ProgressiVis: a Toolkit for Steerable Progressive Analytics and Visualization

Jean-Daniel Fekete*
INRIA

ABSTRACT

ProgressiVis is a Python toolkit that implements a new *programming paradigm* that we call *Progressive Analytics* aimed at performing analytics in a progressive way. It allows analysts to see the progress of their analysis and to steer it while the computation is being done. While there have been a lot of articles arguing for introducing progressive visualization for visual analytics, ProgressiVis's novel computation paradigm allows the practical implementation of progressive visualization and computation in a general way with a reasonable complexity for the programmer.

Instead of running algorithms to completion one after the other, as done in all existing scientific analysis systems, ProgressiVis modules run in short batches, each batch being only allowed to run for a specific quantum of time—typically 1 second— producing a usable result in the end, and yielding control to the next module. To perform the whole computation, ProgressiVis loops over the modules as many times as necessary to converge to a result that the analyst considers satisfactory.

This article introduces the new paradigm and its prototype implementation, provides some example, and lists some implications for future work.

Index Terms: D.2.13 [Reusable Software]: Reusable libraries—; D.3.2 [Language Classifications]: Specialized application languages—.

1 INTRODUCTION

In this article, we present a novel *programming paradigm* that we call *Progressive Analytics* to analyze, steer the analysis, visualize, and interact with data in a progressive way; we also describe a prototype implementation called ProgressiVis. ProgressiVis allows analysts to perform exploratory analysis: monitoring the progress of their computations and steering it interactively. This monitoring and steering is an efficient way to *explore* large amounts of data; it prevents analysts from waiting idly for computations to complete.

Several software and hardware infrastructures have been developed in the recent years to analyze large amounts of data, including scientific workflows [2, 19], Grid- or Cloud-based systems [5, 31, 27, 24]. However, these infrastructures are mostly aimed at improving distribution of computation and data management, allowing very large and complex applications to use vast amounts of data, providing a high computation throughput at the cost of a high latency. This latency is incompatible with exploratory analysis [8, 14, 6] that needs a tight interaction loop to see overviews of data, try several algorithms to assess which one will find or reveal useful properties of the data. When the computation loop exceeds a few seconds, exploration becomes harder for humans since their short-term memory cannot be used to remember exploration plans and competing hypotheses.

The next sections describe ProgressiVis, some examples of its use, and discuss its potentials before concluding.

*e-mail: jean-Daniel.Fekete@inria.fr

2 BACKGROUND

In the recent years, there have been a growing number of articles related to visual analytics, visualization and interaction with large amounts of data [7, 17, 18, 28]. While some of the articles describe high-performance information visualization systems [17, 18], others are trying to change the visualization paradigm to allow some kind of progressive, iterative, or incremental visualization, moving from a synchronous world where operations are performed instantaneously (as perceived by the user), to an asynchronous one where operations can take time to perform and the system as well as the user should adapt. With the continuous growth of datasets, the synchronous approach will reach its limits, although these limits are quite high and allow the visualization and exploration of multi-millions of records.

The literature has been describing the issues related to visualizing large amounts of data, and in particular the need to aggregate data to avoid over-plotting [18, 4, 25, 28, 29], few address the general problem of visual analytics with big data [28, 7], and no general framework for visual analytics on big data has been described in the literature so far.

Scientific workflows have been designed to perform long and complex computations, sometimes using distributed parallel systems. However, they are not meant for exploration: they need a complete description of the operations to perform to the data before running, and will run to completion before allowing the user to change the analysis parameters. Systems such as VisTrails [2] are better suited to exploration but long-lasting computations still hamper its exploratory capabilities. We have explored in the EdiFlow system [1] several strategies to manage analytics workflow, but it was driven by data-table changes. Still, ProgressiVis reuses some of these strategies.

Finally, as mentioned in the introduction, there have been a lot of new software and hardware architecture design to carry long and complex computations on huge amounts of data [5, 31, 27, 24], but these frameworks provide high-throughput with high-latency, incompatible with the exploratory process.

