detection latency, as well as its 95% prediction interval, as a function of the number

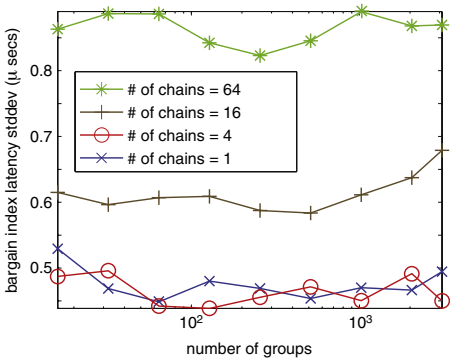

Fig. 9. Standard deviation of the latency as a function of the number of groups (stock symbols).

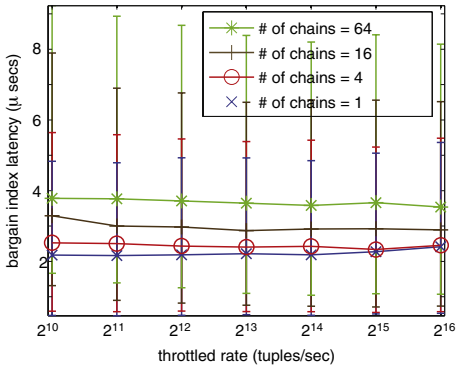

Fig. 10. Latency with 95% prediction interval as a function of the throttled input rate.

of groups in the stream, i.e. the number of stock symbols, for different number of processing chains used. Fig. 9 shows the standard deviation of the latency, as a function of the number of groups in the stream. Note that the per-group processing capability of SPADE provides us with the means to flexibly set the number of chains in our stream processing graph and thus distribute the processing load across processors (in the symmetric multiprocessing – SMP – sense) as well as across the COW nodes. Without the per-group processing capability, one has to resort to creating over 3000 chains (one per stock symbol) in order to run the Bargain Discovery application. We, on the other hand, are capable of bundling multiple stock symbols in a stream, and we process them in isolation using per-group Aggregate and Join operators. The amount of bundling can be tweaked to match the computational capabilities of a node. Note that for these experiments this was manually done — our automatic pre-planning optimization work is underway. In the next section, we study the benefits of the per-group modifier under highly parallel set-ups. For now, we return back to the single-processor scenario for studying the reduced processing overhead that results from using the per-group modifier.

The graphs in Fig. 10 plot the mean bargain detection latency and its 95% prediction interval, as a function of the throttled input rate,[12] for different number of processing chains used. Fig. 10 shows that 64 chains cause around 100% increase in the mean bargain index latency throughout the *x*-axis range (throttled rate). Similarly, the 95% prediction interval of the 64-chain scenario is around 50% larger compared to that of a single chain. Fig. 11 shows that the standard deviation also increases as the number of chains increases.

[12] The SPADE source edge adapter can be configured with throttling parameters for controlling the ingest data rate.

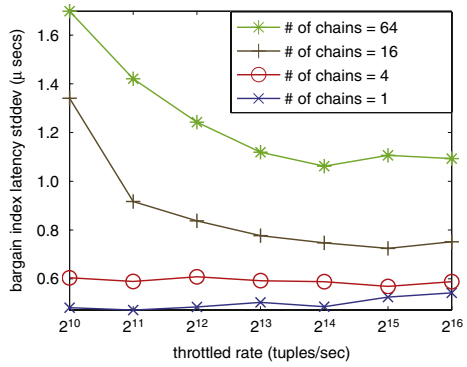

Fig. 11. Standard deviation of the latency as a function of the throttled input rate.

