RC25382 (WAT1305-032) May 8, 2013
Computer Science

IBM Research Report

Visualizing Jobs with Shared Resources in Distributed Environments

Wim De Pauw, Joel Wolf
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 208
Yorktown Heights, NY 10598
USA

Andrey Balmin
IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099
USA

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Visualizing Jobs with Shared Resources in Distributed Environments

Wim De Pauw
IBM T.J. Watson Research
Yorktown Heights NY, USA
wim@us.ibm.com

Joel Wolf
IBM T.J. Watson Research
Yorktown Heights NY, USA
jlwolf@us.ibm.com

Andrey Balmin
IBM Almaden Research
Almaden CA, USA
abalmin@us.ibm.com

***Abstract*— In this paper we describe a visualization system that shows the behavior of jobs in large, distributed computing clusters. The system has been in use for two years, and is sufficiently generic to be applied in two quite different domains: a Hadoop MapReduce environment and the Watson DeepQA DUCC cluster. Scalable and flexible data processing systems typically run hundreds or more of simultaneous jobs. The creation, termination, expansion and contraction of these jobs can be very dynamic and transient, and it is difficult to understand this behavior without showing its evolution over time. While traditional monitoring tools typically show either snapshots of the current load balancing or aggregate trends over time, our new visualization technique shows the behavior of each of the jobs over time in the context of the cluster, and in either a real-time or post-mortem view. Its new algorithm runs in real-time mode and can make retroactive adjustments to produce smooth layouts. Moreover, our system allows users to drill down to see details about individual jobs. The visualization has been proven useful for administrators to see the overall occupancy, trends and job allocations in the cluster, and for users to spot errors or to monitor how many resources are given to their jobs.**

***Index Terms*— Visualizing jobs, stacked charts, time graphs, load balancing, MapReduce visualization, cluster management.**

I. Introduction

Scalable and flexible data processing has been a longstanding requirement for enterprises. In the past decade, Web companies such as Google, Yahoo, Amazon and IBM have pioneered new data processing platforms, such as MapReduce [1,2] and Watson [3], which scale to clusters of thousands of servers and have a capacity of many petabytes. In this paper we will describe a new visualization tool that is generic enough to work with several such platforms.

As a first example, the MapReduce framework, and its open source implementation, Apache Hadoop, is being adopted widely in industry and academia. A MapReduce cluster consists of a coordinator node (the master) and many worker nodes (slaves). Worker nodes are each configured with a number of Map and Reduce "slots" specifying how many tasks of a given type can run on the node concurrently. Cluster administrators configure the number of slots per node based on available resources (e.g. the number of cores and disks, the amount of memory) and resource requirements of typical tasks.

A MapReduce job consists of some number of independent Map tasks and some number of independent Reduce tasks. A Map task reads an amount of input data (called a split) from disk, processes it, and writes its output to local disk. Once a Map task is done, its output gets partitioned and sent to their respective Reduce tasks, in a process called the shuffle. Once a Reduce task receives its inputs from all the Map tasks, it can perform its own processing, producing a partition of a job's final output. Thus, Map tasks start first, and once some fraction of the job's Map tasks is done, the shuffle phase and the Reduce tasks can begin.

Since the cluster contains a finite number of slots, simultaneous MapReduce jobs from multiple users will compete for slots. Workload management is a key concern for cluster administrators, who want to satisfy their users while making effective use of resources. Our work on the FLEX scheduler [4] motivated the need for sophisticated visualization tools in order to see the effects of scheduling decisions on Hadoop workloads and cluster resources. As we will illustrate later, we observed that visualizing job behavior at a cluster level and at a detailed level was useful not only for administrators but also for MapReduce users and developers.

A second example of large-scale data processing is the Distributed UIMA Cluster Computing (DUCC) that drives the Watson DeepQA [5] infrastructure, used to develop Watson algorithms. (Watson won the Jeopardy television quiz show contest against world-class human opponents in 2011 [6].) Similar to the MapReduce environment, DUCC can run hundreds of jobs, submitted by different users. The programming model is somewhat different from MapReduce, but is similar in that its jobs exhibit a massive, embarrassingly parallel character. As opposed to slots, the granularity of shared resources is expressed in multiples of a "quantum" of RAM of a server – typically 15 GB. A DUCC job can be configured to run in Java Virtual Machines (JVMs), each using space that fits in one or multiple quanta. For example, a cluster node with 60 GB of RAM available can run two JVMs using 30 GB, or 4 JVMs using 15 GB and so on. JVMs launched by a given job will all be of the same size, but the cluster may have a heterogeneous mix of JVM sizes for different jobs. In this shared environment, a common job scheduler launches new jobs, may expand and contract the number of JVMs for each job, and terminates them. It is worth noticing that the expansion and contraction of DUCC jobs may come at some application level cost. Expanding a job not only involves launching new JVMs, but also loading new data for the applications that are to be run. Therefore, frequent switching of resources in the Watson DeepQA environment is typically avoided.

