Sintr: Experimenting with liveness at scale

Luke Church       Mariana Mărăsoiu       Alan F. Blackwell

Computer Laboratory, University of Cambridge, United Kingdom

luke@church.name, {mariana.marasoiu, alan.blackwell}@cl.cam.ac.uk

Abstract
We describe Sintr, a work in progress system that supports live programming for interactive data-analysis at scale. We explain the existing practices of data analysis, motivate why liveness at scale is important, describe a prototype system and reflect on the lessons learnt so far.

1. Introduction

When we say ‘interactive data analysis’ what do we mean? Data analysis is the practice taking one or more data sources and answering questions from it. These questions are often transformed from being domain specific to being computational in nature. For example, a fairly high-level question such as "How are the marketing campaigns performing?" (Banasiewicz 2013) is transformed into analytical activities that may involve one or more of the following: collecting and curating data, computing summary statistics, running cluster analysis, and even creating prediction models (Kandel et al. 2012).

Characterising this further, the first studies of data analysts involved looking at intelligence analysis (Heuer 1999; Johnston 2005), whilst later work studied data analysts in other domains (Fink et al. 2009; Kandel et al. 2012). Kandel et al. (2012) offer a detailed account of data analysts in enterprises, reporting on 35 interviews with analysts from a range of domains, including healthcare, retail and finance. They define three categories of analysts based on their work practices: application users, scripters and hackers.

Application users perform most of their tasks in spreadsheets or other statistical applications, such as Excel, SPSS and SAS/JMP. The data they analyse is usually prepared for them by the IT staff, who maintain the data warehouses. The data is limited in size by the application they use, and as such they work with smaller datasets than the following two groups.

Scripters take advantage of domain-specific languages (e.g. R, Matlab) to perform more sophisticated data modelling than either application users or hackers. Kandel et al. (2012) suggest that these increased capabilities are enhanced by specialist packages and working environments. As with application users, data used by scripters has often been pre-processed, to be ready for analysis when they receive it.

Hackers have the most complex workflows as identified by the study, as well as being “the most proficient of the programmers of the three groups”, being able to manipulate data through multiple processing scripts, run either locally or in a distributed system. They use a variety of programming languages, including statistical languages (R, Matlab, etc.), scripting languages (Python, Perl, etc.) and data processing languages (SQL, Pig Latin for Apache Pig infrastructure). In fact, their work practices are similar if not the same to what is described by the term "data scientist" (Hammerbacher 2009), which was first used to describe roles at Facebook which encompassed a wide range of tasks which included creating data processing pipelines in Python, performing statistical modelling in R and communicating results of analyses to team members.

Looking at the tools used here, at one end, spreadsheets and other specialist analytics software can be considered live end-user programming tools. The analyst can manipulate data through functions and create static visualisations that update automatically when the underlying data changes. This is ‘interactive data analysis’. However, in Excel for example, the maximum size of the dataset is limited, as the application gets slower and stops responding when reaching a few hundred thousand data points.

At the other end of the spectrum, Pig and other MapReduce-based systems scale, but are not live. They’re akin to a software development workflow: the user writes a program, then sends it to be run on a cluster of machines, which write their results either to text files or to a database system, and only after the processing is finished does the user check the results. Some of these systems use static properties of the code to try and assist in the development process, but there is little inherent liveness.

This presents a great opportunity for an experiment: can we build a system that has some of the interactive liveness properties of Excel with the scale of MapReduce?
2. Data analysis as a domain for liveness experiments

As we have seen above, data analysts are already familiar with using live tools in the context of, for example, Excel. It may be that the lack of liveness is one of the issues that limits the uptake of the more scalable tools. This pent-up demand for liveness makes data analysis an easier context to carry out experiments in than a general purpose programming environment.

There are further reasons why we see data analysis as a suitable context for experiments with liveness; these reasons include theories of interaction behaviour, technical suitability and performance programming.

2.1 Theories of interaction

Much of the work that a data analyst does is in the management and preparation of data, followed by exploring a number of hypotheses. In Cognitive Dimensions (Green and Petre 1996) parlance this would be described as ‘exploratory design’; the simultaneous construction of data structures and the programs that operate over them. However, this exploration is being performed between three distinct notational layers: the layer containing the input data, the layer containing the program being manipulated that operates over the data and the output layer which will ultimately contain the results of the analysis.

