Spreadsheets for Stream Processing with Unbounded Windows and Partitions

Martin Hirzel Rodric Rabbah Philippe Suter Olivier Tardieu Mandana Vaziri
IBM T.J. Watson Research Center
{hirzel,rabbah,psuter,tardieu,mvaziri}@us.ibm.com

ABSTRACT

Stream processing is a computational paradigm that allows the analysis of live data streams as they are produced. This paper describes a programming model, based on enhancements to spreadsheets, that enables users with limited programming experience to participate directly in the development of complex streaming applications. The programming model augments a conventional spreadsheet with streaming features that permit operating over unbounded data sets despite the finite interface provided by the spreadsheet. The new constructs include time-based windows and partitioning. We introduce a spreadsheet compiler that generates C++ code to achieve integration with existing stream processing systems. Our experimental study illustrates the expressivity of the new features and finds that our implementation is between 8x slower and 2x faster than hand-written stream programs.

CCS Concepts

•**Software and its engineering** → **Data flow languages;**

Keywords

Spreadsheets; end-user programming; stream processing

1. INTRODUCTION

Continuous data streams arise in many different domains: finance, health care, telecommunications, and transportation, among others. Stream processing is a computational paradigm that allows the analysis of these data streams as they are being produced. This is necessary since immediate insights are more valuable than delayed insights, and since there is often too much data to efficiently persist to disk for offline analysis. Domain experts often have limited programming experience and must rely on developers to implement their models. Our objective is to bridge the gap between domain experts and developers by enabling the former to participate directly in the development of complex streaming applications. In doing so, they can apply their domain knowledge to evolve, refine, and customize data analysis.

Business analysts, scientists, and researchers commonly use spreadsheets for data collection, analysis, and reporting. The versatility and ease-of-use of spreadsheets enables millions of users worldwide [19, 21] to organize their data in two dimensions, perform analysis by applying operations and transformations, and use built-in charting capabilities to visualize and present their models and findings.

Unfortunately, two-dimensional spreadsheets fall short on two wide-spread requirements for streaming applications. First, since recent data tends to be more relevant than older data, most streaming applications use some form of time-based sliding windows. And second, since streams tend to have a partitioning key, most streaming applications analyze streams independently by partition. Connectors and adapters that ingest data from external sources into a spreadsheet are necessary. But they are not sufficient for most streaming applications if the organization of that data is limited to a fixed number of rows and columns.

With two dimensions, it is challenging to connect the spreadsheet to a live data stream of stock quotes, for example, and compute the average stock price over a sliding window of time (e.g., 5 minutes worth of data) — since it is impractical to know a priori how many cells to allocate for a time-based window. In a spreadsheet programming model, a function to compute the average value of a range of cells requires a static bound on the range. A *time-based window* however defines a variable-sized range with indeterminate updates to the range as data is streamed from a live source. Such variability affects both the usability of the spreadsheet and the computation ordering. While it is common for live analysis of data to perform aggregations and analysis over time-based windows, it is difficult to map these concepts naturally into typical spreadsheets interfaces.

Similarly, it is equally challenging to use a two-dimensional spreadsheet to compute the average trade price for *all* stock symbols that might appear in the live stream since this too may be a large and dynamic number of symbols. One might *partition* the stock stream by symbol, allocate a different sheet for each symbol, and repeat their analysis per sheet. However this is tedious, and unless all symbols are known in advance, it is not a practical solution.

This paper extends the expressivity of the spreadsheet programming model to overcome the static and finite nature of the spreadsheet interface, and thus, support general streaming applications. Our approach treats windows as first-class constructs and decouples windows from their graphical representation. An individual cell can represent a

DEBS '16, June 20-24, 2016, Irvine, CA, USA

DOI: http://dx.doi.org/10.1145/2933267.2933607

time-based window that is variable in size and specified using a duration. We also provide a mechanism for specifying partitions, that is, performing the same computation (e.g., average stock price) for different keys (e.g., stock symbols) of a possibly large or unknown data set. The user specifies the partitioning criterion, the spreadsheet user interface displays a debug view showing the computation for one user-selected key, and the server performs the same computation separately for all keys. In this way, we enable processing variable-sized and unbounded data sets that are inherent to time-based windows and partitions within an otherwise conventional spreadsheet. This espouses all the benefits of the spreadsheet interface and its ease of use while providing powerful features for modern day streaming analysis.

We implement our enhanced spreadsheet programming model by extending ActiveSheets [42]. In that paper, a spreadsheet ingests streaming data, and the user can organize and analyze that data as they would a static spreadsheet. That paper however did not overcome the challenges of dynamic-sized windows and partitions, and remained restricted to the finite nature of the conventional spreadsheet. Whereas the previous paper implemented streaming spreadsheets as an interpreter only, this paper implements them as a compiler instead to achieve superior performance. The compiler and its lightweight runtime system make it possible to integrate a spreadsheet into existing streaming applications via a simple native interface. Our spreadsheet compiler generates C++ code that implements the logic encoded in the spreadsheet and supports time-based windows and partitions. It also includes a number of optimizations important to make the implementation of the programming model efficient.

To demonstrate the versatility of our programming model, we have integrated spreadsheets to execute natively in two existing systems: Node-RED [32] and IBM Streams [22]. The former is a platform for creating "Internet of Things" workflows that stream and process data between hardware devices, RESTful online services, and arbitrary user code written in JavaScript. The latter is an industry-strength platform for high-speed data analytics.

We implemented a number of example benchmarks to evaluate the expressive power of our spreadsheet programming model and its performance implications. We found that it is both easy to express and work with time-based windows and partitions within a spreadsheet interface (we used Microsoft Excel). The evaluation shows that our optimization techniques provide up to 2x speedup for spreadsheets that use time-based windows compared to IBM Streams, and results are no worse than 8x slower for other benchmarks.

The contributions of this paper are:

- An extension of the spreadsheet programming model to support dynamic and variable sized time-based windows and partitions (Section 2).
- Formal semantics for the new features (Section 3).
- A spreadsheet compiler with optimizations specific to time-based windows and a lightweight runtime for native spreadsheet execution (Section 4).
- An experimental study to demonstrate the expressivity of the new features and the efficiency of the output of our compiler implementation (Section 5).

This paper provides spreadsheet extensions for stream processing and working with unbounded data sets arising

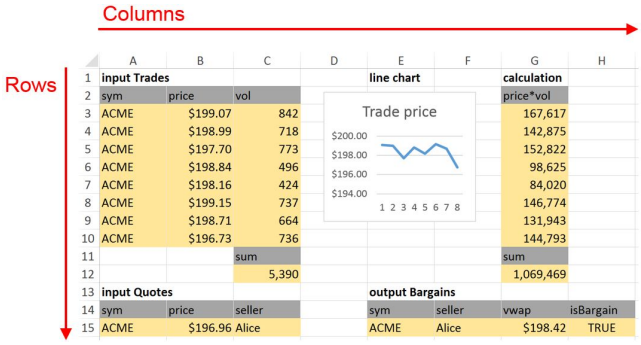

Figure 1: Conventional streaming spreadsheet.

from time-based windows and partitions. The resulting programming model is easy to use for non-programmers, with an implementation that makes it usable with modern stream processing systems.

2. OVERVIEW

This section gives an overview of our approach using a streaming bargain calculator as a running example. The calculator consumes two input streams: Trades contains the transactions that have been made; Quotes contains quoted stock prices. A stream is a series of tuples consisting of named attributes. For each quoted price, the objective is to determine whether or not it is a bargain by comparing it to an average obtained using recent trades. The average we consider is the volume-weighted average price (VWAP). Given a window of prices P_i and volumes V_i, the VWAP is defined as:

$$\text{VWAP} = \frac{\sum_i P_i \times V_i}{\sum_i V_i}$$

After computing the VWAP over Trades, the bargain calculator joins that with Quotes to determine whether or not each quoted price is less than the VWAP. If yes, it outputs a bargain. This computation needs to be done for each stock symbol that appears in the Quotes stream, and averages are taken over five-minute windows.

In a conventional streaming spreadsheet [42], there is a bounded number of cells available for data layout and computation. Figure 1 shows a simplified bargain calculator with averages taken over a small fixed-size window and only for a single stock symbol (ACME). Cells A3 through C10 contain incoming tuples from input Trades that are updated live. Cells A15 through C15 show the Quotes input. Cells G3 through G10 contain the computation of the price times the volume, and cells C12 and G12 aggregate the volume and price times volume, respectively. Finally, G15 shows the result of the VWAP computation and H15 whether or not the quoted price is a bargain.