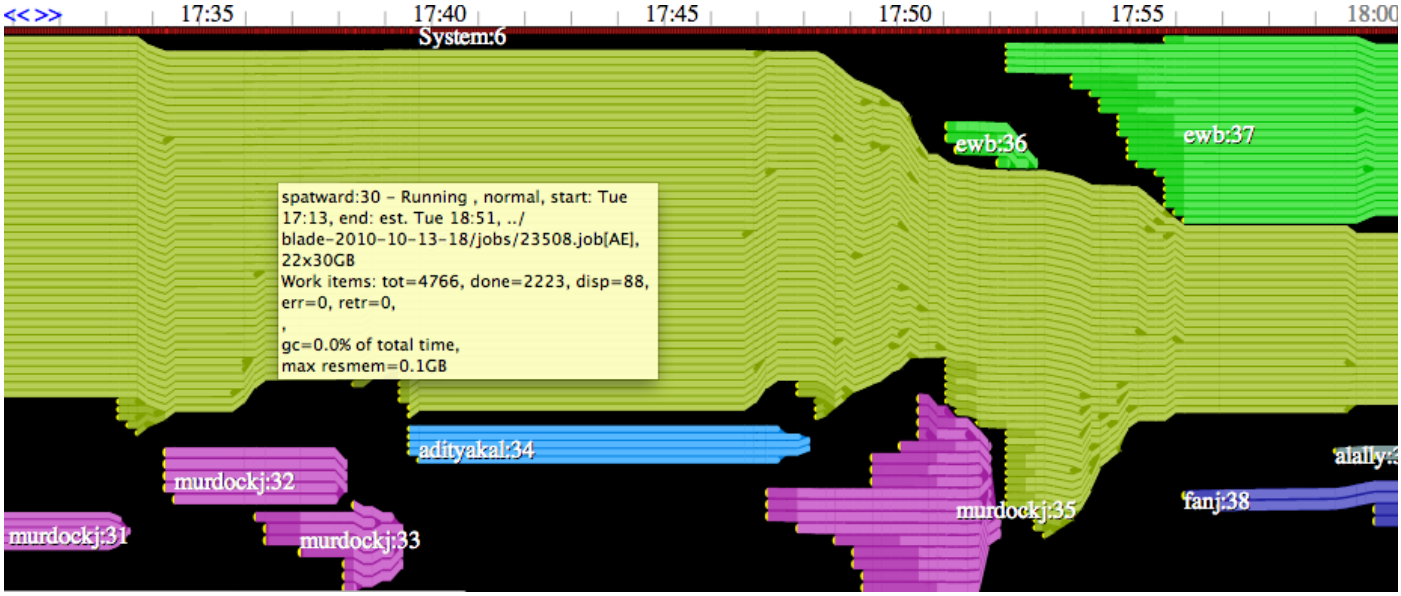

Figure 1. The cluster view shows Watson DeepQA jobs colored by user, scrolling from right to left; the height of a job at any time indicates the number of resources it is consuming then; the height of the view reflects the total capacity of the cluster; the thickness of the lines inside each job shows the size of JVMs.

In this paper we will present a new visualization tool that shows how jobs consume shared resources (for these examples, either MapReduce slots or DUCC JVMs) over time. Our initial motivation was to visualize this behavior in order to support load-balancing, correctness and performance analysis in large scale computing. We noticed that the visualization tool was sufficiently generic to be useful, with modest customization, for *both* the MapReduce platform and the DUCC cluster in Watson DeepQA, and presumably others.

The new visualization technique shows trends in multiple variables over time much more clearly than standard time-based graphs. Time is still depicted in the horizontal direction, but as the example in Figure 1 illustrates, we depict jobs as "floating" bundles with variable heights reflecting their resource consumption over time. The absolute vertical position of a job is merely used to enhance the layout.

The rest of the paper is organized as follows. In section II we briefly describe existing visualization tools that show resource usage. The requirements and the design of our new visualization are described in section III. Section IV describes how we applied this technology to these two domains. Section V is about interacting with the visualization, and we validate the proposed methods in section VI with use cases. Finally, we describe the architecture in section VII and our design methodology with user feedback in section VIII.

II. Related Work

Stacked graphs are typically used to show the contribution of multiple variables over time. Examples are Microsoft Excel's "Stacked Area Charts" [7] and Many Eyes' "Stack Graph for Categories" [8]. These charts are compact and can clearly show the trend of the aggregation over all variables. However, it may be difficult to discern the trend of individual variables, especially when the slopes are steep. While the layers in Stacked Graphs are arranged by downward gravity, the ThemeRiver [9]) and Streamgraphs [10] visualizations let the layers gravitate towards a center axis, usually a horizontal straight line. This visualization suffers from the same problem: the outer layers may fluctuate wildly, since their position depends directly on the layers near the center. As a result, observing the evolution of an individual layer can be tedious, especially near the outer layers or when slopes are steep. Yau offers an animated comparison on FlowingData [11] between Streamgraphs and Stacked Graphs.

Another category of visualization focuses on the affinity between different entities over time. Ogawa's work [12] illustrates participants of a software project over time as lines. Lines for new participants are added close to lines for existing participants, based on the degree of their mutual interaction. When many new events occur, the outer lines can exhibit significant fluctuations. Tanahashi and Ma [13] use a similar theme of storylines, but achieve a more fluent layout by rearranging and relaxing the lines, in addition to removing white space. Neither of these visualizations, however, convey any quantitative information attached to the participant entities.

Earlier tools help with monitoring key resources such as CPU load and network traffic (e.g., Ganglia [14]), as well as metrics that are specific to Hadoop's workload [15,16]. However, such tools do not offer a sufficiently high-level overview of cluster activity or reflect the behavior at the conceptual level of MapReduce (i.e. jobs, Map and Reduce tasks or workflows). The Ambrose [17] tool visualizes MapReduce workflows but does not provide detailed information about resource usage trends in clusters.

Visualization techniques in the domain of software evolution often draw software entities and their attributes over time. Lungu and Lanza use stacked charts in their work on Software Ecosystems [18] to show dependencies between software projects. Voinea and Telea [19] also propose time-based charts to track software evolution, but they keep related

entities horizontal and group related entities together in their 'cluster map'.

Bernadin et al. built Lumière [20], a visual tool to help understand scheduling out-of-core algorithms. They use a stacked graph visualization to track the performance of the algorithms.

III. Design of a Visualization Environment

A. Design Requirements

Since the behavior of the jobs large computing platforms can be very dynamic and we wanted to show this transient behavior, we chose to use time as an explicit dimension. Time '*now*' corresponds to the current time; the leftmost time coordinate depends on a configurable time span parameter. So the horizontal dimension in our visualization is essentially the same as in standard time-based graphs. It is in the vertical dimension that our tool is new. For example, techniques like Stacked Graphs or Streamgraphs would organize jobs as layers stacked on top of each other, with the height of a job at a given time reflecting its resource consumption. However, the variability of resource usage by jobs in a system usually makes it hard to spot trends of individual jobs this way. We therefore opted instead for a different vertical organization of the jobs. Our goal was to be able to follow the trend of each individual job easily, while still seeing the overall context. We still depict the resource usage of a job at a given time as the height of the job at this time coordinate. However, rather than packing the jobs together by gravity towards the bottom (like Stacked Graphs) or organized around a center axis (like Streamgraphs), we let jobs "float" in the vertical dimension. The absolute vertical position of a job does not carry any significance and is just used to produce relatively stable bundles while allowing for future expansion. This new degree of vertical freedom allows us to better fulfill some of the original design requirements, which were as follows.

1. Time should be laid out explicitly (horizontally) in order to show trends and evolution.
2. The height of a job at a given time coordinate must accurately reflect the number of resources it is using at that time. This number will vary as the job uses more or fewer resources.
3. Jobs should not overlap.
4. Jobs should vary as little and as stably as possible in the vertical dimension over time, in order to easily see their individual resource consumption trends. Some variation is inevitable as jobs gain or lose resources, and because of new jobs arriving, but unnecessary variation should be avoided. (We will refer to this goal somewhat colloquially as "minimizing the slope of the jobs".)
5. The visualization should clearly illustrate the overall aggregate resource utilization of the system.
6. The layout should cluster related jobs. For example, jobs submitted by the same user should be shown vertically close to each other at any given time coordinate and should not be separated by jobs from another user. This allows for an easy visual assessment of the aggregate resource consumption by this user. Similarly, in a platform supporting workflows, jobs in the same workflow should be drawn close together, and as a bundle if they execute in parallel.
7. Ingesting, calculating, laying out and rendering of the execution data should be performed in real-time, in order to give users and administrators fresh and actionable information.
8. Users should be able to request and receive more detailed information about a specific job on demand.
9. In addition to showing live information, the visualization environment should allow automatic archiving, so a user or administrator can analyze past information if desired.

