I²: Interactive Real-Time Visualization for Streaming Data

Jonas Traub
Technische Universität Berlin
jonas.traub@tu-berlin.de

Nikolaas Steenbergen
German Research Center for Artificial Intelligence (DFKI)
nikolaas.steenbergen@dfki.de

Philipp M. Grulich
German Research Center for Artificial Intelligence (DFKI)
philipp.grulich@dfki.de

Tilmann Rabl
Technische Universität Berlin
rabl@tu-berlin.de

Volker Markl
Technische Universität Berlin
volker.markl@tu-berlin.de

ABSTRACT

Developing scalable real-time data analysis programs is a challenging task. Developers need insights from the data to define meaningful analysis flows, which often makes the development a trial and error process. Data visualization techniques can provide insights to aid the development, but the sheer amount of available data frequently makes it impossible to visualize all data points at the same time. We present I², an interactive development environment that coordinates running cluster applications and corresponding visualizations such that only the currently depicted data points are processed and transferred. To this end, we present an algorithm for the real-time visualization of time series, which is proven to be correct and minimal in terms of transferred data. Moreover, we show how cluster programs can adapt to changed visualization properties at runtime to allow interactive data exploration on data streams.

1. INTRODUCTION

The amount of available real-time data increases rapidly with the growth of the Internet of Things. Such data is provided in the form of continuous data streams and includes various kinds of information such as stock prices, Twitter messages, Wikipedia edits, weather data, and GPS positions. Systems such as Apache Spark and Storm can process huge amounts of data with low latencies in a cluster to provide real-time analysis. Nevertheless, the development of analysis programs for these platforms remains a complex task, which requires insights about the processed data.

A visualization of the incoming datastream can provide such insights, but visualizing big data in real-time is a challenge itself. Since display capabilities are limited to a certain plot resolution (height and width of the screen) and local processing capabilities (e.g., a browser), it is usually impossible to show all individual data points from a high bandwidth data stream. For example, even though a time series may consist of 2000 measurements per second, the visualization of a second in a line chart is limited to a certain amount of pixel columns. Thus, a user has to trade off between the length of the shown history (time span covered on the time axis) and the resolution of the provided plot (available pixel columns per time) as shown in Figure 1a.

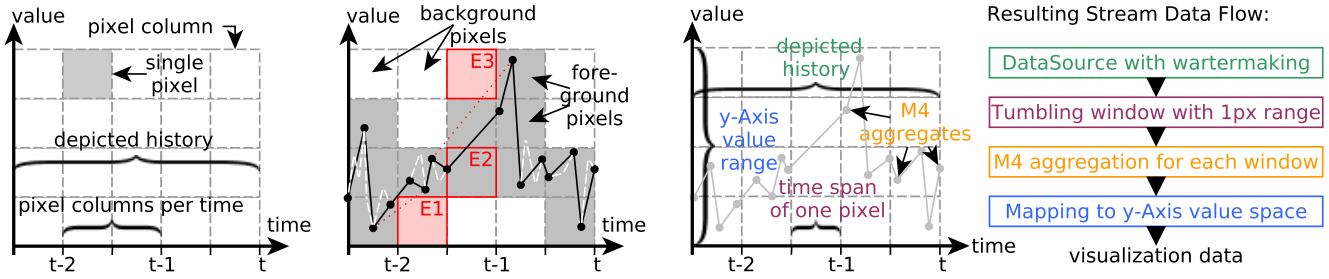
Interestingly, it is proven that the amount of data which is required to plot a correct line chart depends only on the number of pixel columns and not on the data. Jugel et al. [8] derive standard SQL queries from a given plot resolution and provide a loss-free plot from only four values per pixel column which reduces the computational load of the system. We show how the same values can be computed in a parallel dataflow program to allow the live visualization of incoming streaming data. Additionally, we take care of differences between event time and processing time as well as tuples arriving out-of-order, which makes processing streaming data a more complex task.