Figure 1 over-simplifies the bargain calculator application in two important ways. First, aggregations can only happen over graphical windows in the spreadsheet, so the programming model is not sufficiently expressive to support windows defined as five minutes worth of data. Time-based windows can be very large, variable in size, and may require an unbounded space on the spreadsheet. Second, Figure 1 is only computing bargains for one stock symbol. If Trades contains many different symbols, the user would have to select each

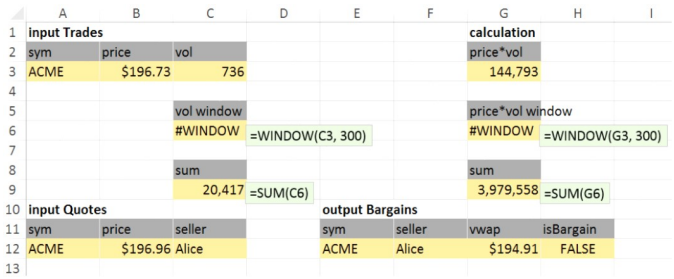

Figure 2: Time-based windows and partitioning.

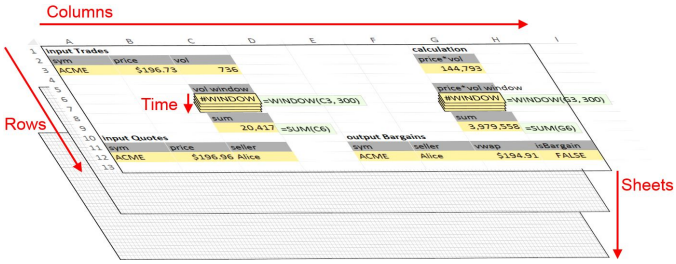

Figure 3: New spreadsheet dimensions: time and partitioning key.

and perform similar bargain calculations, requiring a potentially unbounded space on the spreadsheet. To make matters worse, these symbols may not be known in advance. This paper overcomes the finite nature of the spreadsheet, by augmenting the language with support for time-based windows, and a construct, called *partition*, that allows specifying a computation *for all* elements of a data set.

Figure 2 shows the running example in the augmented language. In this figure, the spreadsheet is *partitioned* according to the stock symbol (the *partitioning key*), which means that all computations are done for a sample symbol (in this case, ACME). The runtime engine performs the same computation for all symbols. Partitioning is specified as part of the meta-data for the spreadsheet and does not appear in it directly. The results of all these computations are recombined—merged in order—by the runtime engine to form the output stream of the spreadsheet. As a convention, each sheet allows a single partitioning key. The user may employ additional sheets to work with other keys (e.g., currency). In Figure 2, cells A3 through C3 contain the input Trades, and A12 through C12 the input Quotes, partitioned by key sym.

The formula WINDOW(C3,300) in C6 means that this cell represents a window of past values that have appeared in C3. Every tuple that appears in the spreadsheet has an implicit timestamp. The window contains those whose timestamps are within 300 seconds of the last time that the value of C3 changed. When a new value comes into C3, then windows on that cell are updated. Similarly, cell G6 is a time-based window of price times volume. The VWAP and bargain calculations are done as before.

Figure 2 contains fewer cells to specify the calculation of VWAP compared to the conventional streaming spreadsheet of Figure 1. Despite its conciseness, it is more powerful and expressive as illustrated in Figure 3. This figure shows how the new constructs add two new dimensions to the spreadsheet. Time-based windows contain an unbounded set of values using a single cell as its representation. The choice of a partitioning key results in an unbounded number of sheets that independently perform the same computation for all the different values of the key.

Streaming spreadsheets can be used as nodes or operators in a larger stream processing system. To this end, they must provide adequate performance in the presence of time-based windows and partitioning. We have implemented a compiler from spreadsheets to C++ that incrementalizes computation over windows. We demonstrate that the performance of our implementation permits using spreadsheets with modern stream processing systems.

3. FORMAL SEMANTICS

This section formalizes our enhancements to streaming spreadsheets. We write *spreadsheet* to mean a streaming spreadsheet enhanced with our new constructs. The design objectives for our work are as follows. Spreadsheets should capture many real-world streaming use cases by offering time-based windows and partitioning. But at the same time, the behavior of conventional spreadsheet features should be compatible with off-the-shelf spreadsheets, such as Excel. Furthermore, the semantics should be clear, deterministic, and efficiently implementable.

To meet these objectives, we formalized the semantics. The formalization is based on our prior work [42], but with new contributions to support windows and partitions. Windows are supported via a new WINDOW construct. Partitions are supported by revising the semantics of the existing ActiveSheets constructs (but note that this paper contains a self-contained description). Formalizing the semantics makes it possible to establish important properties: partition isolation (which enables semantics-preserving data parallelism) and incrementality (which limits storage overhead and redundant recomputation). Section 4 presents an optimizing compiler that implements the semantics.

3.1 Overview

Streams are sequences of values with timestamps. Values can either be scalars (e.g., numbers or strings) or windows (i.e., sequences) of values. Notice that for the formalization, streams of tuples can be encoded as multiple streams—one per attribute of the tuple—with the same timestamp.

A spreadsheet is a collection of cells containing formulas. Formulas combine references to input streams and cells—cyclic references are discussed below—in order to compute new streams using spreadsheet functions such as arithmetic, filtering, windowing, and aggregation functions. These computed streams are the output streams of the spreadsheet.

Spreadsheets are reactive agents. A formula is (re)computed only when one of the input streams or cells it refers to is updated, i.e., arrival of a new value on a stream or recomputed cell. Values are persistent. A cell retains the last computed value until the next computation.

Spreadsheets are synchronous agents. They adopt the approach of synchronous languages [8]. The timestamp attached to the value computed by a formula is simply the timestamp of the event that triggers the (re)computation, with no delay.

A spreadsheet has one special input stream: the partitioning stream or key (e.g., sym of the Trades input in Figure 2), possibly constant if partitioning is not needed. All input

$$v ::= \ell \mid \{\ell_1^{t_1}, ..., \ell_n^{t_n}\} \qquad f ::= s_0 \mid \phi(c_1, ..., c_n) \mid \mathsf{SELECT}(c_0, c_1) \mid \mathsf{WINDOW}(c_0, \ell_0, c_1) \mid \mathsf{PRE}(c_0, c_1, \ell_0)$$

$$\mathcal{T}_{s:\ell}(c,t) = \begin{cases} \{t_0 \in \mathrm{dom}(s_0) \cap [0,t] | s(t_0) = \ell\} & \text{if } c \mapsto s_0 \\ \bigcup_{i=1}^{n} \mathcal{T}_{s:\ell}(c_i, t) & \text{if } c \mapsto \phi(c_1, ..., c_n) \\ \{t_0 \in \mathcal{T}_{s:\ell}(c_1, t) | \mathcal{E}_{s:\ell}(c_1, t_0) = \mathsf{TRUE}, \mathcal{T}_{s:\ell}(c_0, t_0) \neq \emptyset\} & \text{if } c \mapsto \mathsf{SELECT}(c_0, c_1) \\ \mathcal{T}_{s:\ell}(c_1, t) & \text{if } c \mapsto \mathsf{WINDOW}(c_0, \ell_0, c_1) \\ \mathcal{T}_{s:\ell}(c_1, t) & \text{if } c \mapsto \mathsf{PRE}(c_0, c_1, \ell_0) \end{cases} \qquad \mathrm{deps}(c) = \begin{cases} \emptyset \\ \{c_1, ..., c_n\} \\ \{c_0, c_1\} \\ \{c_0, c_1\} \\ \{c_1\} \end{cases}$$