B. The cluster view

Figure 1 shows an example of the new visualization applied to the Watson DeepQA DUCC domain. The horizontal dimension is indicated by the timeline at the top, and current time is shown on the right. The time span shown in the visualization (30 minutes in this example) is entirely adjustable. Shorter time spans give more detailed information about recent behavior, while longer time spans give a broader overview. The height of the view represents the total capacity of the system in terms of resources (in this case, quanta). Notice the absence of a vertical scale, since the absolute y-coordinates in this view are only calculated to produce a stable, more readable layout. Each job is shown as a connected bundle, colored according to its user. At any given time the height of a job bundle represents the total number of resources it is using then. Since the number of resources used by all jobs at a given time can never exceed the cluster capacity, it is always possible to arrange the jobs without overlapping each other. Empty (black) spaces between jobs at a given time indicate idle resources. This allows an administrator to easily observe the occupancy of the cluster.

C. Layout Algorithm

In order to realize the first four design requirements, we used a combination of three methods to determine the y-coordinates of the jobs. The view is organized in time slices determined by the timestamps of incoming live execution events, as shown in the hypothetical diagram of Figure 2.

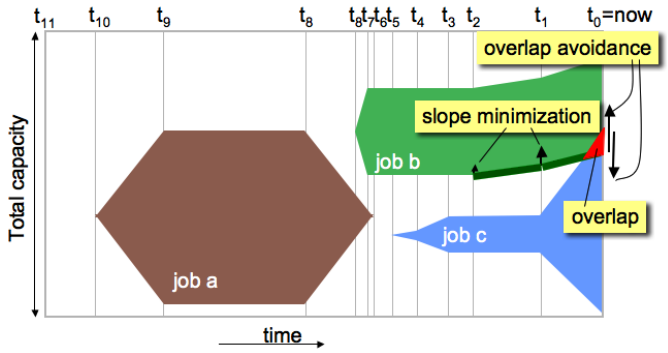

Figure 2. Two forces in the layout model move the green job: overlap avoidance and slope minimization

For a given time slice, we adjust the y-coordinates of each job present at that time to avoid overlaps with other jobs above or below and to optionally provide idle space (if possible) surrounding the job. In this example, a sudden increase in resources for the blue job causes an overlap between the green and the blue job at time '*now*' in the layout model, indicated by the red area in the figure. The "overlap avoidance" forces will result in coordinate adjustments to fix this, and this will become apparent in Figure 3.

But merely avoiding overlapping jobs may still leave us with uneven, less readable layouts. Therefore we apply a second adjustment scheme (at a lower priority than the overlap avoidance) to the vertical position of a job. For a given time slice, this adjustment takes into account the job's position in the time slices immediately to the left and right, if applicable. More specifically, we try to minimize the slopes of the boundaries of the job bundles, as illustrated by the dark green line in Figure 2. Again, we are trying to force the job bundle boundaries to be as flat as is possible even with the growth or shrinkage of resources of the job. We also want as little perturbation as possible caused by *other* job bundles. The idea can be thought of in terms of straightening the job shapes first from right to left, then from left to right, and iterating. In the process we remove irregularities as much as possible.

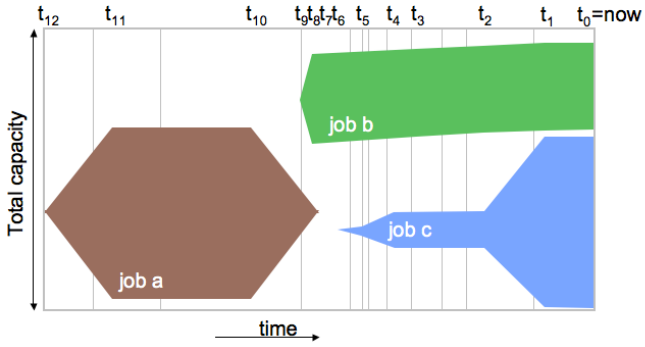

Figure 3. Same view as in Figure 2 after scrolling to the left, moving job b to avoid overlap and to reduce its slope.

Our layout algorithm iterates within each time slice as well as across time slices. Figure 3 shows the effect of a new event (shifting all time slices to the left); the overlap avoidance (moving job b up); and the slope minimization (job b is more horizontal). When feeding this visualization with live data, new events, and as a result, the largest changes, will appear on the right side of the view. We therefore apply the coordinate adjustments with stronger impact on the right side and decrease the impact as we move left. As in force-directed graph drawing algorithms, we repeat this two-dimensional iteration with decreasing force until the coordinate changes become small. This retroactive adjustment of vertical job positions continuously tries to improve the overall layout, including past time slices, as a response to new incoming events.

The third way to obtain smoother layouts is to position new jobs (at time '*now*') at a location where they have the most room for future expansion, which is typically at the middle of the largest vertical gap between jobs present at time '*now*'. We say typically because we may have to take into account the 6th design requirement about job clustering. We cannot insert a job from a user in between two jobs from a different user. This initial placement of jobs determines the vertical stacking order of jobs at all subsequent times. Note that finding this is an on-line problem, and there is no knowledge of the future, only the past. The corresponding off-line problem, finding a truly optimal set of job placements for any given time span in an omniscient environment, is actually NP-hard. And, of course, such knowledge is impossible. All told, this suggests a real-time alternative such as our current iterative greedy algorithm.

Of course, obtaining smooth layouts becomes more difficult as system utilization increases. For a system at 100% capacity, the jobs saturate the system and cannot be moved up or down to minimize the slopes.

D. Animating the cluster view

Animation may be used gratuitously, for effect. In our tool, we use animation sparingly, to enable transitions. Recall first that we wanted to observe the behavior of a cluster in real-time. Since time is depicted as an explicit dimension, we let the cluster view scroll to the left, in sync with the timestamps of the incoming events. These can come at irregular intervals of a few seconds and can be any of the following:

1. Creation of a new job
2. Termination of a job
3. Job expansion
4. Job contraction
5. A change of job attributes (e.g. errors)

The second way that we applied animation was to avoid sudden transitions as a result of new, incoming events. Given that we cannot predict future input, the layout calculated after each new input can at best be optimal for the current set of data. Rather than attempting an optimal layout quickly after a new input event, we dampened the impact for the slope minimization in the algorithm enough to slow down the movement of jobs over multiple time intervals. As a result, the jobs move up or down relatively more slowly, over multiple time slices. While the sideways scrolling of the view in a realistic setting is usually noticeable, the vertical change in job position is slow enough not to distract, but still fast enough to obtain near optimal layouts most of the time.