We integrate the efficient live visualization of time series as line chart together with other types of visualizations in I², our interactive development environment which connects distributed data analysis programs with the visualization of the results.[1] The name I² emphasizes two types of interactivity: *(i)* through code changes and *(ii)* through an interactive visualization GUI. With I², developers can change and deploy the code of analysis pipelines and corresponding result visualizations in a one-click fashion. Moreover, running applications adapt to changes in the visualization, e.g., if the user zooms into a map, and ensure that only the data points which are depicted in the current visualization are processed and transferred towards the front end. As a result, I² decreases the workload in the cluster backend as well as the visualization front end. Summarizing, our contributions are:

1. We present an interactive environment for visualization supported development of streaming cluster applications.
2. We show that our solution significantly reduces the amount of processed and transferred data while still providing loss-free visualizations.
3. We provide an algorithm for the live visualization of time series in line charts, which is proven to be correct and minimal in terms transferred data.

In our demonstration, we use I² for real-time event-based sport analytics. We therefore explore a data set from the DEBS 2013 Grand Challenge [10] consisting of more than 2.6 GB sensor data recorded at a football match with up to 2000Hz sampling rates. The data provides detailed real-time information about all players as well as the ball.

[1] https://github.com/TU-Berlin-DIMA/i2



(a) The tradeoff between depicted history and plot precision.

(b) The M4 aggregation technique for time-series data.

(c) Deriving a stream data flow program for the real-time visualization of time-series data with M4.

Figure 1: Efficient real-time visualization of time series data.

Related Work. In contrast to existing data exploration techniques [7], our demonstration combines three functionalities within a single environment: *(i)* the rapid development and deployment of cluster applications with streaming data, *(ii)* the automatic adaptation of running cluster jobs to changed visualization properties, and *(iii)* the efficient reduction of data to prevent overload of the visualization front-end. While other solution require an additional intermediate layer between database and visualization [4], I^2 directly integrates into data analysis applications. Other approaches like [1, 2, 9, 12] use sampling strategies for fast visualizations of huge amounts of data, but as opposed to I^2 disregard physical display properties and do not cover live plots of streaming data. Wu et al. [14] take into account visualization properties and automatically derive SQL-queries, but use a domain specific language. In contrast, I^2 works with any query language integrated in Apache Zeppelin.

In the remainder of this paper, we first present our solution for the visualization of time-series in line charts in Section 2. We then present the over-all architecture of I^2 in Section 3 and our demonstration in Section 4.

2. VISUALIZATION OF TIME SERIES

High volume time series data is omnipresent in many domains such as banking, weather data, facility monitoring, or, as in our demonstration, sport analytics. A naive approach for the visualization of time series would send all available data points towards the front end, which causes the visualization to crash in case the amount of input data increases as we will show in Section 4. The M4 aggregation technique [8] overcomes this limitation and constantly transfers just four values per pixel column. Furthermore, M4 is proven to provide loss-free plots compared to plots of the original data.

Figure 1b illustrates the functioning of the M4 aggregation. For each pixel column, M4 finds the minimum and maximum value as well as the first and the last value (minimum and maximum timestamp). All pixels which are crossed by the line connecting the extracted data points are colored and thus become foreground pixels. The intuitive approach to take only the minimum and maximum values into consideration would be insufficient. This would result in the red dotted line in Figure 1b and cause the pixel errors E1, E3 (wrongly colored) as well as E2 (not colored).

In I^2, we want to visualize streaming data in real-time. While M4 only considers finite data stored in a relational database, the real-time requirement adds several new challenges: instead of standard SQL queries, we now need *parallelizable processing pipelines*. Due to network delays and failures, there might be a gap between event time (the point in time a measure is taken) and processing time (the point in time the data is processed). Since data points may arrive out-of-order, we can never guarantee that the data for a pixel column is complete and possibly need to update past pixel columns in case of delayed input data. We address these challenges, as we derive a complete stream processing pipeline from a given plot resolution and the length of the depicted history as shown in Figure 1c. The pipeline mainly consists of four steps each of which can be executed as an operator with possibly multiple parallel instances.

Watermarks. Watermarks flow through the pipeline alongside the regular data and propagate the progress of event time. A watermark of time t_w means that no later processed event will have a timestamp $t_e < t_w$. We input watermarks at the data source of our pipeline to mark the smallest timestamp which is still covered by the live plot. Hence, we update pixel columns in case data arrives out-of-order. However, we avoid unnecessary processing of out-of-order data which arrives so late that the corresponding pixel column of the live chart is no longer displayed.