Typically the exploratory design is being performed on output as the analyst works to understand something about the domain that the data arose from, however they can only do so by manipulating the program layer. This type of scenario where the layer of interaction is decoupled from the layer of investigation by a non-trivial process is often described as emergence (Church and Blackwell 2008, Bedau and Humphreys 2008).

The profile of desirable properties for doing exploratory design in emergent systems is a need for minimal Viscosity\textsubscript{CD} to make experiments as cheap as possible, strong support for Progressive Evaluation\textsubscript{CD} so the analyst can know if they’re going in the right direction, a need for an Abstraction\textsubscript{CD} tolerance to make it possible to express complex problems, and low Premature Commitment\textsubscript{CD} to let the analyst change their mind about the direction they’re going in.

Liveness acts to reduce the viscosity of the system in two ways. Firstly, setup steps do not need to be repeated between executions. In a conventional non-live system any setup work must be repeated between changes; this means that an edit that is conceptually one change to the user may require repetition of many operations to get the system back into the state the system was in. Fully live system eliminate this problem. Secondly, the liveness property tends to make the edit quicker to apply by decreasing the amount of computational work that needs to be done decreasing the cost of each change.

This decrease in computational work helps with building a system that supports progressive evaluation as it becomes possible to show the feedback after each change. This increase in the support for progressive evaluation is one of the most significant changes in the resultant user experience between a live and a non-live programming environment. In the data analysis case, it allows the hidden connection between the data layer, the program layer and the output layer to be explored much more effectively.

Abstraction is needed in to be able to reuse chunks of infrastructure code (e.g. how to parse a particular data format) and to modularise the analysis into components that can then be reused (e.g. statistical functions). We see this profile of decreased viscosity, improved progressive evaluation abstraction support and low premature commitment as being a good design target for experiments in live programming language design, implying that data analysis may be a good context to study such tools in.

2.2 Technical Suitability

There are a number of serious technical challenges in the design and implementation of live systems, for example those described by Karumuri (2013) and Church et al. (2016). In particular, general purpose live systems must manage the fact that the code can change the context of execution. In the worst case, this could be irreversible, for example if a program were to delete files. This means that when a change is made to the code it is not apparent how to re-execute the existing code given the altered context.

This problem is not common within data analysis. The code that the user is exploring, the processing algorithm, is often ‘pure’ in the sense that its primary value is the computational result rather than its side-effects. For example, the result of a regression analysis is usually just a coefficient and possibly a chart - there is no persistent modification of the external environment.

Further, the processing algorithm is often a small piece of code, both in size and execution time, that is executed repeatedly over different data. It is often reasonable to avoid the full complexity of a hot-patching VM, but to be able to provide a live experience to the user by simply executing the new code over its input.

These properties have a couple of benefits: Firstly it limits the need for deploying complex effect management systems such as Warth’s worlds (Warth et al. 2011), or full I/O virtualisation. Secondly, it means that liveness can be reasonably simulated on virtual machines that don’t have intrinsic support for liveness.

2.3 Performance programming

A third reason to explore support for liveness in the context of data analysis is that data analysis is often a performance, as we argue below. There is a rich tradition within the Live Coding community (Blackwell et al. 2014) of constructing programs while they execute, in front of an audience. Whilst
little data analysis is currently done at this level of performance, it is often done collaboratively.

The collaborative practices in the data analytics community are often currently in the form of meetings followed by individual work (Dix et al. 2003), however their spreadsheets are important collaborative artefacts that are shared in an ad-hoc manner (Hermans and Murphy-Hill 2015). As such there is a growing opportunity for more interactive collaboration potentially along the lines of pair programming.

From our own interviews with data analysts, discussed in (Marasoiu et al. 2016), we have observed that data analysts need to communicate with the people who request an analysis piece from them in order to clarify what needs to be achieved. We could imagine using live programming techniques to make this process much more interactive, rather than entrenching the boundary between the producer and consumer of the analysis.

Finally, data analysts frequently present their analysis through static reports and presentations, which contain summaries of findings, charts and suggestions for action (Kandel et al. 2012). This is a form of storytelling in the context of data analytics that is exemplified by Hans Rosling’s TED talks (Rosling 2006, 2009). Whilst meetings between data analysts and their clients don’t probably look like TED talks, they have a similar purpose: to explain and inform the audience about new findings from data, with the goal of learning, discussing and decision-making. In these scenarios, the analyst is a performer, a presenter, a storyteller.