$$\mathcal{E}_{s:\ell}(c,t) = \begin{cases} s_0(\max(\mathcal{T}_{s:\ell}(c,t))) & \text{if } c \mapsto s_0 \text{ and } \mathcal{T}_{s:\ell}(c,t) \neq \emptyset \\ \phi(\mathcal{E}_{s:\ell}(c_1,t), ..., \mathcal{E}_{s:\ell}(c_n,t)) & \text{if } c \mapsto \phi(c_1, ..., c_n) \text{ and } \mathcal{T}_{s:\ell}(c,t) \neq \emptyset \\ \mathcal{E}_{s:\ell}(c_0, \max(\mathcal{T}_{s:\ell}(c,t))) & \text{if } c \mapsto \mathsf{SELECT}(c_0, c_1) \text{ and } \mathcal{T}_{s:\ell}(c,t) \neq \emptyset \\ \{\mathcal{E}_{s:\ell}(c_0, t_0)^{t_0} | t_0 \in \mathcal{T}_{s:\ell}(c_0, t) \cap (\max(\mathcal{T}_{s:\ell}(c,t)) - \ell_0, t]\} & \text{if } c \mapsto \mathsf{WINDOW}(c_0, \ell_0, c_1) \text{ and } \mathcal{T}_{s:\ell}(c,t) \neq \emptyset \\ \mathcal{E}_{s:\ell}(c_0, \mathrm{prev}(\mathcal{T}_{s:\ell}(c_1,t))) & \text{if } c \mapsto \mathsf{PRE}(c_0, c_1, \ell_0) \text{ and } |\mathcal{T}_{s:\ell}(c,t)| \geq 2 \text{ and } \mathcal{T}_{s:\ell}(c_0, \mathrm{prev}(\mathcal{T}_{s:\ell}(c_1,t))) \neq \emptyset \\ \ell_0 & \text{if } c \mapsto \mathsf{PRE}(c_0, c_1, \ell_0) \text{ and } |\mathcal{T}_{s:\ell}(c,t)| \geq 2 \text{ and } \mathcal{T}_{s:\ell}(c_0, \mathrm{prev}(\mathcal{T}_{s:\ell}(c_1,t))) = \emptyset \\ \ell_0 & \text{if } c \mapsto \mathsf{PRE}(c_0, c_1, \ell_0) \text{ and } |\mathcal{T}_{s:\ell}(c,t)| = 1 \\ \bot & \text{if } \mathcal{T}_{s:\ell}(c,t) = \emptyset \end{cases}$$

$$\mathcal{T}_s(c,t) = \bigcup_{\ell} \mathcal{T}_{s:\ell}(c,t) \qquad \mathcal{E}_s(c,t) = \mathcal{E}_{s:s(\max(\mathcal{T}_s(c,t)))}(c,t) \text{ if } \mathcal{T}_s(c,t) \neq \emptyset$$

Figure 4: Formal semantics; ℓ is a literal, t is a timestamp, s is a stream, c is a cell, and ϕ is a function.

streams and computations are partitioned according to the value of the partitioning stream, e.g., ACME, which identifies the partition. This means that the spreadsheet is really a collection of sheets, one for each value for the partitioning stream. Each sheet represents a partition and contains calculations for a specific value of the partitioning stream. The formulas are computed for each partition independently. The output streams are obtained by merging the computed values for all partitions in order of timestamps. For instance, the VWAP formula of our example spreadsheet outputs for each input trade the volume-weighted average for the traded stock, i.e, the specific stock in this particular trade.

3.2 Definitions

Let a *tick* T be a possibly empty, possibly infinite sequence of natural numbers $\{t_1, t_2, ...\}$ denoting timestamps, e.g., microseconds since midnight. A non-empty finite tick T always admits a *maximal element* $\max(T)$. Given a finite tick T with at least two elements, we define the *second-to-max element* $\mathrm{prev}(T)$.

Let a *value* v be either a literal ℓ or a *window* w—a finite, possibly empty set of literals with pairwise distinct timestamps: $\{\ell_1^{t_1}, ..., \ell_{|w|}^{t_{|w|}}\}$.

Let a *stream* s be a map from a tick to values. We write $\mathrm{dom}(s)$ for the tick of s and $s(t)$ for the value of s at time t. We say that s *ticks* at time t if $t \in \mathrm{dom}(s)$. For convenience, if $t \notin \mathrm{dom}(s)$ but $t \geq \min(\mathrm{dom}(s))$, we write $s(t)$ for the most recent value of s at time t, i.e., $s(\max(\mathrm{dom}(s) \cap [0,t]))$. A stream or cell value is *absent* before its first tick, denoted by $\bot$.

Let a *spreadsheet* S be a finite collection of *cells*. Each cell has a unique name c and a *formula* f. We write $c \mapsto f$ if cell c maps to formula f.

The syntax of formulas is specified in Figure 4 where ϕ denotes a family of functions (such as division /, greater-than >, or Excel's IF or SUM functions). For simplicity, we do not formalize nested function applications and model constant formulas implicitly by means of constant streams. There are no types. Functions can consume and produce error values. They also accept absent values but do not produce the absent value.

3.3 Semantics

The semantics of a spreadsheet S is defined as a function of its *partitioning stream* s. Figure 4 first specifies the tick of cell c up to time t—$\mathcal{T}_{s:\ell}(c,t)$—and the value of c at time t—$\mathcal{E}_{s:\ell}(c,t)$—for the *partition* ℓ. For example, in Figure 2, the partitioning stream s is sym and the observed partition ℓ is ACME. Hence, $\mathcal{T}_{\mathsf{sym:ACME}}(\mathsf{B3}, t)$ defines the tick of Cell B3 and $\mathcal{E}_{\mathsf{sym:ACME}}(\mathsf{B3}, t)$ defines the value of Cell B3 at time t.

Even though the screenshot of the client in Figure 2 displays only one partition $\ell = \mathsf{ACME}$, the server computes all partitions. The tick and values of all partitions of c—$\mathcal{T}_s(c,t)$ and $\mathcal{E}_s(c,t)$—are obtained by combining the ticks and values for all partitions. As defined at the bottom of Figure 4, the tick of c is simply the union of the ticks of c for all partitions. The value of c at time t is the most recently computed value for c across all partitions.

We now explain the specification of $\mathcal{T}_{s:\ell}(c,t)$ and $\mathcal{E}_{s:\ell}(c,t)$.

Function application. A function application $\phi(c_1, ..., c_n)$ ticks when any argument does—the tick of the application is defined as the union of the ticks of the arguments. In particular, it starts producing values as soon as one of the argument cells c_i does. The function application at time t is computed from the current values of the cells at time t. By design, the value of c_i at time t is the most recently computed value of c_i or the absent value if c_i has not been computed yet. E.g., $\mathsf{SUM}(c_1, ..., c_n)$ is the sum of the current values of the cells $c_1, ..., c_n$ where an absent value is interpreted as zero.

Selection. The SELECT construct filters a data stream according to a condition stream. $\mathsf{SELECT}(c_0, c_1)$ ticks when c_1

does and evaluates to TRUE, returning the most recent value of c_0. For instance, if c_1 contains formula $c_0 > 0$ then SELECT(c_0, c_1) streams the positive values in c_0.

Windowing. The WINDOW construct collects the most recent values of a stream: WINDOW(c_0, ℓ_0, c_1) captures the values of c_0 with their respective timestamps in the last ℓ_0 time units up to the latest tick of c_1. If t is the latest tick of c_1, the window contains the values of c_0 from the time interval $(t - \ell_0, t]$. Because the window duration ℓ_0 is a statically-known constant, windows can be maintained incrementally by appending new values and evicting old values.

Since WINDOW(c_0, ℓ_0, c_1) is only computed when c_1 ticks, it can contain stale values. But the only way to observe such stale values is to use the window in combination with faster-paced streams than c_1. And no stream in a spreadsheet partition can be faster-paced than the combined inputs to that partition. For the VWAP example from Figure 2, if $c_1 \mapsto \phi'(\mathsf{Trades}.f, \mathsf{Quotes}.g)$, where ϕ' is an arbitrary spreadsheet function and f and g are arbitrary fields, then it is impossible to observe stale window contents in the spreadsheet partition. Updating the window contents more frequently would yield no benefit, as the updates could not be observed. Indeed, it would even be harmful, as it would violate partition isolation discussed later in this section.

Setting c_1 by hand may be burdensome for the programmer. Since it is not that important in our benchmarks, our implementation provides a binary WINDOW(c_0, ℓ_0) that expands to WINDOW(c_0, ℓ_0, c_0). An simple alternative implementation would be for the compiler to generate a c_1 that is the combination of all input streams.

Pre. The PRE construct makes it possible to record values—PRE(c_0, c_1, ℓ_0) ticks when c_1 does returning the value of c_0 at the previous tick of c_1. If c_1 ticked only once so far or c_0 was absent at the previous tick of c_1, it returns ℓ_0 instead.