While ad-hoc systems are developed to address the needs of interactive exploratory data analysis, no general system or architecture has been proposed so far to tackle the problem. *Progressive Analytics* is meant to remedy this problem with ProgressiVis as a prototype implementation.

3 PROGRESSIVIS

The fundamental idea of ProgressiVis consists in splitting long computations into small chunks, each of these chunks delivering some output, either incomplete or approximate, but still useful to the analysis. ProgressiVis implements the idea of computing by chunks by providing workflow modules that will be run as many times as needed to complete their work, and will deliver *useful* results at each run bounded by a specified amount of time that we call *quantum*. After they consume that quantum of time, they yield the execution to other modules. This execution strategy based on nonpreemptive cooperative unit of execution sometimes called *Fiber*, or with many other names such as *cooperative multi-tasking*, *nonpreemptive multi-tasking*, *user-level threads*, and *green threads*. However, to our knowledge, splitting a whole analytics pipeline into progressive chunks that should deliver useful data before yielding control is novel and unexplored.

In term of implementation, it turns out that a large number of analytics algorithms and related components (e.g. database loading, conversions) can be adapted to work by chunk, and visualization can also be adapted to display progressive information arriving in chunks. Therefore, we believe ProgressiVis provides a good tradeoff in terms of performance, scalability, and amount of code to write to address a large number of problems at scale.

3.1 Aggregated Scatterplot Example

Let's take as simple example the implementation of an aggregated scatterplot that scale. It is made of a Heatmap in the background, which uses a 2d histogram computation rendered as an image [4] with some well-chosen color transfer function, on top of which a sample of landmark points with labels is visualized to provide contextual information. Its implementation with ProgressiVis consists in loading a tabular dataset, computing the 2D histogram of 2 of its columns using e.g. 512×512 bins, and rendering it for displaying on screen. There is an intermediary step that is usually implicit in non-progressive systems, that consists in computing the min/max values of the 2 columns to visualize because the 2D histogram needs these values to compute its bins. Implementing this pipeline with ProgressiVis requires at least 3 modules: a dataset loader, a min-max computation module, and a histogram computation and rendering module, in addition to the visualization module that binds all the modules together. The dataset will be loaded progressively and delivered chunk by chunk to the min-max computation. This module will in turn deliver approximations of the final bounds of the 2 data columns to the histogram computation that will eventually compute the histogram bins and render them. The overlay scatterplot of samples could just perform a random sampling of the data, or a more sophisticated computation of landmarks or computation of "level-of-interest" to the points being loaded for later selection.

A naive implementation like we just described would work but the histogram computation will need to restart from scratch frequently initially because the min-max module will likely generate varying values, slowly converging to the real bounds. Three simple strategies can be used to avoid too many computations at that stage: 1) loading the data in random order, 2) delaying the histogram computation to a time when the bounds stabilize, and 3) slightly extending the bounds used for the histogram computation after the initial runs to account for their likely evolution. These three strategies are all conceptually simple, although loading data in random order is not always easy to do efficiently. As a final note, when the histogram gets computed, it will accumulate the number of points in each bin so the computation will generate substantially new results as data gets processed. However, the final image for the Heatmap shows normalized values (the bin counts divided by the max bin count, or a simple transformation of that fraction), and these values should become stable when more data become aggregated, so the histogram module can avoid generating new images if normalized pixel intensities do not change after a certain number of runs. Therefore, the display will not have to change to update the latest results.

By using both aggregated views and sampled details, this visualization technique can work with unbounded amounts of data; it might take some time to converge to a steady state, but a useful overview might still be visible quickly if the visualized points are correctly sampled. If not, arguably, seeing something might be better than waiting and seeing nothing. Finding an algorithm to select useful samples of a large datasets that is progressively loaded is still an open research question. However, the aggregated scatterplot technique will still work with any sampling strategy; it becomes more effective with better sampling strategies.

4 Progressive MDS

The progressive scatterplot example is bounded in throughput by the database transfer speed (I/O bound): its accuracy is mostly limited by the loading time. Analytics algorithms are more often bounded in throughput by the computation itself. This is the case for high-dimension projection techniques such as MDS [15], which can take hours to compute a decent projection for a large dataset.