However, one of the challenges reported by analysts in Kandel et al.’s study, and anecdotally confirmed by Finkelstein (2012), is that the reports were inflexible and did not allow verification of assumptions, or modification of parameters to observe how the results would change. This is caused by the fact that the form used for presentation (static reports and PowerPoint slides) is disconnected from the data that it summarises. There is an analogy that can be drawn between this and the tensions around the acceptability of pre-recording the music ahead of the performance (Parkinson and Bell 2015). This suggests that there are potential research avenues in transferring the practices of performing live coders to data analysis, e.g. Aaron’s performance diaries (Blackwell and Aaron 2015) with snippets of code for reuse of Lawson (2015)’s work showing practices for virtuoso tactics for editing graphics programs.

2.4 Existing systems

Live programming for scripter analysts has been previously considered in the Tempe system (DeLine et al. 2015), a notebook-style interface for analysing temporal data. The focus of Tempe is on level 3 liveness for stored data, the script being run shortly after the user has stopped typing, and level 4 liveness for streaming data, when the new script is run automatically on new incoming data. Similar to to the approach we will outline, Tempe provides incremental feedback, the interface being updated every second with new computational results. Tempe uses C# with the LINQ feature for creating SQL-like queries, which allows it to communicate easily with the Trill (Chandramouli et al. 2014) backend.

This context of an available audience, theoretical support, a tractable technical problem and existing expertise in understanding novel interactions is the backdrop for our prototype, Sintr.

3. Introducing Sintr

Sintr is an experiment into building a system that combines the liveness and interactivity of Excel with scalability in the same range as Hadoop. It is intended for use as an experimental platform for the exploratory phase during which analysts develop their questions. The primary interaction with Sintr is writing Dart code in a live environment, a screenshot of which is shown in Figure 1.

To structure the discussion let’s look at the process by which we might detect the presence of a hedgehog from an accelerometer attached to a food sensor. For more details as to the structuring of this example, please see Church (2016a). The task for the data analyst here is to attempt to determine "what are the patterns of hedgehog feeding behaviour?” from bulk accelerometer data.

3.1 What does an interaction with Sintr look like?

The user first uploads the raw data from the accelerometer to cloud storage. A sample of the data appears on the left hand side panel (A), this is to start the process of reasoning about concrete examples. The data should be in a sufficiently large number of pieces or shards, to be able to keep the entire cluster busy i.e. the number of inputs should be at least several times the number of machines they’re willing to spend in the cluster. They start the infrastructure, which commissions cluster worker nodes and a local worker process. The status of the infrastructure is shown in (F).

The user then starts writing the processing code (B), split into the familiar abstractions of map and reduce functions (Dean and Ghemawat 2004) written in Dart. There are two modes of liveness, representing levels L3 and L4 from Tanimoto’s framework (Tanimoto 1990). In L3 mode, the code is statically analyzed as it is inputted using the dart-services infrastructure (Church 2016b). Whenever the user presses run, the code begins executing and the results are streamed to the user interface (C, D, E). In L4 mode, or ‘auto’ the code begins executing as soon as the code has no syntax errors and user has not typed anything for a short period (typically around 1.2s, derived from experience running dartpad.dartlang.org).

Typically the first feedback they will experience is from the map phase. Whilst this is important in the early phase of analysis it is often not sufficient. For example, in this context the result of the map phase is to report if at a specific
instant a hedgehog is likely to be present. The result of ‘what fraction of all time is a hedgehog present’ is computed by the reducer. It is important, and more challenging, to give a live experience with these aggregated results. We achieve this repeatedly re-running the reducer over the results of the map phase whenever new data is available and updating the resulting aggregate (D). Whilst this performs repeated work, it provides the opportunity for live feedback over very complex questions.

This design uses a distributed compute environment to provide feedback to the analyst about the higher level question they are exploring at minimum latency. Let’s look next at how this environment is structured.

3.2 Operating Sintr

A Sintr deployment consists of a number of different components, as outlined in the diagram from Figure 2.