IV. Domain Adaptation

Using this basic layout algorithm, we can fulfill the design requirements listed in this section. But we can also tailor the new visualization with domain specific concepts. For example, MapReduce jobs have tasks of two different types (Map and Reduce) within the same job. A DUCC job, on the other hand, runs equally sized JVMs, but their memory size is configurable on a per job basis.

These domain customizations also entailed defining and implementing specific user interaction with the environment. For example, MapReduce users were interested in seeing task behavior for a single job, whereas Watson DeepQA developers asked for detailed performance reports to be used for regression testing.

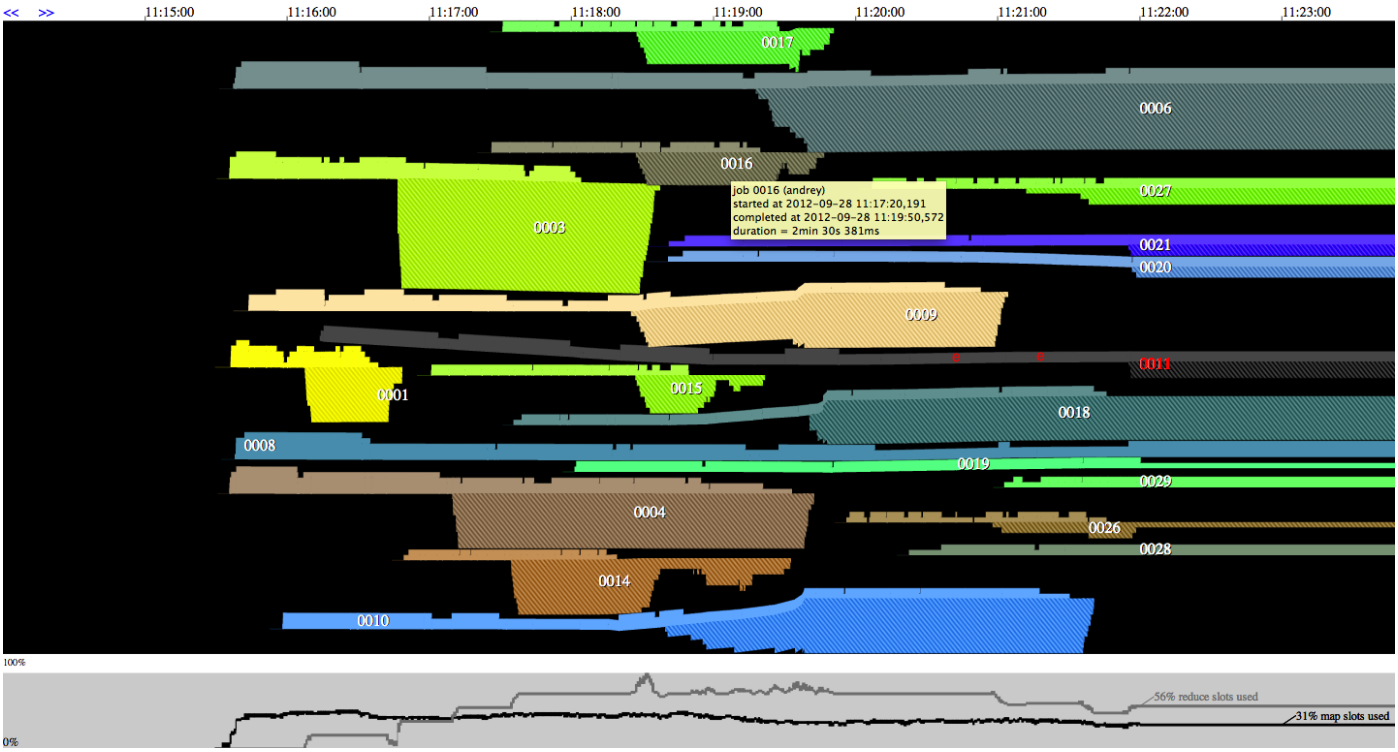

Figure 4. The cluster view of the MapReduce System scrolls to the left as time proceeds; jobs are colored by user, their height at any time indicates the number of slots in use; solid color areas indicate Map tasks, hatched patterns indicate Reduce tasks in a job. The line chart at the bottom shows the total Map slots and Reduce slots utilizations.

A. MapReduce adaptation

A MapReduce job consists of some number of Map tasks that process input data and some number of Reduce tasks that process Map outputs. Reduce tasks start only after some fraction of the job's Map tasks is complete. Thus, MapReduce developers want to see not only when and how their jobs were running, but also when and how many Map tasks versus Reduce tasks were active. Showing individual slots used by tasks in the overall cluster view would not scale well, since a cluster can be configured with many thousands of slots. We indicate instead the Map portion of a job in a solid color and the Reduce portion of a job in a hatched pattern of the same color, as shown in Figure 4 and in the zoomed-in detail in Figure 5. This color is automatically determined by a color-hashing scheme based on user name. When a job has both Map and Reduce tasks active at the same time, we draw the Map portion at the top.

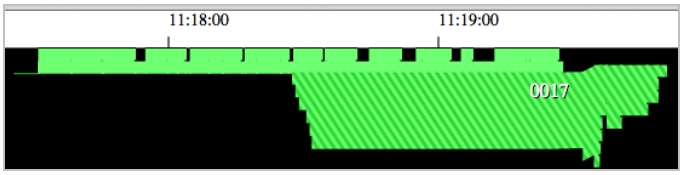

Figure 5. Zoomed-in detail of the cluster view in Figure 4 shows the Map activity of Job *17* in solid green, and the Reduce activity in the green hatched pattern.

If a job layout is not constrained by other jobs, we draw the portion that contains the active Map slots above and the portion with the active Reduce slots below an imaginary horizontal axis, as shown in Figures 4 and 5. Notice that the total number of slots used by a job at any time is still reflected by the height of the job at that time coordinate.

We adapted the slope minimization part in the layout algorithm so that it minimizes the slope of the axis that divides the Maps and Reduce slots. Of course, with other jobs competing for space in this view, this axis may not always be horizontal. One disadvantage of our "floating jobs" layout compared to stacked charts is that the aggregate resource consumption is reflected by the empty space in between jobs, which is therefore not depicted in as clean a manner as does Stacked Graphs. To address this shortcoming, we add a line chart at the bottom of the visualization, shown in the gray area below in Figure 4. It shares the time coordinate axis with the cluster view above and has two standard graph lines revealing the total utilization of Map slots and Reduce slots, aggregated over all jobs. These are normalized to be between 0% and 100% utilization.

