International Conference on Computational Science, ICCS 2012

Sliding Window Calculations on Streaming Data using the Kepler Scientific Workflow System

Sven Köhlera,∗, Supriya Gulatia, Gongjing Caoo, Quinn Hartb, Bertram Ludäschera

aDepartment of Computer Science, University of California, Davis
bDepartment of Land, Air and Water Resources, University of California, Davis

Abstract

In many areas of science unbounded (potentially infinite) data streams need to be processed in a continuous manner, e.g., to compute running or sliding window aggregates. One important example is the computation of Growing Degree Days (GDD) from a stream of temperature data, which provides a heuristic tool to predict plant development and the maturity of crops. The process of data acquisition, processing, storage, and presentation forms a scientific workflow and scientific workflow systems have been developed to automate their execution. The whole workflow is decomposed into its individual steps, represented by actors, which in turn are connected by channels that describe the flow of data. This workflow representation allows to reuse existing components for different workflows, and, in principle, easy modification of existing workflows. In current streaming workflow designs in Kepler, data belonging to a particular time window is typically identified by counting data tokens on channels between actors. For example, this token-counting approach does not work for windows of variable length nor for overlapping windows. In this paper, we address these limitations and present a new actor design with two incoming streams: a time-stamp ordered data stream, and a stream of aggregation windows, ordered by their start time. We present a new Chunker actor that “stream-joins” the data from one stream with the windows presented on the second stream, where windows represent aggregation intervals of variable length and possibly overlapping time. Windows containing the corresponding data are output as soon as they are completed, i.e., once timestamps in the data stream pass the end time of a window. We illustrate the approach with an improved GDD workflow based on our new Chunker actor.

Keywords: Scientific workflow, data streaming, continuous queries

1. Introduction

Scientists are faced with a range of computational problems in dealing with data collection, integration, analysis and visualization. Scientists often run parts of their experiments using different software and hardware tools and thus need to integrate these tools. Scientific workflows are representations of scientific processes that unambiguously describe data management, analysis, simulation, and visualization tasks. Scientific workflow management systems are a class of applications that pipeline scientific tasks and automate the execution of scientific workflows. They typically

∗Corresponding author
Email address: svkoehler@ucdavis.edu (Sven Köhler)
contain a toolkit of common components, which are used as building blocks to develop more complex workflows. Kepler is an open source scientific workflow system that builds upon the dataflow-oriented Ptolemy II system [1]. Kepler provides an intuitive GUI for the design and execution of workflows.

**Actor-Oriented Modeling.** The actor-oriented modeling paradigm of Kepler is an important feature derived from Ptolemy II. In Ptolemy, a system is viewed as a composition of components (or actors) that are interconnected with channels representing the flow of data. Every actor has input and output ports connecting them to channels. In Kepler, different models of computation are supported by explicitly assigning director to a workflow. The director defines how different components of a workflow communicate and how the workflow is executed.

**A Workflow Example.** In environmental science, data often takes the form of historical or real-time continuous data streams rather than static data sets. Processing and analysis of such data often requires grouping of values within a given time window and subsequently applying aggregation functions. In fact, those operations are a common component in any stream processing.

The REAP project [2] developed an example workflow in Kepler [3] to integrate sensor networks and to calculate Growing Degree Day (GDD) for a particular sensor. GDD is a measure of heat accumulation used to predict plant maturity. The crop development rate from emergence to maturity reliably depends upon the accrued daily temperatures. GDD is important in other agricultural studies including pest and weed management, and irrigation scheduling [4, 5]. The GDD measure is calculated as follows:

\[
GDD(T_{\text{min}}, T_{\text{max}}) = \begin{cases} 
0 & \text{if } T_{\text{max}} < T_{\text{base}} \\
\frac{T_{\text{max}} + T_{\text{min}}}{2} - T_{\text{base}} & \text{if } T_{\text{base}} \leq T_{\text{max}} \leq T_{\text{top}} \\
\frac{T_{\text{max}} + T_{\text{top}}}{2} - T_{\text{base}} & \text{if } T_{\text{top}} < T_{\text{max}} 
\end{cases}
\]

\(T_{\text{min}}\) and \(T_{\text{max}}\) are the day’s minimum and maximum temperatures. \(T_{\text{base}}\), the base temperature, varies by crop type or application and sets a minimum threshold on the effect of the day’s contribution to GDD. Similarly, \(T_{\text{top}}\) acts as an upper bound on the contribution to GDD for a single day. It is usually capped at 30 °C because most plants do not improve their growth beyond that temperature. Figure 1 shows an example plot of the GDD and average temperatures for a time window from the CIMIS02 [6] sensor station.