PRE is useful for constructing state machines via feedback loops, for instance, to derive the value of a cell from the previous value of the same cell. In fact, all cyclic dependencies in a spreadsheet must use PRE, because other cycles lack well-founded semantics. Our previous paper [42] demonstrates that spreadsheets free of cyclic immediate dependencies have well founded semantics. Formally, a spreadsheet S is *well-formed* if the directed graph $\mathcal{G}$ of immediate dependencies in S is acyclic, where the vertices of $\mathcal{G}$ are the cells in S and there exists an edge (c, c') in $\mathcal{G}$ if $c' \in \mathrm{deps}(c)$ (see Figure 4). PRE has no immediate dependency on its first argument. The ticks and values up to time t of the cells of a well-formed spreadsheet S are defined via a well-founded recursion from the ticks and values of the input streams of S up to time t.

This work adopts the same approach and simply adds that a window depends on its arguments: $\mathrm{deps}(c) = \{c_0, c_1\}$ if $c \mapsto$ WINDOW(c_0, ℓ_0, c_1). Because of this immediate dependency WINDOW cannot replace PRE to build a spreadsheet with cyclic dependencies. While both constructs encapsulate memory, they serve different purposes: PRE with its built-in delay is meant for feedback loops whereas WINDOW is intended for on-the-fly data aggregation.

Partitioning. Partitioning is enforced by the specification of $\mathcal{T}_{s:\ell}(c, t)$ when c maps to an input-stream formula s_0. The condition $s(t_0) = \ell$ masks the ticks of s_0 that occur while the most recent value of the partitioning stream s is not ℓ. By induction, computations in partition ℓ only depend on input values received when $s(t_0) = \ell$.

In a sense, this semantics maintains one hidden instance of the spreadsheet for each partition. At any point in time, only the instance corresponding to the current value of the partitioning stream receives new input values and is updated accordingly. The other instances lie dormant and unchanged. The spreadsheet collects all the updates from all the hidden instances by merging them in order. One input value can only trigger computation in one partition. By induction, values computed for distinct partitions have distinct timestamps, hence the ordered merge is unambiguously defined. An equivalent interpretation of this specification is that all cells in the spreadsheet persist not just a single value, but rather a map from partitions to values.

Our approach to partitioning guarantees *partition isolation* [44], i.e., the combination of two properties:

- Temporal isolation: computations for distinct partitions cannot happen at the same time.
- Spatial isolation: computations for distinct partitions depend on disjoint sets of input values.

Formally, given a spreadsheet S with partitioning stream s and distinct partitions ℓ_0 and ℓ_1, we establish:

THEOREM 1. *The intersection of $\mathcal{T}_{s:\ell_0}(c, t)$ and $\mathcal{T}_{s:\ell_1}(c, t)$ is empty for any cells c in S and time t.*

PROOF SKETCH. Corollary of $t_0 \in \mathcal{T}_{s:\ell}(c, t) \Rightarrow s(t_0) = \ell$. Proof by induction on the structure of $\mathcal{T}_{s:\ell}$. □

THEOREM 2. *Adding, removing, or replacing the value of an input stream s_0 at time t_0 such that $s(t_0) = \ell_0$ does not affect the semantics $\mathcal{T}_{s:\ell_1}(c, t)$ and $\mathcal{E}_{s:\ell_1}(c, t)$ of partition $\ell_1 \neq \ell_0$ for all cells c in S and time t.*

PROOF SKETCH. For input-stream formulas, if $c_0 \mapsto s_0$ and $c_1 \mapsto s_1$ and $s(t_0) = \ell_0$ and $\mathrm{dom}(s_1)$ is the disjoint union of $\mathrm{dom}(s_0)$ and $\{t_0\}$ and $s_1(t) = s_0(t)$ for all $t \in \mathrm{dom}(s_0)$ then $\mathcal{T}_{s:\ell_1}(c_0, t) = \{t \in \mathrm{dom}(s_0) \cap [0, t] | s(t) = \ell_1\} = \{t \in \mathrm{dom}(s_1) \cap [0, t] | s(t) = \ell_1\} = \mathcal{T}_{s:\ell_1}(c_1, t)$ for all t since $s(t_0) = \ell_0$. Moreover $\mathcal{E}_{s:\ell_1}(c_0, t) = s_0(\max(\mathcal{T}_{s:\ell_1}(c_0, t))) = s_0(\max(\mathcal{T}_{s:\ell_1}(c_1, t))) = s_1(\max(\mathcal{T}_{s:\ell_1}(c_1, t))) = \mathcal{E}_{s:\ell_1}(c_1, t)$ for all t since $\max(\mathcal{T}_{s:\ell_1}(c_1, t)) \in \mathrm{dom}(s_0)$. For other formulas the result follows by structural induction over the mutually recursive definitions of $\mathcal{T}_{s:\ell}$ and $\mathcal{E}_{s:\ell}$. This recursion is well-founded for well-formed spreadsheet (see [42] for a proof of well-formedness). □

While the amount of state in a spreadsheet depends on the number of partitions, the amount of computation does not. Moreover, the state is partitioned and the partitions can be maintained independently from one another.

Following our prior work [42], we also establish that the semantics of a spreadsheet can be computed incrementally over time. Informally, at each tick t of an input stream, the set of cells to recompute and the resulting values depend only on the input values at time t and the current state of the spreadsheet—the values of the cells and the values stored by each occurrence of PRE.

The space and time complexity of the incremental computation can be large because of partitions and windows. Large numbers of keys and large windows can result in a lot of data. Computing aggregates can therefore become a bottleneck. Section 4 discusses how to parallelize computations across partitions and incrementalize computations over windows to mitigate the cost of these new capabilities.

```
1 stream<Bargain> Bargains = SpreadSheet(Trades; Quotes) {
2  param
3   spreadsheet    : "vwap.xls";
4   inputs         : {
5    A3 =Trades.sym, B3 =Trades.price, C3 =Trades.vol,
6    A12=Quotes.sym, B12=Quotes.price, C12=Quotes.seller};
7   partitionByLHS : Trades.sym;
8   partitionByRHS : Quotes.sym;
9   timeByLHS      : Trades.ts;
10  timeByRHS      : Quotes.ts;
11 output
12  Bargains       :
13   sym =RString("E12"), seller =RString("F12"),
14   vwap=Float64("G12"), bargain=Boolean("H12");
15 }
```

Figure 5: Configuring SPL's spreadsheet operator for the spreadsheet in Figure 2.

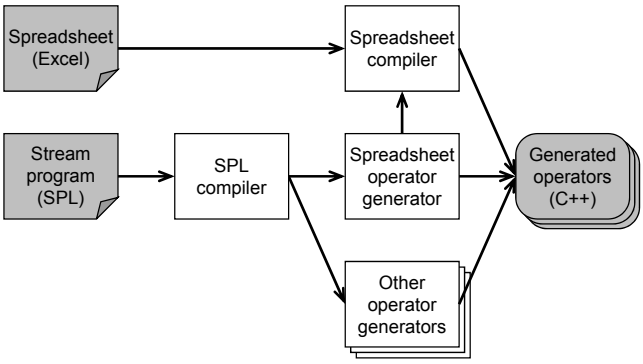

Figure 6: Compiling spreadsheets used from SPL.

4. SPREADSHEET COMPILATION

This section presents a compiler that implements the formal semantics from Section 3. This is the first published compiler for streaming spreadsheets, and the first implementation of partitioning and time-based windows in streaming spreadsheets.

4.1 Overview

In the scenario that we envision, the user writes a spreadsheet and describes where it fits in a larger stream program, i.e., how it connects to other upstream and downstream operators. To support end-users, a simple harness may be pre-defined or auto-generated to input data into and output data from the spreadsheet (e.g., following Chang and Myers's user interface [13]). This paper uses stream programs written in SPL [22], but the approach could be adapted to other streaming languages. A stream program describes a directed graph of streams and operators. Each stream is a conceptually infinite ordered sequence of tuples, where a tuple is a record with named attributes. Each operator is a source, a sink, or a stream transformer. The program configures the operators and arranges them in a graph.

Figure 5 shows an example for how to configure a spreadsheet as an operator for a larger stream program in SPL. Line 1 connects the operator to an output stream Bargains and two input streams Trades and Quotes. Line 3 names the file containing the actual spreadsheet from Figure 2. Lines 4–6 assign attributes of input tuples to spreadsheet cells. Lines 7–10 identify the attributes serving as timestamps for time-based windows and as partitioning keys. (Note that timestamps and keys need only be specified when the user wants to take advantage of time-based windows and partitioning, respectively.) Finally, Lines 12–14 assign spreadsheet cells to attributes of output tuples. The SPL development environment provides wizards for configuring operators without having to enter the SPL code by hand.