Jobs with errors or exceptions are easy to spot by their red job labels, such as the dark gray job *0011* in Figure 4. The locations where the errors or exceptions occurred are marked with a small "e", around *11:20:40* and *11:21:15* for this job *0011*. Mousing over these red markers will reveal a tooltip with more detailed information about the exception.

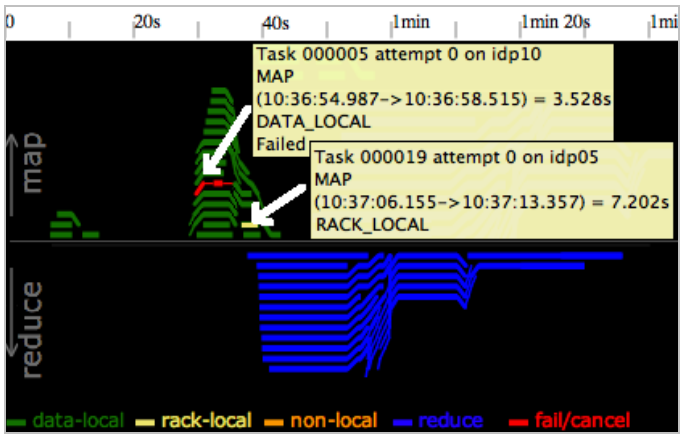

Figure 6. The single job view shows individual tasks with data-locality and errors. We overlaid tooltips to illustrate different data localities for tasks.

A user at all times can click on a job in this view, to bring up a "single job" view, as shown in Figure 6. We use the same graphical syntax here, time proceeding horizontally and the total height of this job bundle at any time reflecting the total number of slots it is using. This single job view automatically covers the time between job submission and either current time or job completion. Individual Map tasks are now shown as layered lines in the upper half, while individual Reduce tasks are shown as layered lines in the lower half. By layering Map task lines on top of each other over time, the shape of the upper half of this job indicates the total number of Map tasks over time. Similarly, Reduce task lines are layered so that the shape of the lower half of this job reflects the Reduce slot usage over time. As can be seen from Figure 6, we packed the task lines by gravity around a horizontal axis, similar to Stacked Graphs. We opted not to let the task lines float freely as we did for the job shapes in the previous section. Instead, we wanted the boundary of the job in this view to be similar to its shape in the cluster view. Moreover, there was no need to observe trends in the thickness of the task lines as it remains constant.

MapReduce application developers may also want to see the behavior of individual tasks in order to gain insight into possible failures, or performance indicators such as data locality, CPU usage and heap space. Since there is only one job shown in this view, there is no need to distinguish different jobs by color as we did in the cluster view. Therefore we can color individual task lines by a user selectable option, such as data locality or any of the other performance indicators. The single job view in Figure 6 actually illustrates both errors and data-locality. In the tooltip at the top we can observe that task *5* has failed, as is also evident by the red line. Moving the mouse to the right of this red area will reveal a "task cleanup". Finally, following the task line of task *5* further to the right will show that another attempt for this task completed successfully (not shown). We overlaid the same figure with a second tooltip capture, illustrating another useful metric. One of the task lines, colored in yellow, indicates that this task attempt was "rack-local". This means that the task processed a split of data that was not available on the machine that executed the task (*idp05* in this case), but *was* available in the same rack of the cluster.

B. DUCC adaptation

DUCC users prefer to see how many JVMs their jobs were able to obtain. As already mentioned, users configure a job with an upper limit of memory. DUCC will then allocate JVMs running with a multiple of a particular quantum (e.g. 15 GB of RAM) defined by the system to cover this need. As a result, a cluster may have a mix of 15 GB jobs, 30 GB jobs, 45 GB jobs, etc. In this domain, we visualize a job as a bundle of curved lines ("noodles"), each noodle representing a JVM. This is shown in Figure 1. Jobs are colored automatically based on the user ID, using a color-hashing scheme. This makes it easy for users to spot their own jobs (as well as their colleagues' jobs). The thickness of the noodles reflects the memory amount apportioned to each JVM: noodles for 60 GB JVMs are twice as thick as those for 30 GB JVMs, and so on. The total height of each job at any given time reflects the total amount of memory allocated for the JVMs spawned by this job. Starting a new JVM in DUCC is quick, but the Watson DeepQA applications tend to have large memory requirements and may load lots of data during the initialization. The launch of a JVM is marked by a small yellow edge at the beginning of the noodle. In order to indicate the initialization phase for each JVM, we use a slightly darker color (in the same hue) at the beginning of each noodle. Figure 1 illustrates how the short running purple *murdockj:35* job starting at *17:47* causes the long running green *spatward:30* job to give up almost half of its resources. DUCC typically starts a new job with a small number (two to four) of JVMs and only lets the job expand if it runs without errors for a short 'trial' period. This can be observed for the job starts that are visible in Figure 1. The figure also shows how *spatward:30* soon returns back to its original share of JVMs. But it also has to initialize the new JVMs, as is apparent from the darker shades in this job between *17:51* and *17:53*.

We customized the layout mechanism for DUCC in terms of the way jobs expand and contract. When expanding a job with one or more JVMs, we add new JVM noodles either at the lower or at the upper side of the job shape, depending on where more empty space is available. The aforementioned expansion of *spatward:30* between *17:51* and *17:53* shows the job expanding downward, because there was more empty space available below the job at that time. Expanding a job on the side with more room for expansion leads to noodles that appear more horizontal and thus to a more readable layout.

When a JVM in a job terminates, we shrink the job shape by moving the surrounding noodles up or down by half a noodle, as appropriate. This is shown in Figure 7. DUCC users somtimes are interested in knowing when work in a JVM terminated normally or whether the scheduler preempted the JVM in order to make room for other jobs. Instead of overloading the visualization with too many symbols and legends, we opted for a simple visual encoding with the line caps style. A rounded noodle end is used for a normal JVM termination, whereas a hollow end is used for preemption of the JVM, as illustrated in Figure 7. Similar to the MapReduce visualization, red job labels flag errors or exceptions in real-time.

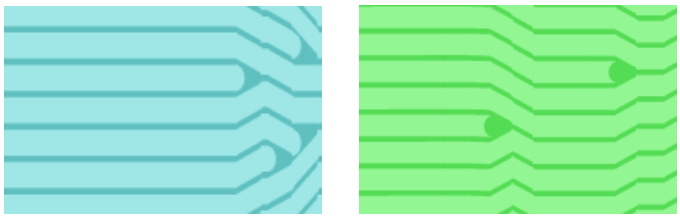

Figure 7. Left, rounded line caps denote normal JVM terminations; right, hollow line caps denote preemption.