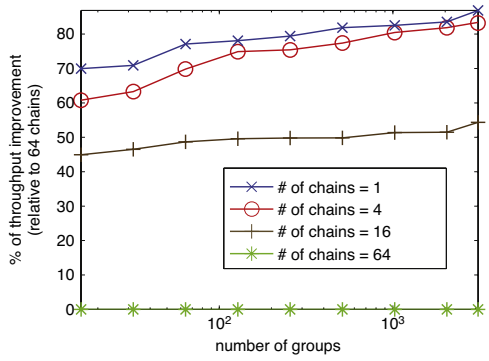

Fig. 12. Throughput as a function of the number of groups (stock symbols).

7.2. Throughput and scalability

In this section, we evaluate the performance in terms of throughput and scalability. We start with a single-node, single-processor scenario to illustrate the advantages of single-chain processing by using the per-group construct. We then extend the discussion to the multiple nodes and processors scenario and show how the flexibility provided by per-group windows and operators in setting the number of chains helps easily and effectively parallelize the Bargain Discovery application using the split/aggregate/join architectural pattern.

7.2.1. Single node and processor

The graph in Fig. 12 shows the relative increase in throughput, compared to the case of 64 chains on a single processor, as a function of the number of groups in the stream, i.e., the number of stock symbols, for different number of processing chains used. We observe that a single chain provides 70–87% higher throughput compared to 64 chains, whereas the improvement numbers are 60–80% for 4 chains and 45–55% for 16 chains. Moreover, the improvement we get by using fewer chains is more pronounced when the number of groups is larger, that is, when the overall processing is more costly. These results clearly illustrate that in a single-threaded environment, a single-chain implementation is superior and an implementation that does not rely on per-group support should be avoided, since it will require more than 3000 chains in the Bargain Discovery application, assuming that we know that we always have 3000 stock symbols (we do not necessarily know that).

7.2.2. Multiple nodes and processors

To achieve scalability, we increase the number of nodes used to execute the Bargain Discovery application, following the parallelization strategy discussed in Section 6. Note that all this is possible with a simple recompilation of the SPADE program. We report

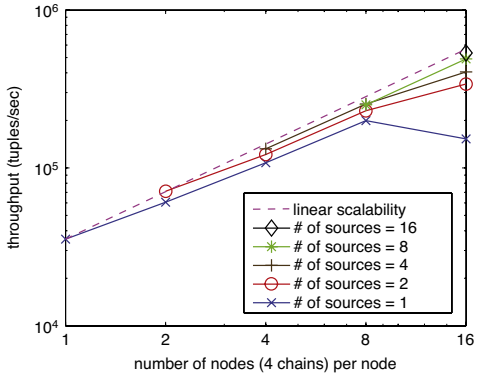

Fig. 13. Throughput as a function of the number of compute nodes, for different numbers of *distributed* sources.

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 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 Fig. 13 plot the throughout (in tuples/s) as a function of the number of nodes used to execute the Bargain Discovery application, for different number of distributed primal sources. In this set-up, 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 (≈35 K tuples/s to ≈540 K tuples/s going from 1 node/1 source to 16 nodes/16 sources, i.e. a 15.5 speed-up). Nevertheless, even when the number of sources is 4, SPADE provides good scalability (a 7-fold speed-up with 8 nodes and an 11-fold speed-up 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 a larger number of sources implies 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.

The graph in Fig. 14 plots the throughout (in tuples/s) as a function of the number of nodes used to execute the Bargain Discovery application, for different numbers of replicated primal sources. In this set-up, 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 speed-up achievable is limited compared to the ideal case of linear speed-up. This is due to Amdahl's law [16], i.e., in the pipelined processing chain, the initial steps of data ingestion are inherently sequential, bounding the speed-up that can be obtained by the remaining processing chain.