The spreadsheet compiler, described below, is independent of SPL. It reads the spreadsheet file and generates optimized C++ code for it. Figure 6 depicts how the spreadsheet compiler is used in the context of SPL. The SPL compiler is extensible with a library of operator generators; it parses the SPL program and performs some checks, but delegates the compilation of individual operators to the corresponding code generators. Specifically, when the SPL compiler encounters a use of the spreadsheet operator, it invokes the spreadsheet operator generator. The spreadsheet operator generator checks and resolves names and types of parameters and input/output assignments. If there are no errors, it invokes the spreadsheet compiler. In addition, it generates surrounding code for calling the cell setters, cell getters, and computing functions produced by the spreadsheet compiler. The resulting code is then linked with the C++ code produced by all the operator generators and with the SPL runtime library to yield binaries that can execute on either a single machine (typically a multi-core) or on a cluster of machines.

4.2 Design Considerations

Overall, the spreadsheet compilation is faithful to the formal semantics in Section 3. Conventional spreadsheet functions (represented by ϕ in the semantics) are pure and deterministic.[1] Most spreadsheet processors come with rich function libraries, but a small subset accounts for most practical uses. Our current research prototype implements relatively few functions, but can be easily extended with additional functions as needed. Computed cell references are only supported via Excel's INDEX, VLOOKUP, and MATCH functions, all other references must use explicit cell names.

The supported types are floating point numbers, strings, Booleans, and errors. As is typical in spreadsheet processors, functions are total and handle all types, producing or forwarding error values as needed. The calculus constructs PRE, WINDOW, and SELECT are exposed to the spreadsheet programmer as new functions. We check at compile time that results of invocations of WINDOW can only flow into aggregation functions that return a simple scalar, such as SUM, COUNT, or AVERAGE. This means that consistently with the formal calculus, windows cannot nest, and can be thought of as enforcing a simple type system on functions.

Our calculus and implementation rely on *universal partitioning*, where either no input streams are partitioned, or all input streams are partitioned using the same key type. Universal partitioning is sufficient but not necessary for partition isolation. This enables an implementation where partitioning is handled entirely by the operator in the stream graph and the spreadsheet compiler is oblivious to it. Note that different operators in the stream graph can be partitioned differently.

Timestamps needed by time-based windows are given by attributes of input tuples. All attributes of a single tuple

[1]The overwhelming majority of typical spreadsheet functions have these properties. Exceptions such as RAND are not currently supported.

are synchronous with each other. The tick of an output is defined as specified in the formal semantics in Section 3. In particular, upon a given input, not all outputs necessarily tick. The ticking outputs are determined by static dependencies (cell references) and dynamic dependencies (uses of SELECT). Our implementation submits an output tuple if at least one of the cells feeding its attributes ticks.

Our calculus and implementation rely on *universal time*, where time is strictly monotonically increasing across all input streams. Universal time is easy to establish in the common case where there is only a single input stream, but is inherently difficult when there are multiple input streams, such as Trades and Quotes for computing a bargain. This difficulty stems from clock skew in distributed systems: timestamps of tuples from different remote sources cannot be assumed to be based on the same clock.

This problem is well-recognized in the streaming literature, and there are different solutions. Srivastava and Widom describe a solution that waits for tuples that are slightly out of order, while dropping and logging tuples that are substantially out of order [37]. There are other cases where the problem is easier to solve; for instance, if input streams lack a sender-assigned timestamp attribute, the receiver can inject timestamps satisfying universal time. External time management is orthogonal to spreadsheet compilation. The experiments in this paper simply use test input streams that satisfy universal time by construction. To support different time management solutions, our SpreadSheet operator can be configured to either fire immediately on each tuple, or to only use tuples for setting input cells, but delay firing until punctuations [41].

4.3 Spreadsheet Compiler

The spreadsheet compiler is implemented as a Java application that consumes a spreadsheet in Microsoft Excel format and generates a C++ class that stores the state of the sheet and can perform its computation. The compiler also requires as arguments a specification of input and output cells. The input cells are passed as a list of lists, representing the mapping of input streams, each with its own list of attributes, to cells in the spreadsheet.

The compiler front-end leverages Apache's POI library [3] to process Excel spreadsheets in their original binary format. After parsing, spreadsheets are internally represented as sets of key-value pairs. The compiler applies a series of standard phases (expression flattening, constant propagation, and dead code elimination), introducing additional synthetic cells when necessary.

After normalization, the compiler computes, for each cell, a conservative over-approximation of the set of input streams for which it ticks. In the absence of SELECT, this set can be computed exactly; but after SELECT, an output cell may dynamically skip a tick of an input cell it depends upon statically. The computation follows the semantics described in Section 3: constants never tick, PRE cells tick when their second argument ticks, SELECT cells tick when their second argument ticks and evaluates to TRUE with the value specified in their first argument, and all other cells (including WINDOW cells) tick when any of their arguments ticks.

Using this information, the compiler generates, for each input stream, a function that propagates the computation through all ticking cells. This function operates in two steps: it first updates all cells that contain an invocation of PRE, copying parts of the previous state as appropriate, then computes the new values for all other cells. PRE cells can potentially reference each other in cycles, and updating their values may require additional temporary memory (at most the number of such cells). Other cells, by construction, do not have cyclic dependencies, and the compiler emits code that updates them in-place, following a topological ordering of their dependency graph. For a spreadsheet with p invocations of PRE and n other cells, the generated class will therefore need to store at most $2 \cdot p + n$ values (not counting time-based windows). The actual storage requirements are reduced by an optimization phase that identifies cells occurring in a single propagation function and promotes them to temporary, locally-allocated, variables.

The generated code is supported by a companion C++ library for manipulating spreadsheet values. Values are represented using a single universal type encoded as a tagged union. Spreadsheet functions (IF, SUM, etc.) are written in header-only, templated code, such that the output of the spreadsheet compiler can be properly optimized when passed to the C++ compiler. For instance, functions of variable arity such as SUM are implemented using loops, but the loop bounds are always determined statically and passed as template arguments. The language of supported spreadsheet functions is extended simply by writing C++ implementations for the desired functions.

The compiled class exposes public member functions serving three purposes: 1) setters, used to communicate new values to fill input cells, 2) compute, used to trigger the recomputation of the spreadsheet, and 3) getters, used to retrieve the values of output cells. The protocol for a client to process a tuple from a stream is to first invoke the setters corresponding to each attribute, then trigger the computation, and finally to inspect the values of the desired output cells. The getters accept as a parameter a pointer to a Boolean, allowing the compiled spreadsheet to communicate to the client whether the output value has ticked since the last inspection. In the case of a spreadsheet using time-based windows, the timestamp corresponding to the tuple arrival time is passed as an argument to compute.

4.4 Time-Based Windows

Recall that the results of windows can only flow into aggregation functions. In a streaming spreadsheet, *aggregation functions* are functions with a window argument that return a scalar. Besides traditional streaming aggregations familiar from databases [39], such as SUM(w) or AVERAGE(w), this also encompasses common spreadsheet functions, such as COUNTIF(w,v) or INDEX(w,i). These spreadsheet functions differ from traditional aggregations in that they may have additional non-window arguments. Furthermore, in our experience, it is common for multiple aggregates to refer to the same window: the formula COUNTIF(w,INDEX(w,COUNT(w)))=1 checks whether the last number inserted into window w is unique. This formula can be used to encode the IStream operator from stream-relational algebra [4].

The requirements for our implementation of windows are therefore that they should be *incremental* (their time complexity should be sub-linear in the window size $|w|$) and *irredundant* (the window should be shared among multiple aggregates whenever applicable, and updating the window of an aggregate should be separate from updating its non-window parameters). To accomplish this, we support win-

Function	Description	Data structure	Time, Space
SUM(w)	Total of the numbers in w.	Float	O(1), O(1)
AVERAGE(w)	Arithmetic mean of the numbers in w.	Two floats	O(1), O(1)
COUNT(w)	Number of elements in w with numbers.	Integer	O(1), O(1)
COUNTIF(w, v)	Number of elements in w that equal v.	Hash multi-set	O(1), O($\|w\|$)
INDEX(w, i)	Element of w at index i, where 1 is the oldest.	Resizable circular buffer	O(1), O($\|w\|$)
MATCH(v, w, m)	Index of element equal to v in w if m is 0 (exact match).	Tree multi-map, integer	O(log $\|w\|$), O($\|w\|$)
LARGE(w, k)	Number in w that is the kth largest, where 1 is the max.	Order statistics tree	O(log $\|w\|$), O($\|w\|$)

Table 1: Incremental sliding-window aggregations, where w is the window and $|w|$ is the window size.

dows and aggregations by a Window class and an Aggregate class with its subclasses in the companion C++ library. A Window object maintains a FIFO buffer of time/value pairs, as well as the elements evicted and inserted in the current tick. Each Aggregate object holds a constant pointer to its base window and maintains a data structure for fast incremental aggregation.