The two most important goals for DUCC users who develop Watson DeepQA algorithms are improving the overall quality of the algorithms, and increasing or at least maintaining their performance. To help them with their performance analysis, we added real-time, one-click performance reports: when a user clicks on a job in the main cluster view, a new view will pop up (see Figure 8) to show the job information and its performance statistics per component. The view also provides information about failures, swapping, CPU utilization and garbage collection. Our visualization environment automatically archives these job reports, and users employed the archive intensively for regression testing.

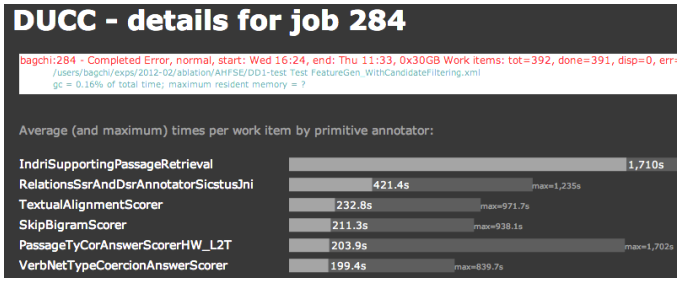

Figure 8. A DUCC job report shows the key performance indicators and component performance breakdown for one job.

The DUCC platform not only accommodates a heterogeneous mix of JVM sizes, it can also run on a set of nodes with different memory capacities. The node view shown in Figure 9 provides a real-time overview of the nodes and the jobs they are currently hosting. Each node is a square with an area proportional to its amount of RAM. Notice that the first two nodes at the top left are the largest in the cluster, each with 60 GB of RAM, the next 46 nodes (from left to right and wrapping on the page as though they were text) each have 45 GB of RAM, and the last 52 are 30 GB machines. Inside each node square we draw a treemap in which the tiles reflect the amount of RAM used by the jobs on this node. Each tile has the job ID and is colored according to the job user. As we can see in Figure 9, most of the node squares consist of a single tile treemap, indicating these nodes run one or more JVMs of the same job. A few nodes have two tiles. For example, the first node has a tile for job *239* and a somewhat smaller tile for job *188*. We can see from their color that both jobs on this node belong to the same user. Some nodes have a small black tile at the bottom, indicating RAM on this machine that was not allocated by the scheduler to any job. Our visualization environment detects the arrival or removal of nodes by tracking the 'heartbeat' of each node. Any change in the set of active nodes will be reflected with little delay in this view. At the same time, the new total of available resources will be reflected in the cluster view (Figure 1). We are currently in the process of adapting the node view in Figure 9 for the MapReduce environment.

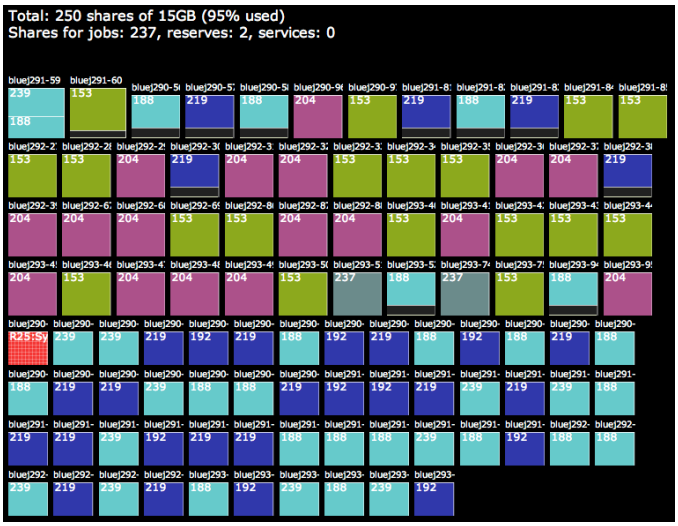

Figure 9. Nodes are shown as squares; their size is proportional to the amount of RAM; jobs are drawn inside each square with their job ID and colored by user.

V. Interaction

Users can interact with the visualization environment to get more detailed information. Rather than overloading users upfront with overly complicated views, we encapsulate detailed information as much as possible in tooltips in all views. Users can mouse over jobs, tasks, nodes or error markers to get accurate details about any of these entities.

Clicking on a job in the cluster view (Figure 1 or Figure 4) or in the node view (Figure 9) will bring up a single job view (Figure 6 or Figure 8). This provides an intuitive drill-down mechanism to see more specifics.

When running the tool in live mode, the cluster view shows the behavior of the cluster for a certain time span (user configurable) up until 'now'. However, at any time, navigation buttons (shown as the double blue arrows in the upper left corners of Figures 1 and 4) let the user browse back and forth in time, analogous to the functionality of a DVR. One of the shortcomings in our current implementation is that the time span shown in the cluster view is fixed at configuration time. We are currently experimenting with variable time scales, instead of the linear time scale that is in use now. The goal would be to see recent events in more detail than past events. The timescale would be linear on the right side of the view, as it is now, but going to the left, time would be more and more compressed, in an exponential way, to give the impression of a curving mural.

We plan to add more commands to the MapReduce visualization to let a user with the appropriate credentials cancel a job, change the priority of a job, or change its configured minimum and maximum number of slots.

VI. Use Cases

A. MapReduce use cases

One popular application of our tool is to ensure that MapReduce tasks are balanced correctly. Tasks that run significantly longer than others often point to a suboptimal job setup. In one case, a user observed a view like Figure 6, but with many long Map task lines in yellow ('rack-local') or in orange ('non-local'). This was evidence of Map tasks running slower because they were not data-local. The user verified and found that the input data was improperly balanced across the cluster, which was subsequently corrected by initiating an HDFS rebalance process.

In another case, a user saw a few blue Reduce task lines in the single job view that were significantly longer than the others. Clearly, these Reduce tasks were receiving much more data and, as a result, took much longer to finish. After the user modified the "partitioner" component of the job, the Map tasks distributed output to the Reduce tasks much more uniformly, fixing the long lines and speeding up the job. Without the intuitive graphical representation in the single job view, our user would have had to look up this information from the standard Hadoop user interface, which lists task runtime statistics in a numeric table. This table can be hard to analyze, as large jobs tend to have thousands of tasks.

Another important task for MapReduce developers is to understand the resource contention between tasks of their job by looking at trends in task durations. The standard textual Hadoop user interface is virtually useless to spot such trends. For example, a developer had a scenario where our tool was used to understand why a large job with 5000 Map tasks ran significantly longer than expected. From the single job view it became clear that the first roughly 2000 tasks all ran in a 5 to 15 seconds range, as expected. However, the rest of the tasks each took about 20 to 50 seconds each. What jumped out from the visualization was that this phase transition happened at exactly the same time when Reduce tasks of the job started. Clearly, this was a sign of heavy resource contention between Map and Reduce tasks. As it turned out, the contention was for disk access. The developer experimented with reducing the degree of parallelism and delaying the start of Reducers, which resulted in successfully removing the performance bottleneck.