The conventional Kepler workflow design as developed in the REAP project implements the GDD computation, but also has a number of limitations. For example, grouping of data and therefore aggregation functions can only be applied to time windows of fixed length. Furthermore, overlapping windows and gaps between windows are not realizable. S. Gulati [7] developed the concept of a new actor that supports freely definable time-based sliding window computations and efficient aggregation of streaming data. In this paper, we present the main concepts of this new “Chunker” actor that was developed for Kepler. Furthermore, we describe limitations of the conventional design on the GDD workflow example and describe how the design can be improved to fully support streaming.

In particular, we make the following contributions:

- We present the design of our new Chunker actor that can extract data from a stream within arbitrarily definable time windows.
- We demonstrate a new version of the GDD workflow that is based on our new actor and supports streaming data.

The rest of the paper is organized as follows. Section 2 describes what we call the conventional design of the GDD workflow in more detail. Section 3 introduces our new actor. In Section 4 we demonstrate a new workflow design using our Chunker actor before we conclude in Section 5.

2. Conventional GDD Workflow Design

The conventional GDD workflow design implemented in Kepler as shown in Figure 2 calculates GDD from temperature measurements and plots it. The sensor data source, here a relational PostgreSQL database, outputs measurements as tuples \((t_i, d_i)\) where temperatures constitute a data field \(d_i\) of the record and the time \(t_i\) associated with the temperature reading constitutes a timestamp field for the record. As daily minimum temperatures and daily maximum temperatures are needed to calculate GDD, the incoming data stream is grouped into daily time windows of fixed
length using the Sequence_to_Array actor where each array contains a single day’s temperature measurements. This method can only be applied for regularly spaced inputs where the beginning of a day is known a priori. Kepler inherits aggregation actors from Ptolemy that operate on a single array input at a time. Using such array-based aggregation, the minimum, maximum and average temperatures can be calculated easily: First, the minimum, maximum and average values are computed for each array of temperature readings form a whole day. Then, the GDD is computed using an R actor. The R actor in Kepler works on a single input tuple: An array containing all readings is read from an input port. Then, it is converted to an R object, a user-defined script is executed, and the result is emitted on the output port. Finally, the GDD is plotted against timestamps with a plotting actor. This workflow will execute properly if it is guaranteed that the number of temperature readings is constant every day. However, in a realistic stream model the rate of measurements may vary, e.g., due to dropped readings or defective sensors. This requires special handling and a strategy to fill “gaps” in the data stream. The DataTurbine actor, for example, includes a “fillGaps()” for that purpose.

**Aggregation by Token Counting.** The Sequence_to_Array actor used in the GDD workflow of Figure 2 divides the incoming stream into arrays with a given, fixed number of records. The basic principle of this actor is shown in Figure 3. One incoming port receives a stream of data $D = d_1, d_2, \ldots$ indexed by timestamps $t_i$. The actor groups incoming data items into groups of a fixed size specified by parameter $s$. Grouping of data starts immediately from receiving the first token and thereby implicitly with the timestamp of the first data tokens $b$. There is no integrated mechanism to introduce gaps between windows while grouping. After receiving the specified number of data tokens, the completed window is output as an array.

Due to the fixed parameters for grouping, windows of different size, as in the case of Window 1 and Window 4 shown in Figure 4, are not possible. In the conventional “Token Counting” approach, the incoming stream is split into arrays. Thus, it is not possible to insert a data item into multiple time windows preventing overlapping windows as shown for Window 1 and Window 2 in Figure 4.

The conventional GDD workflow design also uses the synchronous dataflow director (SDF). This is a statically scheduling director that executes all actors serially according to a precalculated schedule. Thus, it does not allow a variable number of data tokens emitted by an actor on a channel in different scheduling rounds within the same
workflow execution. Therefore, the use of the SDF director is another reason preventing the use of variable windows, since the array to sequence actor is required to output an array containing a window in a constant interval after consuming a constant number of tokens from the data stream.