Please cite this article in press as: H. Andrade, et al., Processing high data rate streams in System S, J. Parallel Distrib. Comput. (2010), doi:10.1016/j.jpdc.2010.08.007

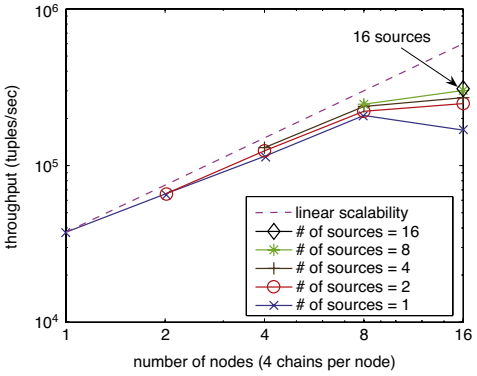

Fig. 14. Throughput as a function of the number of compute nodes, for different numbers of *replicated* sources.

To see this in more detail, let us consider the *ingest rate* and *processing rate* under the replicated primal sources model.

Let t_d be the time it takes to receive and drop a tuple from the primal source and, similarly, let t_f be the time it takes to receive and fully process a tuple. Thus, our single-node processing rate is $P_1 = \frac{1}{t_f}$, and similarly the single-node ingest rate (receive and drop everything) is $I_1 = \frac{1}{t_d}$. If we are to divide the job among n nodes assuming n replicated primal sources, then each node will process only $\frac{1}{n}$th of the tuples it receives, and will drop $\frac{n-1}{n}$th of them. As a result, the total aggregate tuple processing rate is given by

$$P_n = n \cdot (t_d \cdot (n-1) + t_f)^{-1}.$$

The speed-up relative to the single-node case is given by

$$S_n = n \cdot \left(\frac{t_d}{t_f} \cdot (n-1) + 1\right)^{-1}.$$

Given that $t_f > t_d$, we make two important observations.

1. The aggregate tuple processing rate achievable under the replicated primal sources model is bounded by the single-node ingest rate I_1, i.e. $\lim_{n\to\infty} P_n = I_1$.
2. The speed-up achievable under the replicated primal sources model is bounded by the single-node ingest rate to single-node processing rate ratio $\frac{I_1}{P_1}$, i.e. $\lim_{n\to\infty} S_n = \frac{I_1}{P_1} = \frac{t_f}{t_d}$.

Overall, we obtained very good results — with a small set of nodes, non-trivial data processing could be carried out for *all* Trade and Quote transactions at speeds that are five times the peak market feed rate we observed. More importantly, we showed how workload partitioning is trivially supported by the split/aggregate/join architectural pattern and, therefore, if additional processing is to be carried out, one can easily split the work across additional nodes.

8. Related work

The distributed computing area and its subarea of distributed stream processing have received a lot of recent attention. The availability of large-scale affordable computational infrastructure makes it possible to implement large, real-world continuous streaming data analysis applications.

First, let us consider the existing distributed computing middleware — the Parallel Virtual Machine (PVM) and the Message Passing Interface (MPI). While the architectural pattern we identified can be implemented in either one, there are advantages in doing it using SPADE. First, from an application writer's standpoint, a developer will concentrate on the analytics and not worry about distributed computing plumbing. As described in Section 5.2, the knobs for controlling the compile-time fusion of operators as well as placement of application components were used in this implementation and are the foundation for future automatic optimization approaches.

Second, in the relational data processing world, frameworks such as STREAM [3], Borealis [1], StreamBase [21], TelegraphCQ [7], among others, the focus is on providing stream processing middleware and, in some cases, declarative language for writing applications. Less focus is on the distributed and potentially large-scale nature of the problem and on ways of mapping the computation onto the underlying large-scale distributed environment. None of these systems give the developer the language constructs or the compiler optimization knobs to write the application in a granular way to truly leverage the levels of parallelism available in modern distributed architectures such as large-scale clusters as well as supercomputing platforms.