We observed MapReduce developers using the real-time cluster view when they were testing their jobs in a shared environment. This allowed them to adjust their performance expectations according to the cluster utilization level. They reported that the ability to see historical trends with the cluster view was vastly superior to the standard Hadoop user interface that only displays how many slots are occupied at the present time.

MapReduce administrators tend to be the heaviest users of the real-time cluster view, since they need to see utilization levels per user. They either manually throttle down heavy users by changing their job priority, or change the cluster scheduler policy to achieve the desired effect. Administrators often utilize the post-mortem capabilities of our visualization environment as well, to analyze demand for, and effects of, configuration changes in both hardware and software.

B. DUCC use cases

Figure 10 shows an example of erroneous behavior in the DUCC cluster. The green job indicated instability in scheduling behavior. As can be observed from this figure, only two JVMs remain allocated for the green job. Other JVMs are allocated and then quickly preempted a number of times, without apparent reason. The extent of the darker shade of green during these sudden expansions exposes the amount of initialization work. The scientific results calculated by this green job were proven accurate (eventually) by the developer in question, and it would have been difficult to spot this problem in the platform without the visualization.

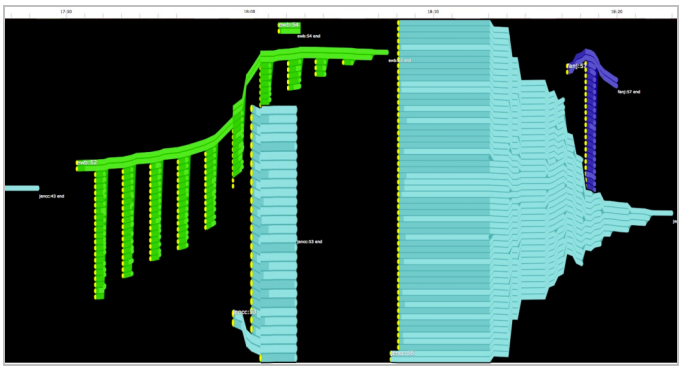

Figure 10. Strange resource allocation pattern for the green job.

After correcting the problem in the scheduler, the new behavior of the same green job looked much more condensed, and was faster. This is shown in Figure 11. This problem was detected in early testing of the DUCC system. The node view was helpful for fine-tuning the quantum size (now 15GB) in the cluster e.g. by observing and minimizing the black 'idle' tiles in the nodes (Figure 9). The cluster view proved useful in numerous cases before DUCC went in full 'production' mode.

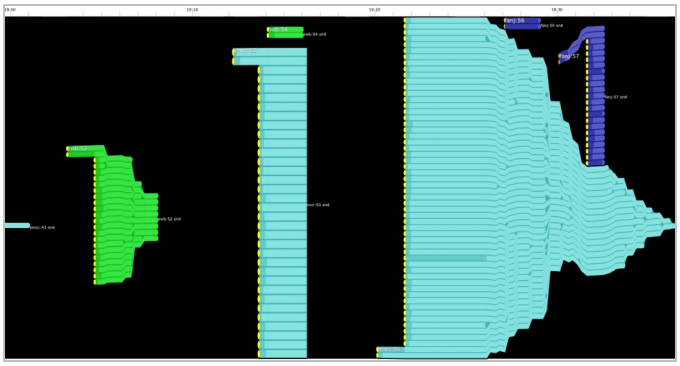

Figure 11. Same run as in Figure 10, after fixing a bug in the scheduler.

We also discovered an interesting social aspect to using this visualization. Greedy users, requesting a lot of resources from the system and thus slowing down other users' jobs, were exposed easily.

From a 20 people team outside our company we received this testimonial: "*During our long hours of work, we used the "race track" or Cluster view to monitor. The views were projected onto a white board in our war room so that every team member could monitor data ingestion jobs. [...] The simple yet rich data visualization helped the team to better understand our system performance. We were able to orchestrate the actions of our team through your visualization dashboard.*"

VII. Architecture and Implementation

A. Acquiring the events live or post-mortem

Our tool is capable of running in three modes: post-mortem, live and headless. In the MapReduce environment we extract the behavior of jobs from the JobTracker log, located on the MapReduce master node. For post-mortem analysis, the tool reads this log, replays the execution in fast-forward, and generates screenshots and single job views for further examination. In live mode, the tool tracks changes in the JobTracker log as it is being updated by a live MapReduce system. A simple polling mechanism with a sample frequency of 500 ms provides almost instantaneous updates. We are currently implementing additional access mechanisms through an HBase repository as well as a REST interface, which can report key performance indicators in addition to the basic job events. The headless mode runs without displaying the execution behavior immediately and only archives the cluster and job information for future usage.

The DUCC platform uses Apache ActiveMQ, an open source message broker implementing the Java Message Service for the communication between its components. Our tool simply subscribes to events from relevant components to get live updates. In addition, we have added the capability to read the messages from stored serialized message objects for post-mortem analysis.

B. Archiving the visualization

Even though the visualization environment is used mostly as a real-time monitoring tool for large computing clusters, it is also valuable as a troubleshooting and performance analysis tool for past runs. Rather than leaving the responsibility for taking snapshots with users, we incorporated archiving as an automatic feature at two levels. First, the tool stores a snapshot after every interval equal to the time span shown in the cluster view, to provide complete coverage. This makes it easy to troubleshoot – for example, to see unexpected behavior of the cluster during periods when it was unsupervised. The snapshots are stored as SVG (Scalar Vector Graphics) files and can be accessed through the tool's web server, or as stand-alone files with a browser. Even when using the SVG files in stand-alone mode, the tooltip information as well as the navigation buttons (back and forth) still function. Secondly, the tool automatically archives details for each job when it terminates and organizes this information in user directories. Developers use this feature for ongoing quality control. Since the cluster snapshots and the jobs are stored as simple, portable SVG files on a shared file system, they also serve as an efficient team communication vehicle.

C. Architecture

Figure 12 shows the architecture of the environment. To the left are the possible sources of information. The Web Server hosts the application logic (written in Java) for ingestion, modeling, the layout algorithm, generating the views, handling interaction and archiving, as well as dynamically serving the web pages to the browser. The combination of AJAX, JavaScript and SVG allows us to update the web pages in the background. As is shown in the figure, archived information can be accessed through the web server as well as directly from a browser.