Windowing. We apply a time window function which splits the stream into finite data chunks spanning the time of one pixel column. We then compute the M4 aggregates over these windows and respectively for each pixel column. For the lack of space, we omit further details about the processing of out-of-order events and refer the reader to [5].

Value compression. Finally, we map the results of the aggregation to the value space of the y-axis which allows us to represent each value with less bytes.

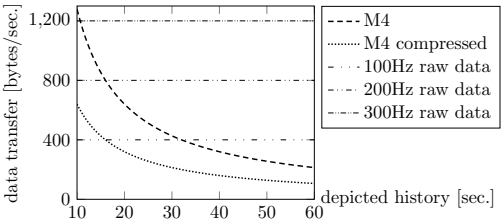


Figure 2: The required bandwidth for an 800x600px plot.

Figure 2 shows the savings in the input bandwidth of the visualization assuming an 800x600px plot showing 4 byte integer values. Note that the bandwidth required by M4 is independent from the frequency of the underlying raw data and solely depends on the length of the depicted history. The longer the depicted history, the more data is aggregated into one pixel column, which causes the required bandwidth to

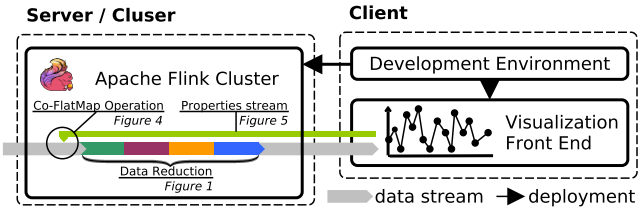


Figure 3: I^2 architecture overview.

decrease. In the next section, we show how our streaming ready M4 aggregation pipeline is integrated into the overall architecture of the I^2 development environment.

3. I² DEVELOPMENT ENVIRONMENT

The I^2 development environment aims to seamlessly connect live data visualization with the development of streaming data analysis pipelines. We, therefore, directly link a development environment and result visualizations within in a single front end (Figure 3). Developers can deploy data analysis pipelines as well as visualizations in a one-click fashion. While the visualization is provided within the same GUI as the code editor, the analytics pipeline is deployed on an Apache Flink cluster to be capable of processing high bandwidth streams in parallel.

Apache Flink [3, 5] is an open source platform for big data batch and stream processing. The basis of Flink is a fault tolerant execution engine. Programs are represented as operator graphs and the full processing pipeline is executed concurrently. Thus, the output tuples of an operator can be processed immediately by succeeding operators. Flink allows operators to have state. An asynchronous snapshot algorithm [6] ensures exactly once processing guarantees even in case of failures. Flink fits perfectly to I^2 since we need stateful operators to store current visualization parameters and low latency processing to quickly adapt running jobs to changes.

Apache Zeppelin. The I^2 front end is based on Apache Zeppelin, but was extended to support automatic data reduction depending on current visualization parameters. In general, Zeppelin aims to support quick development of programs, enabling interactive analytics in web based notebooks. It is similar to IPython [11], but focuses on large scale datasets and distributed computing. Zeppelin notebooks are data driven, interactive, and can be edited collaboratively by multiple users. Moreover, Zeppelin supports a variety of execution back ends. Zeppelin is not limited to classical dashboards; it also allows to develop source code, submit jobs directly to the cluster, and retrieve results immediately.

Runtime Adaptive Operators. I^2 informs running Flink jobs about changes of the visualization parameters. For example, if the user zooms into a map or changes the length of the depicted history of a time series plot. The running cluster program has to adapt to such changes with low latency in order to immediately provide the required data for the visualization. Since a redeployment of a job in the cluster can take more than a minute, we need to adapt jobs at runtime.

We push changes of the visualization parameters as control messages in a separate stream to the running Flink job. Only the type of an operator (e.g., filter or aggregation) is

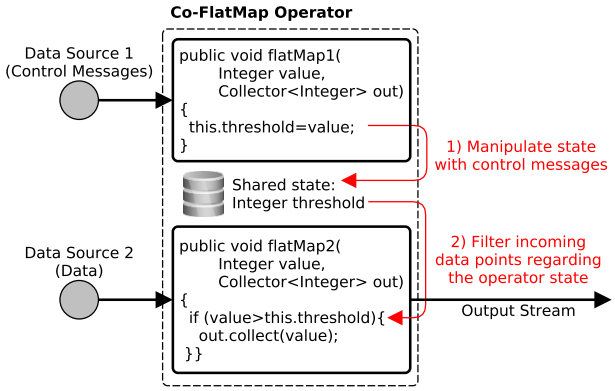


Figure 4: A runtime adaptive filter operator for variable thresholds in Apache Flink.

defined a priori, while we allow to adjust the parameters of the operator (e.g., filter predicate or aggregate function) on the fly at runtime. We use Flink's CoMap operators to process the control messages and the actual data points together in a shared runtime adaptive operator.