Finally, the R actor used to generate a plot requires all data to be available before the actor can create a graph. Thus, it is not possible to display results in a continuous stream but rather this requires the data stream for GDD measures to terminate.

3. Chunker Actor: Flexible Window-Based Grouping

In context of scientific workflows, the goal is to develop a single grouping actor that is simple enough to be used easily in a number of scenarios, but at the same time is general enough that it could be reused in a number of different workflows. We developed the concept of a Chunker actor shown in Figure 5 for general grouping of streaming data into given time windows. A Chunker implementation is available for the Kepler scientific workflow system at [8]. The actor can be combined with different computational models including SDF, PN and Comad as shown in [9].

**Window based Approach.** Given two input streams, a data stream $D$ and a window stream $W$, the Chunker will group input data for each window. The data stream is a sequence of pairs $(t_i, d_i)$ containing a timestamp $t_i$ and a value
The window stream is another sequence of pairs \( w_j = (b_j, e_j) \) containing the beginning \( b_j \) and the end \( e_j \) timestamp of the window. For each window in \( W \), the Chunker will store the window in a list of active windows if the start time \( b_j \) is smaller than the current time stamp in the data stream. For each active window \( w = (b, e) \in W \) the actor checks if incoming data is contained in that window, i.e., \( b \leq t_i \leq e \), and if so the data is stored with this window.

Since both \( D \) and \( W \) are possibly infinite streams, an order needs to be imposed on the timestamps \( t_i \) in \( D \) and on the start time of windows \( b_j \) in \( W \). Thus, groups can be built and output without storing a possibly infinite amount of data that would otherwise be stored in order to complete windows with start times reaching far back in time. Note that there is no requirement for an order on ending times of windows. In fact, for the Chunker actor, windows in \( W \) can overlap in an arbitrary manner. Whenever the timestamp \( t_i \) of the data stream reaches or exceeds the end time \( e_j \) of a stored window, this window can be output and removed for the list of active windows.

Since grouped data is frequently used in aggregation functions and windows can be large, our Chunker actor computes aggregates over windows incrementally. The workflow developer can choose if either the complete data for a window, a selection of aggregates, or both should be output. The algorithm supports all standard aggregations, that can be computed using an initialization state, and function that updates the state based on incoming data, and a function that finalizes the aggregation state for output. Currently supported aggregates are: count, sum, average, maximum, minimum, and array. The array aggregation collects a stream of input data, into a single array for each window. This is essentially emulating the behavior of the \texttt{Sequence to Array} actor described earlier but with support for variable-sized windows.

The Chunker implementation in Kepler requires tokens of type record on input streams. The window stream records need to provide an attribute \texttt{window} that holds a tuple containing the beginning and end time of the window. The data stream records need to provide at least two attributes, the \texttt{timestamp}, and the value. When Kepler sets up the workflow in preparation for execution, input and output ports are created to handle all streams. In addition, a priority queue is instantiated that is used to maintain the order of windows and as a result, windows are output ordered by window’s end timestamp or at their earliest possible computation. An \texttt{AggregationFactory} is provided as a generic

\[\text{Figure 5: Flexible, window-based Grouping: Incoming data tokens from } D \text{ are matched with all incoming windows } W. \text{ If the timestamp } t_i \text{ of a data token is enclosed in window } j: b_j \leq t_i \leq e_j \text{ then the value of this data token is grouped with this window. If configured by the user, the value is also used to update aggregations stored for the window. Once the end time of window } e_j \text{ has passed, i.e. incoming data tokens have a greater timestamp, aggregations are finalized and the completed window is output.}\]
interface to any aggregation. New aggregates are derived from the AggregationFactory by calling the create() method as shown in Algorithm 3.1. Each aggregate has *initialize*, *update* and *finalize* methods used for setup, incremental update and final computation of the aggregate function. The *finalize* method is invoked after all the required data has been processed and returns the aggregates final value.