We have investigated strategies for implementing a progressive version of MDS. These strategies build on prior work aimed at introducing steering in MDS [30], hybrid methods for fast MDS computations by Chalmers et al. [21], and recent work by Munzner and Ingram to accelerate MDS computation for expensive distance functions and text corpora visualization [13, 11, 12]. An MDS algorithm, given a set of points and a distance function in high-dimensional space, finds a 2D projection of the points where Euclidean distances between the 2D points are as similar as possible as their high-dimensional distances.

We started with the implementation of the SMACOF [16] algorithm in Scikit-Learn [22], which has a complexity of $O(n^3)$ where n is the number of points. The implementation is nicely split in two functions: a *driver* function called `smacof` that performs a lot of sanity checks, distance matrix computation and verification, and iterates on a lower-level function called `_smacof_single` to compute a solution with a low stress.

The inner iteration of the algorithm passes over all the points and performs one step of a gradient descent to converge to the minimal "stress" (discrepancy between the desired distances and the actual ones). However, the algorithm needs a valid distance matrix to start with, and this matrix can be costly to compute [11]. Our implementation of MDS requires two modules: one to compute the high-dimensional distances between every pairs of points, and the second to compute the projection. The Scikit-Learn implementation can cope with this separation, but will always check the validity of the distance matrix, which is an expensive quadratic computation. Since we know what module has performed the distance computation, our MDS module can take its results for correct to save time.

Our module uses the Scikit-Learn implementation with limited changes: it only removes the initial check and selects one variant of SMACOF (the metric variant since we provide the complete distance matrix). As it is, the module can perform a limited number of iterations per run, but will still converge to a solution with enough time. The problem is that, when more points arrive, the inner loop takes more time than the quantum and the algorithm cannot even perform one step per run. Since the algorithm only updates its 2D projection at the end of each run, the user cannot see the progress of the algorithm, defeating the purpose of our progressive paradigm.

However, Chalmers and his colleagues have extensively explored solutions to this problem [21]. They have introduced several heuristics to provide a reasonable initial value to newly inserted points using spring forces and interpolations; they also have shown that the algorithm can be fed with samples of the points and still converge to a good solution. They call this variant "Stochastic MDS". Therefore, by applying Chalmers' techniques, the inner loop can still produce useful results on newly arriving points while updating the older points progressively, offering a useful progressive output to the MDS module even for large datasets. However, it does so with tradeoffs. The algorithm could spend more time on improving the computed configuration, or could try to incorporate as many new points as possible, at the cost of a lower quality (higher stress). More work is needed to find automatic methods to choose the best tradeoff, but meanwhile, this choice can be given to the user with some control parameters. In addition, parameters for steering the computation can also be given to the module through input parameters, and controlled interactively, leading to a Steerable MDS implementation similar to the one described in [30].

5 Interaction and Steering

Standard interactions on an aggregated scatterplot include filtering, zooming, and details on demand. On a computation module such as the progressive MDS described before, it includes specifying the tradeoff between increasing the quality of the existing configuration or adding more points at each iteration, as well as specifying a selection of points where more quality is desired, e.g. by providing a viewport of interest. These operations should be fed back either to the sampling and aggregation modules for the scatterplot, or to the distance computation and MDS module for MDS computation. However, feedback loops are forbidden in existing workflows or dataflow languages because they prevent the computation of a topological sort of the modules.

However, in the case of our progressive system, the constraint that the directed dependency graph should be acyclic (DAG) can be relaxed a bit. ProgressiVis modules have mandatory input slots and optional ones. For example, the 2D histogram computation module needs a set of (x,y) coordinates, but has an optional input slot to control its bounds; by default, it is set to $(0,0,1,1)$. This slot can be connected to the output of the min/max computation, or to the selected bounds of the interactive visualization module that will allow interactive zooming and panning by changing the input slot of the histogram module. In that case, the visualization module takes its input from the output of the histogram module, and the histogram module takes parameters from the output of the visualization module, creating a cycle.