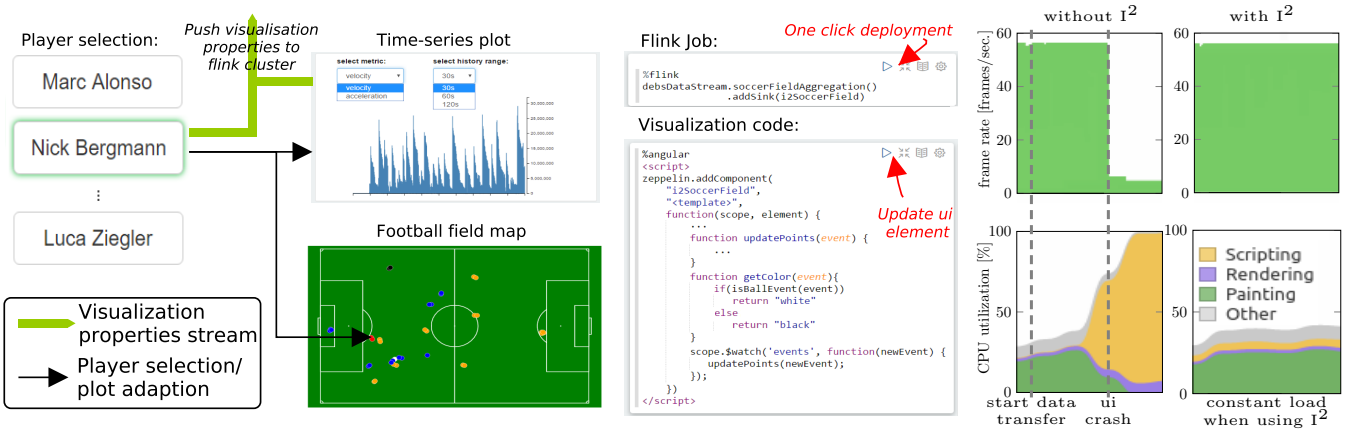
Flink's CoMap operators consume two input streams while input items from each stream are processed by separate *user defined functions* (UDFs). Nevertheless, both UDFs can access a shared operator state which is used to communicate between them. Figure 4 shows how we can utilize a CoFlatMap operator to adapt to changed properties: in this example, one input stream consists of control messages containing changes to the threshold of a filter operation. The responsible UDF saves the current threshold as operator state (Figure 4, 1). Each value from the actual data stream is compared to the currently stored threshold and all smaller values are filtered out (Figure 4, 2). In general, arbitrary changes to a selection criteria, aggregation function, windowing semantics, and other operations are possible using this architecture.

4. DEMONSTRATION

In our demonstration, we allow the visitor to experience the fast visualization supported development with I^2. This covers the development of the Flink job running in the cluster as well as changing the visualizations. At the same time, we continuously show the savings in terms of the transferred data volume which are archived by I^2. When we increase the data rates of the input streams, I^2 will hide that workload from the visualization while without using I^2 the front end would first become unresponsive and finally crash.

Data. We replay the data set which was provided with the DEBS Grand Challenge 2013 [10]. This data set consists of sensor data, which was recorded at a football match. The speed, acceleration, and position of the ball are tracked with a frequency of 2000Hz. In addition, each player has two sensors close to his shoes which are tracked with a 200Hz frequency. In total, roughly 15.000 data points are provided for each second of the match.

Demonstration. We show an interactive dashboard to analyze the performance of individual players in detail. Users can either select a player manually or automatically follow the ball possession, which involves detecting peaks in the measures of the ball sensor as well as correlating these peaks with the data from the player sensors. Our dashboard shows



(a) Interactive Dashboard (b) Development Environment (c) Performance Monitoring

Figure 5: Selected screenshots from the I^2 demonstration.

different metrics (e.g, acceleration and speed) for the selected player as well as the player's current position on the football field (Figure 5a).