On the programming language side, StreamIt [22] is certainly closer to us. But its focus is on implementing stream flows for DSP-based applications. It really does not have a distributed computing underpinning. More recently, the Aspen language [25] shares some commonalities with SPADE — the most important being the philosophy of providing a high-level programming language, shielding users from the complexities of a distributed environment. Another similar approach is Pig Latin [18]. But many distinctions exist when contrasting them with our basic design principles. SPADE is organized in terms of high-level operators, forming toolkits (e.g., a relational algebra toolkit). Toolkits can be extended with additional operators and additional toolkits may be added, extending the language. Like Aspen, SPADE supports user-defined operators too.[13] Furthermore, the SPADE compiler generates code and, hence, can customize the runtime artifacts to the characteristics of the runtime environment, including architecture-specific and topological optimizations, as discussed in Section 5.2.

Finally, contrasting SPADE with Hadoop [14] – as one representative middleware supporting the map/reduce paradigm – or Maryland's Active Data Repository [17] and DataCutter [5], the key difference is the abstraction level employed for writing applications. These approaches rely on "low-level" programming constructs. Analytics are written from scratch as opposed to relying on built-in, granular, operators. Moreover, map/reduce operations can only be used for computations that are associative/commutative by nature.

9. Concluding remarks

The split/aggregate/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. Several features of System S and SPADE were particularly important.

(1) *Support for distributed stream processing*: The ability to deploy an application on a large number of processing nodes was critical in providing the means for being able to achieve scalability as we distributed the load across different processing chains and nodes.

[13] The important distinction here is that user-defined operators (UDOPs, in SPADE lingo), which are the fundamental building blocks in Aspen, are not fully templatized and, therefore, schema-agnostic. On the other hand, the SPADE built-in operators (BIOPs) are fully generic with respect to stream schemas.

(2) *Operator-based language*: From the standpoint of writing the application, developers typically think in terms of operators and how they interconnect (a common approach in other data analysis software platforms, such as general-purpose statistical packages, and simulation packages). In our present work, the support for operators and for operator fusing was critical for finding a physical deployment configuration that permitted us to fully utilize all the computational nodes.

(3) *Per-group operators*: The construct allowed us to radically simplify the application, reducing the number of operators — in particular, Join and Aggregate operators necessary to carry out the computation. Moreover, this construct allowed us to have the flexibility in breaking down the processing to any level of granularity that was adequate as far as fully utilizing the available computational nodes.

Despite our encouraging results, during the manual calibration and experimental evaluation of our case-study application, it became clear that many challenges lie ahead. The most important is in application compile-time pre-planning.[14] While SPADE allows one to write an application in terms of logical operators, the mapping between the logical view of the application to the physical layout is critical to performance. This is not a ground-breaking observation, and the SPADE compiler was designed with that in mind. In the present work, operator fusing and placement were driven by *directives* in the SPADE source code to illustrate the parameters an optimizer can tweak. In another recent study, we have designed and implemented a heuristic-based optimizer [12], illustrating how these parameters can be employed in an automatic optimization step which aims at deciding which operators should be fused. That work is the first step towards an overall compile-time optimization strategy that will employ all the knobs we manually tweaked in the course of the experimental study conducted in the present study.

Acknowledgments

We thank Nagui Halim, the System S principal investigator, for providing us with invaluable guidance throughout the development of SPADE. We also acknowledge Kevin Pleiter for providing guidance on algorithmic trading scenarios.