When ProgressiVis computes a topological order for its modules, it starts by taking into account all the connections and, if a cycle is detected, it recomputes the topological order taking into account the mandatory connections only. This should form a DAG and an error is raised if a cycle is found with these mandatory connections, but the other optional connections will still be taken into account in the end since the scheduler will call all the modules in a round-robin fashion.

Although running all the modules of a workflow can take several seconds with progressive computation, depending on the number of modules and the quantum of time allocated to them, the visual feedback of filtering and zooming can be immediate, although partial. This is what we experience when using a map on the Web: zooming and panning is immediate, even if actual map tiles arrive a bit later due to network delays. Therefore, modules can be controlled and steered through optional input slots, unlike conventional workflow systems where the module computation is all performed in one step. This raises a lot of interesting questions on how best to manage this kind of feedback loops, both in terms of expressive power, management of lag, but also smart scheduling specifically crafted for interactivity.

6 Implementation Details

ProgressiVis uses two threads to work, one is the user thread, where the analyst uses the Python language and IPython notebook to explore data; the other is the running thread where ProgressiVis modules run in the background. Python threads are notoriously inefficient, but for our setting where the user interaction thread is mostly idle, it performs reasonably well.

ProgressiVis relies on existing Python libraries and toolkits and tries to be compliant with the Python scientific computation ecosystem. In particular, the visualization modules are currently implemented to work in *IPython notebook*, which is a web-based interface to execute Python instructions. Recently, the *IPython notebook* has been renamed "Jupyter" notebook, and enriched with interaction capabilities; we use the "Bokeh" visualization toolkit to generate interactive visualization in the notebook (and therefore in the browser). Although the integration is not straightforward, it is feasible thanks to the elegant implementation of the Jupyter notebook.

A complete implementation of ProgressiVis requires several important and non-standard mechanisms that we describe now:

1. a unified representation of data;
2. a mechanism to manage the changes performed by modules so that the subsequent modules know what happened before;
3. a mechanism to manage the time to run in modules.

6.1 Unified Representation of Data

We chose to use Pandas *Data Frames* [20] to represent most of the data managed by ProgressiVis. Data Frames are similar to database tables and popular in statistical software such as R [23]. Most of the values exchanged by ProgressiVis modules are data frames that we extend with a column representing the time when rows have been updated. Therefore, the data frames are decorated with an additional column called `_update` that contains a number (not a date). The scheduler starts its first iteration over all the modules with a run number 1, and increments it every time, producing a virtual time: the *run number* that we use to keep track of when rows have been updated. This run number is then used to know what needs to be updated in the dependent modules.

The scheduler maintains a table that maps virtual time to absolute time. The advantages of using virtual time rather than real time are that: 1) managing an integer is much easier than a date, even when it is encoded as a long value, and 2) the time of a particular run is only known when it finishes whereas the virtual time is assigned at the start of each iteration.

Using one column to keep track of changes in data offers limited precision to the nature of the changes. One value can be changed, or the whole row created or changed: the `_update` column does not tell what happened. However, in our experience, this level of details is enough for most applications. In addition, users can model their problems with multiple data frames if they want a higher precision on the updates.

6.2 Change Management in Modules

When modules receive data frames, they usually need to know what happened to the data since the last run when they were scheduled. A *change manager* provides that information, using the `_update` column and additional information that it stores in each module input slot. This information boils down to what data has already been processed, and sometimes a buffer of data to process.

From that information, each module can get the set of created items, the set of updated items, and the set of deleted items. In addition, if some items have been buffered for further management in the previous runs, they are still in the buffer ready to be managed.

With these potentially three sets, each module can decide how best to pursue its computation. When only new data is available, module that manages their input incrementally can just continue to do so. For example, a module computing the min/max values of data frames can just continue updating these values for the incoming date. If previous data has been modified or deleted, these modules should either restart their computation from scratch, or try to repair it if they have maintained a backup of managed values. In the case of simple modules, such as the one performing min/max computations, recomputing the values is fast (typically millions of values per second). For other modules, the recomputation can be costly.