We first try to run our dashboard without using I^2, meaning that no data reduction is applied and all data - roughly 15.000 tuples/sec. - is transferred towards the frontend. As shown in Figure 5c (left), the UI works only for a short moment before it becomes unresponsive due to a CPU overload.

We now run the same dashboard with I^2, pushing the current visualization properties to the running Flink job as described in Section 3. This information is then used by Flink to apply different data reduction techniques: knowing the currently selected player enables adaptive filtering as shown in Figure 4 and knowing the plot resolution of line charts allows to apply the M4 aggregation technique we presented in Section 2. The soccer field map combines different data reduction techniques. We reduce the precision of the position reports based on the plot resolution and at the same time apply load shedding [13] to reduce the data rate to the current frame rate of the visualization. As shown on Figure 5c (right), the presented dashboard runs fluently when using I^2 with close to 60 frames per second and a CPU utilization below 50%.

Interactivity. We demonstrate the two types of interactivity provided by I^2. First, we show that visualization properties can be changed easily in the dashboard and that the running Flink job adapts with low latency to e.g., changes in the player selection or the length of the depicted history of line charts.

Second, we demonstrate the interactivity through code changes. Interactive code changes allow an even more flexible data exploration and the rapid development of cluster applications. We first show how the code for the visualizations can be adapted and directly deployed without a need to restart the running Flink job. We then show how we can connect an additional data source for twitter messages and how these messages can be correlated to the data we used before. The extended Flink job is directly deployed to the cluster with just one click.

Evaluation. We exemplary compared the performance of I^2 for the dashboard described above (Figure 5c). Our experiment showed that the amount of transferred data, the memory utilization, the CPU load, and the frame rate remain constant throughout the game when I^2 is active. Switching of I^2 causes the visualization to become unresponsive immediately due to the massive amount of arriving data. With activated I^2, the bottleneck is no longer the visualization, but the power of the used Flink cluster.

5. CONCLUSIONS

I^2 enables two types of interactivity: first, the user can specify real-time analysis programs and change them on the fly. Second, the interactive visualization of the results adapts currently running cluster applications without a need to restart. Using I^2, the amount of data points to be processed and transferred to the front end can be reduced significantly without quality loss, enabling the live visualization of high bandwidth data streams. The capabilities of I^2 have been demonstrated in an interactive example using real-world data.

Acknowledgements. This work was supported by the EU Horizon 2020 project Streamline (688191) and the German Ministry for Education and Research as Berlin Big Data Center (01IS14013A) and Software Campus (01IS12056).

6. REFERENCES

[1] S. Agarwal et al. BlinkDB: queries with bounded errors and bounded response times on very large data. EuroSys, 2013.
[2] S. Agarwal et al. Knowing when you're wrong: building fast and reliable approximate query processing systems. SIGMOD, 2014.
[3] A. Alexandrov et al. The stratosphere platform for big data analytics. *VLDBJ*, 2014.
[4] L. Battle et al. Dynamic reduction of query result sets for interactive visualizaton. IEEE BigData, 2013.
[5] P. Carbone et al. Apache Flink^{TM}: Stream and batch processing in a single engine. *IEEE Data Eng. Bulletin*, 2015.
[6] P. Carbone et al. Lightweight asynchronous snapshots for distributed dataflows. *arXiv*, 2015.
[7] S. Idreos et al. Overview of data exploration techniques. SIGMOD, 2015.
[8] U. Jugel et al. M4: a visualization-oriented time series data aggregation. *VLDB*, 2014.
[9] A. Kim et al. Rapid sampling for visualizations with ordering guarantees. 2015.
[10] C. Mutschler et al. The DEBS 2013 grand challenge. 2013.
[11] F. Pérez and B. E. Granger. IPython: a system for interactive scientific computing. *CISE*, 2007.
[12] L. Sidirourgos et al. Scientific discovery through weighted sampling. IEEE BigData, 2013.
[13] N. Tatbul et al. Load shedding in a data stream manager. VLDB, 2003.
[14] E. Wu et al. The case for data visualization management systems: vision paper. *VLDB*, 2014.