Table 1 lists supported aggregations with their signature, description, data structure, and algorithmic complexity. The time complexity is the worst-case of evict, insert, or compute calls (typically, these three have the same complexity). Our implementation of MATCH currently only supports mode $m = 0$, which implements exact matches. It uses a map from values to *stable* indices, which are the indices an element would have if there were no evictions. To obtain the actual index, subtract the total number of evictions that happened so far. Our implementation of LARGE uses an *order statistics tree*, which is a balanced search tree where internal nodes track statistics of the sizes of their subtrees.[2] Since elements in a search tree are ordered, a single root-to-leaf traversal can find the kth largest element using these statistics.

Subclasses of Aggregate offer separate methods update and apply. When a window changes, the update functions of all dependent aggregates are called. They query the window for the elements evicted and inserted in the current tick and update their data structures accordingly. The arguments to the apply function consist of the latest values of all non-window inputs to the aggregation. For example, the inputs to COUNTIF are a window w and a value v, and therefore, COUNTIF::apply has one argument, the value v. In other words, the apply functions are curried on the window argument. The apply function gets called each time the aggregate ticks, i.e., each time any of its inputs ticks.

4.5 Runtime Support

Figure 7 shows a SpreadSheet operator generated by the compilation depicted in Figure 6 in the context of a simple stream graph. The Import and Export operators can be based on TCP; or can use pub-sub when business users create ad-hoc spreadsheets [16]; or can contain local input generators and output validators for testing purposes. The TimeManager establishes universal time as discussed above.

[2]https://en.wikipedia.org/wiki/Order_statistic_tree

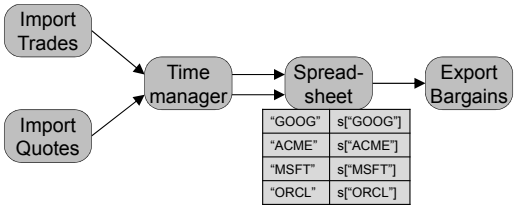

Figure 7: Stream graph with spreadsheet operator.

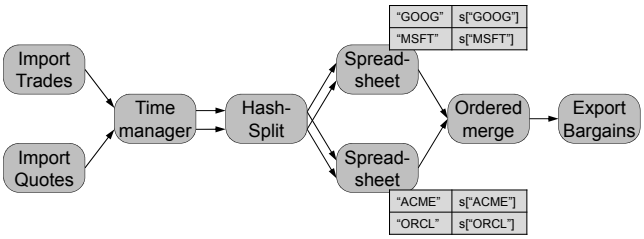

Figure 8: Stream graph with parallel spreadsheet operator.

The state of a partitioned spreadsheet operator holds a mapping from keys (e.g. "ACME") to spreadsheet states (e.g. s["ACME"]). A spreadsheet state holds the values of cells that did not tick along with any data required to implement PRE and WINDOW. When an input tuple arrives, the spreadsheet operator extracts the key, retrieves the spreadsheet state, and calls the appropriate cell setters. Upon a firing (triggered by an input tuple or punctuation), the operator calls compute on the spreadsheet state for the given key, and submits tuples on output streams corresponding to output cells that ticked, if any. In general, the stream graph can of course also contain more operators than shown in Figure 7, such as operators that parse XML or enrich streams with quasi-static information from a database.

Partitioning can be exploited for parallelization [24]. Figure 8 shows a version of Figure 7 that adds parallelism by using multiple replicas of the Spreadsheet operator. We refer to each replica along with its sub-streams as a parallel *channel*. For illustration purposes, Figure 8 shows only two channels, but in general, the number of channels is a trade-off between speedup and resource consumption. The HashSplit operator sends each tuple to a channel determined by its key. That guarantees that tuples with the same key always go to the same channel, and thus, the Spreadsheet operator in each channel holds the correct spreadsheet state. Since state is disjoint, no inter-channel communication is required.

Tuples within a single channel are delivered in order, but tuples in different channels may be out-of-order depending on processing and communication speed. Therefore, the stream graph contains an OrderedMerge operator that interleaves tuples from all channels in an order consistent with their timestamp attributes. Note that the sequence of timestamps can have gaps but no duplicates, since our formal semantics enable sampling but not stuttering. The OrderedMerge maintains one FIFO queue per channel. When it receives a tuple, it inserts it to the corresponding queue, and

	cells	exprs	windows	partition
vwap	9	14	2x5m	ticker
twitter	22	36	3x5m	lang
linearroad	20	18	3x30s & 2x5m	segment
average	33	27	2x6	–
kalman	14	21	2x2	target id
tax	21	37	–	–
pong	35	86	–	game id
forecast	43	60	2x6	location

Table 2: Benchmarks.

	SS *Ktps*	SPL *Ktps*	SS/SPL
vwap	960.61	399.39	2.41
twitter	45.91	30.06	1.52
linearroad	964.32	798.72	1.21
average	3,937.01	8,000.00	0.49
kalman	3,816.79	8,196.72	0.47
tax	1,383.13	4,975.12	0.28
pong	480.77	3,937.01	0.12
forecast	913.24	7,936.51	0.12

Table 3: Throughput results.

then drains all queues as much as possible. It can drain (dequeue and submit) a tuple if its timestamp is minimal among all tuples at the front of channel queues and there is no empty queue. The latter requirement guarantees that there are no tuples with smaller timestamps in-flight on any channel whose queue is currently empty. To avoid deadlock, the channel queue sizes are dynamic; an alternative solution would be to periodically flush all channels [28].

5. EXPERIMENTAL STUDY

We evaluate our approach with respect to the following questions:

RQ1. Are spreadsheets a suitable programming model for applications handling unbounded data sets?

RQ2. Do compiled spreadsheets approach the performance of hand-written code?

RQ3. How do time-based windows affect performance?

We consider a number of benchmarks, described below. We wrote SPL benchmarking harness code around the spreadsheets following the illustration in Figure 6. The following sections describe the benchmarks (5.1), report the performance results (5.2), and quantify the impact of incremental window updates on performance (5.3). We conclude by mentioning threats to the validity of our conclusions, including a discussion of parallelization (5.4).

5.1 Benchmarks (RQ1)

The vwap example was described in Section 2. It uses both time-based windows and stock-ticker based partitions. The twitter example computes the top-k languages with the most verbose tweets over 5 minute windows. The application is composed of two spreadsheets: the first calculates the average length of tweets for every language in a 5-minute time-frame, and the second sorts these to display the top-k languages with the most verbose tweets. The linearroad example is a vehicle toll system for expressways with variable toll rates. Our implementation assumes a partitioning by road segments as this is the most natural to implement in the spreadsheet. It follows the design sketched in [4]. The average example calculates a recency-weighted average. The kalman example implements a Kalman filter to estimate the state of a system based on a stream of noisy and inaccurate measurements [26]. The computation of the estimation depends on past values, and in our spreadsheet we use PRE to retain this state. The tax example calculates progressive income taxes using a table to encode tax brackets. The pong example calculates a 2D line intersect to play the game Pong, i.e., to position the paddle to catch the ball. It assumes a single incoming stream of positions and velocities for all the games currently being played and uses partitioning by game id to separately keep track of the state of each game. Finally, the forecast example performs linear regression using least-square fit to predict future temperatures.

The benchmarks are summarized in Table 2. For each example, we show the number of cells needed to encode the computation in the spreadsheet (aggregating the numbers for both spreadsheets for twitter) as well as the number of live expression nodes in the abstract syntax tree after dead-code elimination, e.g., a cell containing the equation (A1*A2)+(B1*B2) results in three expression nodes.