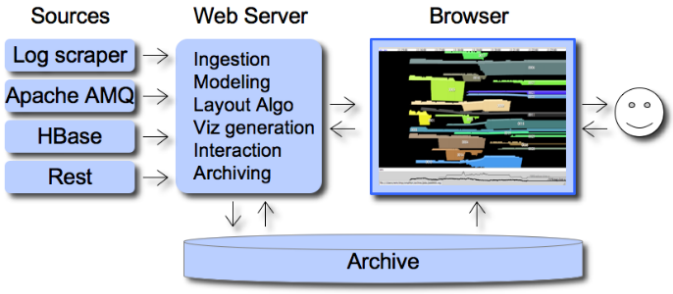

Figure 12. Architecture of the visualization environment

VIII. Iterative Design Driven by User Feedback

Watson DeepQA developers have used our visualization environment intensively since mid 2011, while MapReduce developers employed our technology since early 2012. From early on we had in-depth design reviews with a core of about 10 developers. As we rolled out new versions, we collected informal feedback from a larger group of about 50 users as they were using the visualization to help them in their daily tasks. We did this approximately weekly. The variety of users across different domains and roles allowed us to identify common needs, but at times also conflicting goals. We worked with individual users, often letting them try out prototypes with new features. This allowed us to make more progress than if we had followed a more formal approach with tightly controlled user experiments at a larger scale. It may be worth mentioning that new users sometimes asked about the meaning of the vertical position of jobs in the cluster view; most of them agreed with our explanation that this made it easier to observe trends over time.

Watson DeepQA as well as MapReduce developers liked the ability to "see" their computing platform in real-time with the cluster view. They found it easy to spot their jobs, since each developer had a dedicated color. Developers from both domains reported they used the cluster view for coordinating

and predicting work schedules, but they most frequently used the single job view to help them with performance tuning.

The manager of the Watson DeepQA DUCC team reported to us "*with confidence that they were primarily interested in:*

1. *The projected end time of a job* (visible in tooltips in the cluster view and in single job views)
2. *Details on any errors.*
3. *Performance information like computational breakdown, swapping, garbage collection and CPU utilization. "*

DUCC and MapReduce administrators relied on the cluster view to spot problems and to fine tune cluster parameters. They used the archiving utility to quickly diagnose problems that happened, for example, during nightly runs. The node view (Figure 9) was helpful for administrators when hot-swapping nodes in a live DUCC cluster.

ACKNOWLEDGMENT

Thanks to the Watson DeepQA team for their feedback and their help with the DUCC communication, in particular to Jim Challenger and Edward Epstein. Thanks also to all the MapReduce users for their feedback and suggestions, especially Eric Yang, Stephen Brodsky, Alicia Chin and Aaron Baughman.

CONCLUSION

In this paper we presented a new visualization technique that is effective for showing trends in individual variables coming from live data feeds. Our new layout mechanism is optimized to work in real-time, while simultaneously optimizing the layout in a retroactive manner.

We applied this technique in a visualization environment to observe the resource usage of jobs in two different domains: the Watson DeepQA DUCC cluster and the Hadoop MapReduce environment. System administrators, developers and end-users have used the new visualization environments intensively on clusters of more than 200 nodes since 2011.

We believe our proposed techniques would be applicable to more domains where it is useful to follow historic trends of resource usages.

We are currently working on extending the MapReduce cluster view to show not only past and present events, but to display hypothetical future behavior, by tapping into information from the scheduler. In addition, we also plan to extend our visual syntax to include MapReduce workflows.

REFERENCES

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters", OSDI, pp. 137–150, 2004.

[2] Apache hadoop. http://hadoop.apache.org

[3] D.A. Ferrucci, "Introduction to 'This is Watson'," In IBM Journal of Research and Development, Vol. 56, Issue 3.4, 2012

[4] J. Wolf, D. Rajan, K. Hildrum, R. Khandekar, V. Kumar, S. Parekh, K. Wu, and A. Balmin, "FLEX: A slot allocation scheduling optimizer for MapReduce workloads," in Proceedings of the Int. Conf. on Middleware, 2010.

[5] E. Epstein et al., "Making Watson fast," In IBM Journal of Research and Development, Vol. 56, Issue 3.4, 2012

[6] Watson. https://en.wikipedia.org/wiki/Watson_(computer)

[7] Microsoft Excel Charts. http://office.microsoft.com/en-us/excel

[8] F. Viégas, M. Wattenberg, F. van Ham, J. Kriss, and M. McKeon, "Many Eyes: A site for visualization at internet scale," In IEEE Transactions on Visualization and Computer Graphics, vol. 13, no. 6, pp. 1121-1128.

[9] S. Havre, E. Hetzler, P. Whitney and L. Nowell, "ThemeRiver: Visualizing Thematic Changes in Large Document Collections," IEEE Transactions on Visualization and Computer Graphics, Vol. 8, pp. 9-20, 2002.

[10] L. Byron and M. Wattenberg, "Stacked graphs - geometry & aesthetics," IEEE Transactions on Visualization and Computer Graphics, 14, pp. 1245–1252, November 2008.

[11] N. Yau, "Hurricane Sandy's impact on NYC 311 calls," Flowing Data, http://projects.flowingdata.com/tut/chart_transitions_demo

[12] M. Ogawa and K.-L. Ma., "Software evolution storylines," in Proceedings of the 5th International Symposium on Software Visualization, SOFTVIS 2010, pp. 35–42, 2010, ACM.

[13] Y. Tanahashi and K.-L. Ma, "Design considerations for optimizing storyline visualizations," IEEE Transactions on Visualization and Computer Graphics, Vol. 18, Dec. 2012

[14] Ganglia. http://www.ganglia.info

[15] X. Pan, J. Tan, S. Kavulya, G. Rajeev, and P. Narasimhan. Ganesha: "Black-box diagnosis of MapReduce systems," Proceedings of the 2nd Workshop on Hot Topics in Measurement and Modeling of Computer Systems, 2009.

[16] J. Tan, X. Pan, S. Kavulya, G. Rajeev, and P. Narasimhan, "Mochi: Visual log-analysis based tools for debugging Hadoop," In Proceedings of the 2009 Conf. on Hot topics in Cloud computing (HotCloud'09). USENIX, Berkeley, CA, USA

[17] Ambrose. https://github.com/twitter/ambrose

[18] M. Lungu, M. Lanza, G. Tudor and R. Robbes, "The small project observatory: Visualizing software ecosystems," Science of Computer Programming, Elsevier, Vol. 75, No. 4, pp. 264-275, 2010

[19] L. Voinea and A. Telea, "Multiscale and multivariate visualizations of software evolution, in Proceedings of the ACM Symposium on Software Visualization, SOFTVIS 2006, pp. 115-124, 2006.

[20] T. Bernardin, B. Budge and B. Hamann, "Stacked-widget visualization of scheduling-based algorithms," in Proceedings of the 4th International Symposium on Software Visualization, SOFTVIS 2008, pp. 165-174.