References

[1] D.J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A.S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, S. Zdonik, The design of the Borealis stream processing engine, in: Conference on Innovative Data Systems Research, CIDR, 2005.
[2] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, C. Venkatramani, SPC: a distributed, scalable platform for data mining, in: Workshop on Data Mining Standards, Services and Platforms, DM-SSP, Philadelphia, PA, 2006.
[3] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, J. Widom, STREAM: The Stanford stream data manager, IEEE Data Engineering Bulletin 26.
[4] A. Arasu, S. Babu, J. Widom, The CQL continuous query language: Semantic foundations and query execution, Tech. Rep., InfoLab — Stanford University, October 2003.
[5] M. Beynon, R. Ferreira, T. Kurc, A. Sussman, J. Saltz, DataCutter: middleware for filtering very large scientific datasets on archival storage systems, in: Proceedings of the 8th Goddard Conference on Mass Storage Systems and Technologies/17th IEEE Symposium on Mass Storage Systems, College Park, MD, 2000.
[6] A. Carzaniga, D.S. Rosenblum, A.L. Wolf, Design and evaluation of a wide-area event notification service, ACM Transactions on Computer System 19 (3) (2001) 332–383.
[7] S. Chandrasekaran, O. Cooper, A. Deshpande, M.J. Franklin, J.M. Hellerstein, W. Hong, S. Krishnamurthy, S.R. Madden, V. Raman, F. Reiss, M.A. Shah, TelegraphCQ: Continuous dataflow processing for an uncertain world, in: Conference on Innovative Data Systems Research, CIDR, 2003.
[8] L. Davidsen, Building an ESB without limits, ftp://ftp.software.ibm.com/software/integration/library/whitepapers/WSW11320-USEN-00.pdf, April 2008.
[9] W. De Pauw, H. Andrade, L. Amini, StreamSight — a visualization tool for large-scale streaming applications, in: Symposium on Software Visualization, ACM SoftVis, Herrsching am Ammersee, Germany, 2008.
[10] Energy Information Administration, Electric power industry overview, http://www.eia.doe.gov/cneaf/electricity/page/prim2/toc2.html, October 2007.
[11] J. Emigh, Morningstar detangles options data, http://www.windowsfs.com/TheMag/tabid/54/articleType/ArticleView/articleId/2185/, Morningstar-Detangles-Options-Data.aspx.
[12] B. Gedik, H. Andrade, K.-L. Wu, A code generation approach for optimizing high performance distributed data stream processing, in: International Conference on Information and Knowledge Management, CIKM, Hong Kong, China, 2009.
[13] B. Gedik, H. Andrade, K.-L. Wu, P.S. Yu, M. Doo, SPADE: the System S declarative stream processing engine, in: International Conference on Management of Data, ACM SIGMOD, Vancouver, Canada, 2008.
[14] Hadoop, http://hadoop.apache.org.
[15] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, C. Venkatramani, Design, implementation, and evaluation of the linear road benchmark on the stream processing core, in: International Conference on Management of Data, ACM SIGMOD, Chicago, IL, 2006.
[16] V. Kumar, A. Grama, A. Gupta, G. Karypis, Introduction to Parallel Computing — Design and Analysis of Algorithms, The Benjamin/Cummins Publishing Company, Inc., 1994.
[17] T. Kurc, C. Chang, R. Ferreira, A. Sussman, J. Saltz, Querying very large multi-dimensional datasets in ADR, in: Proceedings of the 1999 ACM/IEEE SC Conference, SC 1999, Portland, OR, 1999.
[18] C. Olston, B. Reed, U. Srivastava, R. Kumar, A. Tomkins, Pig Latin: a not-so-foreign language for data processing, in: International Conference on Management of Data, ACM SIGMOD, Vancouver, Canada, 2008.
[19] Reuters professional — stock market data feed, http://about.reuters.com/productinfo/s/stock_market_data_feed, April 2008.
[20] Caltech, Sensing and responding – Mani Chandy's biologically inspired approach to crisis management, ENGenious – Caltech Division of Engineering and Applied Sciences.
[21] StreamBase Systems, http://www.streambase.com.
[22] W. Thies, M. Karczmarek, S. Amarasinghe, Streamit: a language for streaming applications, in: International Conference on Compiler Construction, ICCC, Grenoble, France, 2002.
[23] Y.-C. Tu, S. Liu, S. Prabhakar, B. Yao, Load shedding in stream databases: a control-based approach, in: Very Large Data Bases Conference, VLDB, 2006.
[24] J.D. Ullman, Database and Knowledge-Base Systems, Computer Science Press, 1988.
[25] G. Upadhyaya, V.S. Pai, S.P. Midkiff, Expressing and exploiting concurrency in networked applications with aspen, in: Symposium on Principles and Practice of Parallel Programming, ACM PPoPP, San Jose, CA, 2007.
[26] J.S. Vitter, Random sampling with a reservoir, ACM Transactions on Mathematical Software 11 (1985) 37–57.
[27] IBM WebSphere front office for financial markets, http://www.ibm.com/software/integration/wfo, October 2007.
[28] J. Wolf, N. Bansal, K. Hildrum, S. Parekh, D. Rajan, R. Wagle, K.-L. Wu, L. Fleischer, SODA: an optimizing scheduler for large-scale stream-based distributed computer systems, in: ACM/IFIP/USENIX International Middleware Conference, Leuven, Belgium, 2008.
[29] K.-L. Wu, P.S. Yu, B. Gedik, K.W. Hildrum, C.C. Aggarwal, E. Bouillet, W. Fan, D.A. George, X. Gu, G. Luo, H. Wang, Challenges and experience in prototyping a multi-modal stream analytic and monitoring application on System S, in: Very Large Data Bases Conference, VLDB, 2007.
[30] X. Zhao, X. Zhang, T. Neylon, D. Shasha, Multiple query processing in deductive databases using query graphs, in: 2005 Database Engineering and Application Symposium, IDEAS 2005, 2005, pp. 3–14.