Modules such as MDS can actually incorporate changed values in a simple way without keeping track of old values, since the iterative computation of MDS will eventually "fix" the projection of the changed values once the distance computation module has recomputed the distances for the changed rows.

Basically, each module needs to decide how to react to value changes, and the best strategy depends on the semantic of the module. In the worst case, the computation can be restarted from scratch, but in typical cases, a much smarter strategy is possible.

A particular case of value change is when a control parameter is modified, usually interactively. As discussed in the previous section, these parameters can be used to tune the behavior of a module, such as controlling how much time will be spent by the MDS module on incorporating new values versus improving the stress of already incorporated points. However, when e.g. the viewport of a 2D histogram is changed, the histogram module should restart the computation from scratch. A more interesting case occur when changing the distance function for MDS; the distance computation module should restart from scratch, but the MDS module can start its computation with the actual 2D positions already computed so, from a user's perspective, the MDS will morph from the old configuration to the new one, helping users to follow the changes.

6.3 Time to Run Management in Modules

Although the principle of allowing a module to run for a specified quantum of time is intuitively simple, implementing this behavior is not that simple. In the worst case, each module could measure the time after each instruction and stop when getting close to the quantum, but measuring the time continuously is costly, and some operations are much faster when done in batches than item per item. For example, computing the minx/max values of data frame columns uses optimized vectorized code that is an order of magnitude faster when done item per item, especially with an interpreted language like Python. Still, this time will vary from one computer to another, and depending on the computer's load.

To address this problem of run-time prediction, ProgressiVis uses a *Time Predictor*. This manager uses a trace of the dynamic behavior or modules collected by the scheduler, to estimate the number of internal steps each module should perform to match its quantum.

For example, when reading a CSV file, the CSV loader will initially read a small number or rows (from 800 to 1200 in our case) and measure the time it takes using a linear regression. After a few internal iterations, it will have a decent approximation of the number of rows per second that it can read for the specific dataset (number of columns) and file-system or communication channel. With this more accurate approximation, the module can read more (or less) rows to match its quantum efficiently.

When a module implements an algorithm that is not linear in the number of items processed, the Time Predictor is still effective because it will compute a value close to the first derivative of the processing speed (number of items per second). This estimation is useful for the module, even if not completely accurate.

Whereas the Time Predictor manages modules by default, it can be overridden if a module needs finer control. This is the case of the MDS module that needs to know when it will reach its limit and should become stochastic. Still, the predictor provides useful information to the MDS module to know how long it takes to incorporate new items, and how long it takes to improve the stress function of older items.

7 Discussion

ProgressiVis is a prototype implementation of progressive computation and it is legitimate to wonder what progressive computation can and cannot do, and what progressive visualization problems should be addressed. Also, ProgressiVis needs to rely on existing components and toolkits that can sometimes be adapted for progressive computation and sometimes not. We try to summarize the most important issues for analytics, data management, and visualization.

Progressive Analytics Not all the popular machine learning and more generally analytics algorithms can be implemented to run in a progressive way, but a large number of the standard high-level operations can be done in a progressive way. For example, implementing hierarchical clustering in a progressive way seems quite difficult since the dendrogram hierarchy might change dramatically from one step to the next, but many clustering algorithms can work in a progressive way. The Scikit-Learn toolkit provides an implementation of K-Means clustering called "Mini-batch K-Means" [26] that can be turned into a progressive module. Looking at the implementation of the Scikit-Learn toolkit, a large number of algorithms it provides can also be made progressive. In particular, among the "Supervised Learning" algorithms, some generalized linear model solvers (e.g. Ordinary Least Square, Stochastic Gradient Descent, Polynomial regression), Support Vector Machines (SVM), etc. We are in the process of creating a repertoire of analytical methods that can be made progressive, and we started integrating the easy ones in ProgressiVis.