In Table 2, the *window* column reports the number and size of each window in the benchmark as NxW where N is the number of windows and W is the size of the windows either in time or number of historical values, e.g., 2x5m represents 2 windows, each of which is 5 minutes long, and 2x6 represents two windows, each containing 6 historical values. The *partition* column records the attribute used for partitioning the input stream.

Our benchmarks are all expressed concisely, and mirror applications from a variety of domains: finance, analytics, Internet-of-things, and engineering.

5.2 Spreadsheet Throughput (RQ2)

We compiled each spreadsheet and ran it as part of an SPL test harness on a 2-processor machine with 32 GB of RAM running Red Hat Enterprise Linux Server release 6.5. Each processor is a 2.93 GHz Intel Xeon X5570 with 4 cores and 8 hardware threads. The experiments were repeated 5 times and the arithmetic mean throughput is reported throughout. We created the input sets using real traces when available and synthetically generated data otherwise.

We report the throughput for each benchmark (SS) in Kilo tuples per second (*Ktps*). This is calculated by recording the total time spent in the code corresponding to the spreadsheet, including the cost to read and write data from the I/O streams. Every benchmark processed a total of 1M input tuples. The results are shown in Table 3. For each benchmark, we manually wrote an equivalent, pure SPL, implementation. We measured the throughput for these implementations as above and report it under the heading SPL.

The last column in the table computes the ratio between the SS and SPL throughputs such that a value less than one indicates the compiled spreadsheet is slower than SPL and conversely a value greater than one indicates the compiled spreadsheet is faster.

The last five compiled spreadsheets are 2x to 8.7x slower than SPL. Given that SPL is a state-of-the-art system designed for very high frequency and low latency applications such as real-time trading, these results are not bad. We at-

tribute most of the slow-down to dynamic type checking and type conversions to and from a tagged union. Three of the compiled spreadsheets (`vwap`, `twitter`, and `linearroad`) are respectively 1.52x, 2.41x, and 1.21x faster than the corresponding hand-written SPL code. We attribute this to extra inter-operator communication in the SPL case, where the SPL code uses multiple operators for a computation that is implemented in a single compiled streaming spreadsheet. The average slowdown compared to SPL is 2x (geometric mean over all the benchmarks).

5.3 Performance of Window Updates (RQ3)

The three test cases with unbounded windows are `vwap`, `twitter` and `linearroad`. For these we measured the impact of incremental window updates, by measuring throughput with incrementalization disabled. The impact depends on the occupancy of a window: for instance, the average occupancy of a window for `vwap` is 7,800 trades and the incremental updates resulted in a 6.8x speedup end-to-end. For `linearroad`, we measure a 83x speedup. The windows in `twitter` have lower average occupancy and the effect of incrementalization is negligible.

Our hand-written SPL codes for the benchmarks also implement incremental window updates. In contrast to our compiler however, incrementalization in SPL was done manually for each aggregate, adding more SPL operators to the stream graph.

5.4 Threats to Validity and Parallelization

A critical reader may argue that our benchmarks are not production-ready applications. Obtaining suitable applications is a challenge, in part because of the novel proposition of making programming with streams accessible to end-users. We have strived to collect proxy applications for a variety of domains which we anticipate will provide the first adopters of our work.

Our baseline consists of hand-written SPL programs. With additional effort, they could have been made to perform better. However, we believe they are representative for what a typical skilled SPL developer would come up with.

Because of the relatively small size of the benchmarks, our experiments do not demonstrate the performance benefits of parallelization; for all benchmarks in Section 5.1, the overhead of parallelization dominates compared to the time spent in computations (for both the SPL and compiled spreadsheet versions). To assess the potential of parallel scalability, we have devised a computation intensive benchmark. It uses 400 cells to compute RGB pixel values of a visualization of the Mandelbrot set. For this benchmark, more channels yield higher throughput (see Figure 9).

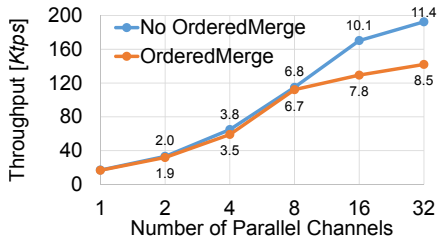

Figure 9: Parallel speedup for Mandelbrot.

6. RELATED WORK

We discuss spreadsheets as a programming platform, programming models for streaming, models of time, and finally windows and partitions in the context of streaming systems.

Only few prior efforts use spreadsheets for stream computing. The ActiveSheets paper was similar to this paper in that it formalized a calculus and implemented a server that enabled both data export and computation export [42]. *Data export* makes values computed by a spreadsheet available as a stream, and *computation export* extracts formulas of a spreadsheet client to run on a server. Chang and Myers also present streaming spreadsheets with a focus on the user interface [13]. Whereas both of these earlier papers implemented the server as an interpreter, this paper uses a compiler. Furthermore, this paper introduces time-based windows and partitioning, neither of which were present in the earlier work. Since the earlier work lacked these features, it could not capture applications like VWAP in their full generality. We also published a 2-page workshop paper with a vision for time-based windows and partitioning, but that paper lacked formal semantics, an implementation, or an evaluation [23].

McGarry introduced a delay-line object for streaming spreadsheets, which can be viewed as a count-based sliding window [31]. StreamBase streamed data both in and out of a spreadsheet, but did not export computation [38]. In contrast, Cloudscale exported stream computation from a spreadsheet, but the computation was not specified by spreadsheet formulas [14]. Finally, Woo et al. used spreadsheets to visualize data from sensor networks, and offered count-based sliding windows [43]. None of these offered time-based windows or partitioning.

There is a long history of scholarly articles on spreadsheets in the areas of user interfaces, programming languages, and software engineering. They share our motivation, recognizing that many end-users are familiar with spreadsheets, and thus, spreadsheets can enable domain expert to develop simple applications. Sajaniemi and Pekkanen did an empirical study on 135 Lotus 1-2-3 and Symphony spreadsheets [34]. Among other things, they found that the most common functions were IF, SUM, and ROUND, which means that even 28 years ago, selection and aggregation already played a big role in spreadsheets. Several papers pursued the idea of putting advanced code in spreadsheet cells [15, 27, 30, 33]. In function spreadsheets, a worksheet defines a function that can be called from elsewhere [6, 36]. FlashFill [19] and NLyze [20] synthesize spreadsheet code from examples and natural language. CheckCell offers data debugging for spreadsheets [7]. Gneiss uses spreadsheets to create web applications [12]. Unlike our work, none of these papers feature streaming spreadsheets, partitioning, or time-based windows.

Languages like Lustre [11], StreamIt [18], CQL [4], and SPL [22], or libraries such as Spark Streaming [45], enable seasoned programmers to express sophisticated streaming applications. At the other end of the spectrum, environments like Mario [10], *if this then that* [25], or Controlled English [5, 29] require little to no programming experience and enable end-users to specify simple streaming applications in intuitive ways. Streaming spreadsheets offer a sweet spot between these extremes, being both flexible and familiar to the end-users.

The formalization of time in this paper follows from synchronous programming languages [8]. Outputs are produced

instantly so that inputs and outputs are formally synchronous. While traditional synchronous models, including the original ActiveSheets paper, consider time to be an abstract quantity, we measure time. In other words, we give a meaning not only to $t_0 < t_1$ but also to $t_1 - t_0$.

Partitioning is a central feature in many streaming systems and languages. Systems that use a dialect of SQL for streaming, such as CQL [4] or StreamInsight [2], usually allow the user to specify any tuple attribute for partitioned windows and aggregations. Likewise, SPL allows using regular tuple attributes for partitioning; while this feature is motivated by functionality, it can also be exploited for parallelism [35]. All stream data items for Spark Streaming are key-value pairs, and this knowledge is baked into the design at a fundamental level [45]. In MillWheel, users specify a key extraction function for partitioning [1]. And finally, Storm relies on partitioning for parallelism, using the term *field grouping* to refer to a partitioned shuffle [40]. Our work is similar to these prior papers in that it brings partitioning to the forefront and enables parallelization based on partitioning. But none of these prior papers applies partitioning to streaming spreadsheets.

Time-based windows are widely used in stream processing systems, even when those systems do not support them as a built-in feature. CQL has a built-in notion of windows: it uses windows to transform streams to relations before applying relational operators from the database world [4]. In StreamInsight, each stream data item carries two timestamps, one for insertion and the second for retraction, thus implementing a window [2]. SPL offers a windowing library to make windows available to user-defined operators [17]. MillWheel [1] and Storm [40] do not explicitly provide windows, but interestingly, in both cases the papers include examples that implement windows by hand on top of a lower-level API. By building sliding windows into our design, we make it easier to take advantage of incremental aggregation. None of these prior papers applies sliding time-based windows to streaming spreadsheets. There are many different variants of sliding windows [9, 17]; future work could extend ActiveSheets to support more options.