[14] We are referring to compile-time planning here. Note that System S also includes a runtime scheduling and planning component called SODA [28].

H. Andrade has been a Research Staff Member in the Exploratory Stream Processing Systems group at IBM Watson (Hawthorne) since July 2004 and an adjunct assistant professor at Columbia University since January 2010. He previously held a post-doctoral research associate position at the University of Maryland, College Park, was a consultant for the Department of Biomedical Informatics at the Ohio State University, and an adjunct assistant professor at the University of Maryland, University College. He obtained his Ph.D. in Computer Science in 2002 at the University of Maryland, College Park. He also holds two Master's degrees in Computer Science (Federal University of Minas Gerais, 1997, and University of Maryland College Park, 1999). More recently, in 2008, he obtained a graduate Certificate in Finance Engineering from Columbia University in New York. He received a Best Student Paper Award at ACM/IEEE Supercomputing 2002, a Best Paper Award at ACM Software Visualization 2008, and has filed over 25 patent disclosures. His research interests are in the area of high-performance computing, in

particular, middleware technologies and optimization techniques for data analysis applications.

B. Gedik is with the IBM Thomas J. Watson Research Center, where he is currently a member of the Data-Intensive Systems and Analytics Group. His current research interests are on large-scale distributed stream processing systems. He is particularly looking into language, compiler, and runtime design, as well as profiling, optimization, and debugging issues in large-scale stream processing systems. Dr. Gedik has served on organizational committees of several international conferences and workshops, including the International Workshop on Scalable Stream Processing Systems (General Co-Chair, SSPS'07), International Conference on Distributed Event-based Systems (PC Co-Chair, ACM DEBS'09), International Conference on Collaborative Computing: Networking, Applications and Worksharing (PC Co-Chair, CollaborateCom'07), International Workshop on Distributed Event Processing, Systems and Applications (PC Co-Chair, DEPSA'07). He is the recipient of the 2003 International Conference on Distributed Computing Systems (ICD CS) Best Paper Award. Dr. Gedik received his B.S. degree in Computer Science from Bilkent University, Ankara, Turkey, and his Ph.D. degree in Computer Science from Georgia Institute of Technology, Atlanta, US.