Progressive Data Management Loading large amounts of data prior to computation is I/O bound and slow compared to simple computations. Doing it in a progressive way mitigates the waiting-idly problem but some extra support would be useful from databases. In particular, two important features would greatly enhance the user experience: delivering query results in random order, and return queries progressively. While some databases provide random sampling, being able to load a selection in random order would improve the convergence of stochastic algorithms and avoid artifacts when loading and visualizing datasets. Progressive queries have been discussed by Fisher et al. [9] for aggregate queries, and implemented with extensions by Chandramouli et al. in the Trill system [3], but would be useful in standard databases.

Progressive Visualization Visualizing data that are dynamically changing is a novel problem in visualization, addressed by a few articles [28, 10]; more research is needed to know the kinds of support required to avoid distraction. For example, when monitoring the progress of an algorithm, the visualization should be updated as often as possible while trying to keep some consistency/continuity. However, when the user is exploring a particular view, e.g. a dense area in a plot created from a multidimensional scaling, changing the visualization at each run would be distracting and confusing. Therefore, the user should be given some control to switch between a monitoring mode and an exploratory mode, and probably to move back and forth in the history of the computation.

For network data (graphs), several systems already show the result of iterative layout algorithms in real time. Again, showing graphs where vertices and edges move is not always appreciated by analysts and more research is needed to understand how to best display the evolution of graph layouts with time.

Several systems have already tackled the visualization scalability problem by constructing smart indexes to provide immediate feedback for exploration and filtering (e.g. [17]). However, constructing these indexes takes time, so progressive methods can be used to provide some information to analysts while indexes are being constructed. We also believe that indexes can also be computed and updated progressively.

8 Conclusion

We have described ProgressiVis, a toolkit for Steerable Progressive Analytics and Visualization. ProgressiVis is still a prototype, available at `http://github.com/jdfekete/progressivis`, and will be extended in the forthcoming months on several types of applications to test its applicability. We have just scratched the surface of Progressive Analytics so far and found the new solutions and problems promising and inspiring, hopefully opening a new line of research to tackle exploratory analysis and visualization on large data for the next 10 years or so. Still, there are numerous extensions that are obvious but probably hard to tackle. Currently, ProgressiVis is not parallel and not distributed, which simplifies its implementation and use greatly. Extending it would be extremely interesting but we will initially focus on extending the sequential version before moving to the daunting task of making it parallel.

9 ACKNOWLEDGMENTS

This work has been conducted during my Sabbatical at NYU-Poly in the Visualization and Data Analytics (ViDA) group, partially funded by the Center for Data Science at NYU. Thanks to Juliana Freire for the fruitful feedback she provided me, to Marie Douriez for her help in implementing and experimenting Progressive MDS, and to Andreas Müller for his advises on Scikit-Learn.

REFERENCES

[1] V. Benzaken, J.-D. Fekete, P.-L. Hémery, W. Khemiri, and I. Manolescu. EdiFlow: data-intensive interactive workflows for visual analytics. In *ICDE 2011*, pages 780–791, 2011.

[2] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo. VisTrails: visualization meets data management. In *SIGMOD Conference*, pages 745–747, 2006.

[3] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *Proc. VLDB Endow.*, 8(4):401–412, Dec. 2014.

[4] J. A. Cottam, A. Lumsdaine, and P. Wang. Abstract rendering: out-of-core rendering for information visualization. *Proc. SPIE*, 9017:90170K–90170K–13, 2013.

[5] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[6] J.-D. Fekete. Visual Analytics Infrastructures: From Data Management to Exploration. *Computer*, 46(7):22–29, July 2013. notable article in computing in 2013 from Computing Reviews' Best of 2013.

[7] J.-D. Fekete, P.-L. Hemery, T. Baudel, and J. Wood. Obvious: A meta-toolkit to encapsulate information visualization toolkits; one toolkit to bind them all. In *Visual Analytics Science and Technology (VAST), 2011 IEEE Conference on*, pages 91–100, Oct 2011.

[8] J.-D. Fekete and C. Silva. Managing Data for Visual Analytics: Opportunities and Challenges. *IEEE Data Eng. Bull.*, 35(3):27–36, Sept. 2012.