**Algorithm 3.1:** fire()

```
cur ← data.getNext()
while windows.hasNext() and cur.timestamp() ≥ windows.peekNext().start()
    win ← windows.getNext()
    newAgg ← aggFac.create()
    newAgg.initialize()
    pq.insert(win.endTime(), newAgg)
while not pq.isEmpty() and pq.minKey() < cur.timestamp()
    minAgg ← pq.getMin()
    out.broadcast(minAgg.finalize())
    pq.apply(update(cur.value()))
```

Algorithm 3.1 is implemented in the *fire* method of the Chunker to compute groups and sliding window aggregates. First, a data token is accepted from the data port and stored in the variable *cur*. *cur* is checked to see if it is contained in a window enqueued on the channel. The check is performed by comparing the *cur* timestamp with the window’s start timestamp and if it is greater, then the window is retrieved from the *windows* port. For each retrieved window a new aggregate is created in *newAgg* and subsequently initialized, e.g. the aggregate sum is initialized with 0. Then the initialized aggregate *newAgg* is stored in the priority queue *pq* sorted by increasing end timestamp of *win*. Note that if *cur* timestamp is greater than *win* end timestamp, then the window *win* is still inserted into *pq*. The algorithm continues to process windows until the *cur* timestamp is less than the next windows start timestamp. After checking for windows, the algorithm processes all windows that are now present in the priority queue and sorted in increasing order of end timestamps. While the heap is not empty and *cur* timestamp is greater than the stop timestamp (*MinKey*) of the entry in front of the priority queue, an aggregate is retrieved from the queue, finalized by calling the corresponding function and output. In the last stage of the algorithm, the *apply* method is executed to update the aggregates of every remaining window in the heap with *cur* value. The postfire method of the Chunker actor returns true so that the fire method will be invoked again to process the next data token.

4. Growing Degree Days Workflow Using A Chunker

To demonstrate the functionality of our Chunker actor, we used it to model a GDD workflow. The workflow shown in Figure 6 demonstrates processing of sliding window aggregate queries on stream of hourly temperatures. The data stream is generated by querying a database containing temperature readings together with a timestamp. The WindowsGenerator actor creates a stream of windows defined by parameters specifying a start time and a variable length. Multiple WindowsGenerator actors can be used to create overlapping windows. For the purpose of calculating and plotting daily GDD and daily temperature averages windows are generated for the start of every day with a window size of one day. To calculate monthly averages overlapping windows with a start time of the first day of a month and a size to cover the whole month are used.

Another limitation of the conventional workflow design is the use of an R actor to calculate and plot the GDD data. The Scatterplot R actor of Figure 2 has no streaming capabilities, so that all required data has to be available before graphs are drawn. In our new design, we use a generic Expression actor, which is able to process streaming data, to calculate the GDD values. Furthermore, we use the SequencePlotter actor to display the final graphs. This actor supports streaming data and updates the graph view whenever new data is received.

Finally, the new workflow design uses a PN director that executes every actor of the workflow in a separate thread and therefore in parallel. In contrast to the conventional GDD workflow design, this does not imply a fixed numbers
of tokens flowing over channels since actors work independently and are only synchronized by writing to and reading from FIFO queues representing channels.

The workflow is considerably simpler than the one in Figure 2, and aggregations can be calculated independently of the number of measurements present every day, making the workflow more robust. The results of the computation with a time window of one day are equivalent to the one produced by the original workflow.

5. Conclusions

Kepler is a scientific workflow management system that exhibits a common problem regarding calculation of aggregations in scientific workflows. There are no mechanisms for computing sliding window aggregates based on timestamps apart from the very limited approach of counting tuples in the data stream. A more general Chunker actor has been developed that can process aggregations based on timestamps. The aggregator actor has shown to be a very useful tool in the world of scientific workflows, and Kepler in particular, for computing sliding window aggregates. It has eliminated the need for Sequence-to-Array based constructs employing token counting and instead directly supports extracting data from a stream within arbitrary time windows. Furthermore, this actor directly supports a selection of common aggregation functions in an efficient streaming manner.

As demonstrated for the GDD workflow, the Chunker actor provides much more flexibility than the conventional token counting approach in creating workflows for answering sliding window queries. Along with standard aggregates min, max, sum, count and average, an array aggregate is provided that collects a stream of input data in a single array for each window, corresponding to those data within that window. A possible future project would be to further extend the applicability of our actor and include user defined aggregates.

Acknowledgments.

This work was supported in part by NSF grants AGS-0619139, EF-0619060, OCI-0722079, and DOE grant DE-FC02-07ER25811.

References