K.L. Wu received his B.S. degree in Electrical Engineering from the National Taiwan University, Taipei, Taiwan, and his M.S. and Ph.D. degrees in Computer Scince from the University of Illinois at Urbana-Champaign. He is the manager of the Data-Intensive Systems and Analytics Group at IBM T.J. Watson Research Center. He is also the development manager of the InfoSphere Language and Developer Tools Team, Information Management, IBM Software Group. The combined research and development team is currently engaged mainly in the product development of System S — a large-scale distributed stream processing system — focusing on the programming language and compiler, the integrated development environment (IDE), and the administration/configuration/installation tools. The team also conducts a wide range of research issues in data-intensive systems and analytics — including programming languages and models for distributed stream processing; advanced analytic algorithms for stream applications; job management and scheduling, resource management and system. Dr. Wu is a Fellow of the IEEE and a member of the ACM. He was an Associate Editor for the IEEE Transactions on Knowledge and Data Engineering, 2000–2004. He was the general chair for the 3rd International Workshop on E-Commerce and Web-Based Information Systems (WECWIS 2001). He has served as an organizing and program committee member on various conferences. He has received various IBM awards, including the IBM Corporate Environmental Affair Excellence Award, Outstanding Technical Achievement Award, Research Division Award and 14 Invention Achievement Awards. He received a Best Paper Award from IEEE EEE 2004. He was also an IBM Master Inventor. He has published extensively in various journals and conferences. He also holds or has applied for many patents.

P.S. Yu is a Professor in the Department of Computer Science at the University of Illinois at Chicago, and he also holds the Wexler Chair in Information and Technology. He was manager of the Software Tools and Techniques group at the IBM Thomas J. Watson Research Center. Dr. Yu has published more than 500 papers in refereed journals and conferences. He holds or has applied for more than 300 US patents. Dr. Yu is a Fellow of the ACM and of the IEEE. He is associate editors of ACM Transactions on the Internet Technology and ACM Transactions on Knowledge Discovery from Data. He is on the steering committee of the IEEE Conference on Data Mining and was a member of the IEEE Data Engineering steering committee. He was the Editor-in-Chief of IEEE Transactions on Knowledge and Data Engineering (2001–2004), an editor, advisory board member and also a guest co-editor of the special issue on mining of databases. He had also served as an associate editor of Knowledge and Information Systems. In addition to serving as program committee member on various conferences, he was the program chair or co-chairs of the IEEE Workshop of Scalable Stream Processing Systems (SSPS'07), the IEEE Workshop on Mining Evolving and Streaming Data (2006), the 2006 joint conferences of the 8th IEEE Conference on E-Commerce Technology (CEC'06) and the 3rd IEEE Conference on Enterprise Computing, E-Commerce and E-Services (EEE'06), the 11th IEEE Intl. Conference on Data Engineering, the 6th Pacific Area Conference on Knowledge Discovery and Data Mining, the 9th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, the 2nd IEEE Intl. Workshop on Research Issues on Data Engineering: Transaction and Query Processing, the PAKDD Workshop on Knowledge Discovery from Advanced Databases, and the 2nd IEEE Intl. Workshop on Advanced Issues of E-Commerce and Web-based Information Systems. He served as the general chair or co-chairs of the 2006 A CM Conference on Information and Knowledge Management, the 14th IEEE Intl. Conference on Data Engineering, and the 2nd IEEE Intl. Conference on Data Mining. He has received several IBM honors including two IBM Outstanding Innovation Awards, an Outstanding Technical Achievement Award, two Research Division Awards and the 93rd plateau of Invention Achievement Awards. He was an IBM Master Inventor. Dr. Yu received a Research Contributions Award from IEEE Intl. Conference on Data Mining in 2003 and also an IEEE Region 1 Award for "promoting and perpetuating numerous new electrical engineering concepts" in 1999. Dr. Yu received his B.S. Degree in Electrical Engineering from National Taiwan University, his M.S. and Ph.D. degrees in Electrical Engineering from Stanford University, and his M.B.A. degree from New York University.

Please cite this article in press as: H. Andrade, et al., Processing high data rate streams in System S, J. Parallel Distrib. Comput. (2010), doi:10.1016/j.jpdc.2010.08.007