[9] D. Fisher, I. Popov, S. Drucker, and M. Schraefel. Trust me, i'm partially right: incremental visualization lets analysts explore large datasets faster. In *CHI '12*, pages 1673–1682, 2012.

[10] M. Glueck, A. Khan, and D. J. Wigdor. Dive in!: Enabling progressive loading for real-time navigation of data visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 561–570, New York, NY, USA, 2014. ACM.

[11] S. Ingram and T. Munzner. Glint: An MDS framework for costly distance functions. In *Proceedings of SIGRAD 2012, Interactive Visual Analysis of Data, Växjö, Sweden, November 29-30, 2012.*, pages 29–38, 2012.

[12] S. Ingram and T. Munzner. Dimensionality reduction for documents with nearest neighbor queries. *Neurocomputing*, 150, Part B:557 – 569, 2015.

[13] S. Ingram, T. Munzner, and M. Olano. Glimmer: Multilevel MDS on the GPU. *Visualization and Computer Graphics, IEEE Transactions on*, 15(2):249–261, March 2009.

[14] Jean-Daniel Fekete. Infrastructure. In Daniel Keim and Jrn Kohlhammer and Geoffrey Ellis and Florian Mansmann, editor, *Mastering The Information Age - Solving Problems with Visual Analytics*, chapter 6, pages 87–108. Eurographics Assoc., 2010.

[15] J. B. Kruskal and M. Wish. *Multidimensional Scaling*. Sage Publications, Beverly Hills, 1978.

[16] J. D. Leeuw and P. Mair. Multidimensional scaling using majorization: SMACOF in R. *Journal of Statistical Software*, 31(3):1–30, 2009.

[17] L. Lins, J. T. Klosowski, and C. Scheidegger. Nanocubes for real-time exploration of spatiotemporal datasets. *Visualization and Computer Graphics, IEEE Transactions on*, 19(12):2456–2465, Dec 2013.

[18] Z. Liu, B. Jiang, and J. Heer. imMens: Real-time visual querying of big data. In *Proceedings of the 15th Eurographics Conference on Visualization*, EuroVis '13, pages 421–430, Chichester, UK, 2013. The Eurographs Association & John Wiley & Sons, Ltd.

[19] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the Kepler system: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(10):1039–1065, Aug. 2006.

[20] W. McKinney. pandas: a foundational python library for data analysis and statistics. *Python for High Performance and Scientific Computing*, pages 1–9, 2011.

[21] A. Morrison, G. Ross, and M. Chalmers. Fast multidimensional scaling through sampling, springs and interpolation. *Information Visualization*, 2(1):68–77, 2003.

[22] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[23] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2013. ISBN 3-900051-07-0.

[24] S. R.-C. Romain Reuillon, Mathieu Leclaire. OpenMOLE, a workflow engine specifically tailored for the distributed exploration of simulation models. *Future Generation Computer Systems*, 29(8):1981 – 1990, 2013.

[25] H.-J. Schulz, M. Angelini, G. Santucci, and H. Schumann. An enhanced visualization process model for incremental visualization. *Visualization and Computer Graphics, IEEE Transactions on*, PP(99):1–1, 2015.

[26] D. Sculley. Web-scale k-means clustering. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 1177–1178, New York, NY, USA, 2010. ACM.

[27] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10, May 2010.

[28] C. D. Stolper, A. Perer, and D. Gotz. Progressive visual analytics: User-driven visual exploration of in-progress analytics. *Visualization and Computer Graphics, IEEE Transactions on*, 20(12):1653–1662, Dec 2014.

[29] C. Stolte, D. Tang, and P. Hanrahan. Polaris: a system for query, analysis, and visualization of multidimensional relational databases. *Visualization and Computer Graphics, IEEE Transactions on*, 8(1):52–65, Jan 2002.

[30] M. Williams and T. Munzner. Steerable, progressive multidimensional scaling. In *Information Visualization, 2004. INFOVIS 2004. IEEE Symposium on*, pages 57–64, 2004.

[31] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.