7. CONCLUSIONS

We presented enhancements to streaming spreadsheets with time-based windows and partitioning, features that manipulate unbounded data sets and overcome the finite nature of the interface. Our enhanced spreadsheet is expressive, yet easy to use for non-programmers. We provide a compiler to C++ to facilitate integration with existing stream processing systems, such as Node-RED and IBM Streams. We presented a variety of benchmarks to illustrate the expressivity of our programming model and the efficiency of the compiler implementation.

Acknowledgements. We thank David Grove, Nagui Halim, Louis Mandel, Frank Tip, and the anonymous reviewers for their constructive comments.

8. REFERENCES

[1] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: Fault-tolerant stream processing at internet scale. In *Very Large Data Bases (VLDB) Industrial Track*, pages 734–746, 2013.

[2] M. H. Ali, C. Gerea, B. Raman, B. Sezgin, T. Tarnavski, T. Verona, P. Wang, P. Zabback, A. Kirilov, A. Ananthanarayan, M. Lu, A. Raizman, R. Krishnan, R. Schindlauer, T. Grabs, S. Bjeletich, B. Chandramouli, J. Goldstein, S. Bhat, Y. Li, V. Di Nicola, X. Wang, D. Maier, I. Santos, O. Nano, and S. Grell. Microsoft CEP server and online behavioral targeting. In *Demo at Very Large Data Bases (VLDB-Demo)*, pages 1558–1561, 2009.

[3] The Apache POI project. http://poi.apache.org. Retrieved Aug., 2015.

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

[5] M. Arnold, D. Grove, B. Herta, M. Hind, M. Hirzel, A. Iyengar, L. Mandel, V. Saraswat, A. Shinnar, J. Siméon, M. Takeuchi, O. Tardieu, and W. Zhang. META: Middleware for events, transactions, and analytics. *IBM Journal of Research & Development*, 60(2/3):15:1–15:10, 2016.

[6] D. Balson and J. Tyszkiewicz. User defined spreadsheet functions in excel. In *European Spreadsheet Risks Interest Group Annual Conference (EuSpRIG)*, 2012.

[7] D. W. Barowy, D. Gochev, and E. D. Berger. Checkcell: Data debugging for spreadsheets. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 507–523, 2014.

[8] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[9] I. Botan, R. Derakhshan, N. Dindar, L. Haas, R. J. Miller, and N. Tatbul. SECRET: A model for analysis of the execution semantics of stream processing systems. In *Very Large Data Bases (VLDB)*, pages 232–243, 2010.

[10] E. Bouillet, M. Feblowitz, Z. Liu, A. Ranganathan, and A. Riabov. A tag-based approach for the design and composition of information processing applications. In *Onward! Track of Object-Oriented Programming, Systems, Languages, and Applications (Onward!)*, pages 585–602, 2008.

[11] P. Caspi, D. Pilaud, N. Halbwachs, and P. Raymond. Lustre: a declarative language for real-time programming. In *Symposium on Principles of Programming Languages (POPL)*, 1987.

[12] K. S.-P. Chang and B. A. Myers. Creating interactive web data applications with spreadsheets. In *Symposium on User Interface Software and Technology (UIST)*, pages 87–96, 2014.

[13] K. S.-P. Chang and B. A. Myers. A spreadsheet model for handling streaming data. In *Conference on Human Factors in Computing Systems (CHI)*, pages 3399–3402, 2015.

[14] Big data analytics. http://www.hashdoc.com/document/8626/big-data-analytics. Retrieved Aug., 2015.

[15] J. Cunha, J. a. P. Fernandes, J. Mendes, and J. a.

Saraiva. MDSheet: A framework for model-driven spreadsheet engineering. In *International Conference on Software Engineering (ICSE)*, pages 1395–1398, 2012.

[16] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, June 2003.

[17] B. Gedik. Generic windowing support for extensible stream processing systems. *Software Practice and Experience (SP&E)*, 2013.

[18] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 151–162, 2006.

[19] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Principles of Programming Languages (POPL)*, 2011.

[20] S. Gulwani and M. Marron. NLyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. In *International Conference on Management of Data (SIGMOD)*, pages 803–814, 2014.

[21] F. Hermans and E. Murphy-Hill. Enron's spreadsheets and related emails: A dataset and analysis. In *International Conference on Software Engineering (ICSE)*, volume 2, pages 7–16, 2015.

[22] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. Mendell, H. Nasgaard, S. Schneider, R. Soulé, and K.-L. Wu. IBM Streams Processing Language: Analyzing big data in motion. *IBM Journal of Research & Development*, 57(3/4):7:1–7:11, 2013.

[23] M. Hirzel, R. Rabbah, P. Suter, O. Tardieu, and M. Vaziri. Spreadsheets for stream partitions and windows. In *Workshop on Software Engineering Methods in Spreadsheets (SEMS@ICSE)*, pages 39–40, 2015.

[24] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)*, 46(4), Apr. 2014.

[25] About IFTTT. https://ifttt.com/wtf. Retrieved Aug., 2015.

[26] R. E. Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, 82(Series D):35–45, 1960.

[27] E. Kandogan, E. Haber, R. Barrett, A. Cypher, P. Maglio, and H. Zhao. A1: End-user programming for web-based system administration. In *Symposium on User Interface Software and Technology (UIST)*, pages 211–220, 2005.

[28] P. Li, K. Agrawal, J. Buhler, and R. D. Chamberlain. Deadlock avoidance for streaming computations with filtering. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 243–252, 2010.

[29] M. H. Linehan, S. Dehors, E. Rabinovich, and F. Fournier. Controlled English language for production and event processing rules. In *Conference on Distributed Event-Based Systems (DEBS)*, 2011.

[30] B. Lisper and J. Malström. Haxcel: A spreadsheet interface to Haskell. In *Workshop on the Implementation of Functional Languages (IFL)*, pages 206–222, 2002.

[31] J. McGarry. Processing continuous data streams in electronic spreadsheets. Patent No. US 6,490,600 B1, 2002.

[32] Node-RED: A visual tool for wiring the internet of things. http://nodered.org. Retrieved Aug., 2015.

[33] K. W. Piersol. Object-oriented spreadsheets: The analytic spreadsheet package. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 385–390, 1986.

[34] J. Sajaniemi and J. Pekkanen. An empirical analysis of spreadsheet calculation. *Software – Practice and Experience (SP&E)*, 18(6):583–596, June 1988.

[35] S. Schneider, M. Hirzel, B. Gedik, and K.-L. Wu. Auto-parallelizing stateful distributed streaming applications. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 53–64, 2012.

[36] P. Sestoft. Implementing function spreadsheets. In *Workshop on End-User Software Engineering (WEUSE)*, pages 91–94, 2008.

[37] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *Principles of Database Systems (PODS)*, pages 263–274, 2004.

[38] StreamBase Microsoft Excel adapter. http://docs.streambase.com/sb66/index.jsp?topic=/com.streambase.sb.ide.help/data/html/samplesinfo/Excel_sample.html. Retrieved Aug., 2015.

[39] K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu. General incremental sliding-window aggregation. In *Conference on Very Large Data Bases (VLDB)*, pages 702–713, 2015.

[40] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm @Twitter. In *International Conference on Management of Data (SIGMOD)*, 2014.

[41] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *Transations on Knowledge and Data Engineering (TKDE)*, 15(3):555–568, 2003.

[42] M. Vaziri, O. Tardieu, R. Rabbah, P. Suter, and M. Hirzel. Stream processing with a spreadsheet. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 360–384, 2014.

[43] A. Woo, S. Seth, T. Olson, J. Liu, and F. Zhao. A spreadsheet approach to programming and managing sensor networks. In *Conference on Information Processing in Sensor Networks (IPSN)*, pages 424–431, 2006.

[44] Z. Xu, M. Hirzel, G. Rothermel, and K.-L. Wu. Testing properties of dataflow program operators. In *Conference on Automated Software Engineering (ASE)*, pages 103–113, 2013.

[45] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Symposium on Operating Systems Principles (SOSP)*, pages 423–438, 2013.