The analyst uses the command line interface to upload data, and to provision and manage the worker cluster. This worker cluster reads the source to execute from datastore, copies the data from cloud storage onto the worker node in the cluster, executes the code and writes the results back to cloud storage. These results are associated with the source code that is executed to generate them which allows previously computed results to be cached. The number of nodes in the worker cluster can be adjusted dynamically at runtime, with a commissioning latency of a few tens of seconds. This allows a highly scalable approach which can provide throughput that correlates with the number of worker nodes. We exploit this to decrease the latency for providing aggregated results. However it cannot give feedback at a lower latency than the round-trip time to data centre.

To improve the liveness whilst writing the map function we supplement this design with a local worker which is running essentially the same code as the worker cluster, but typically achieves a latency to the first result of 34.8ms ($\sigma = 1.9$, $n = 5000$), this is sufficient for an interactive streaming system.

We have operated Sintr for approximately 9 months at the time of writing for a variety of tasks, such as extracting features for machine learning models (Church et al. 2016).
carrying out experiments on programming languages, and understanding the behaviour of hedgehogs. Over the history of the project we have processed in the order of a petabyte of data using Sintr, with the largest single job being 26TB of input data using 500 worker nodes.

Whilst this data volume is relatively small by modern standards, it is sufficient experience for us to learn some properties of the system.

4. Reflecting on Sintr

As discussed above, we have run small scale deployments of Sintr up to around 500 worker nodes, and make daily use of it as a tool to facilitate research. This gives us some experience in operating Sintr in both interactive and batch mode. Here we discuss the preliminary observations:

**Computational resource should be scaled in response to user inaction (sic):** When the user is actively editing the program, their information needs can typically be satisfied by the single local worker node, as they are mainly reasoning about the detailed behaviour of the processing algorithm. When they pause, they often do so to observe how their changes extrapolate to the entire data, in which case the system should provide them with feedback about the statistical properties of the results. This strategy can also decrease the visual distraction caused by large and rapid changes to the displayed data during user interaction. We observe the conversational nature of this interaction, with the human and the computer taking turns in their respective activity (writing code and running code), as well as offering cues for increased efficiency (e.g. small amounts of information coming from the local worker). We also note that this strategy is in support of Progressive Evaluation\textsubscript{CD}, discussed in Section 2.1. In the live, distributed design of Sintr, there is more than one kind of evaluation (local vs. distributed), and there is more than one kind of “progression”, scaling from the behaviour of the data input layer to that of the analytic output layer.

**Structural code changes work best at L3, parametric changes work best at L4:** Using Tanimoto\textsuperscript{2016}'s level of liveness taxonomy, we have found the system to be most workable if we can separate between changes to the program, and changes to the parameters that configure the program. The former seems to work best if the program doesn't begin re-executing unless the user has explicitly requested this, through either clicking a ‘run’ button or using a keyboard shortcut. However, when adjusting a parameter, for example, the smoothing of a processing function, the user is often engaging in “what if” behaviour that doesn’t have a large number of distracting intermediary states, so performing an immediate, live update in response to the user’s change to the program without requiring an explicit interaction creates a more fluid experience during exploration.

\textsuperscript{1}The user icon is under the Creative Commons Attribution-Share Alike 3.0 Unported license Attribution: Font Awesome by Dave Gandy – http://fontawesome.io

\textsuperscript{2}2016/7/20
4.1 Patterns of User Experience and Sintr

Sintr is by no means a finished artefact, rather it is a tool for the authors’ practice (McCullough 1996) and a technology prototype. It is thus useful to consider Sintr in the sense of live coding as originally advocated in the TOPLAP manifesto – creating languages and interfaces whose primary purpose is to be used in ephemeral performance rather than for the construction of engineering artefacts (e.g. Collins et al. 2003; Blackwell 2014). Moreover, from the interface above, it is clear that much further work could be done on polish and user friendliness, however this would distract from the topic being explored specifically around interaction at scale. Consequently, a systematic analytical discussion is useful for critically reflecting on the usability properties than a more traditional empirical user-study style evaluation. We will use the Patterns of User Experience framework (Blackwell and Fincher 2010), which as Blackwell highlights (Blackwell 2015), is a good fit for describing programming as performance and is developed from the extensive Cognitive Dimensions of Notations (Green and Petre 1996) that we used earlier. For brevity we will not consider each of the patterns, but will highlight some areas that we think we have advanced and some which represent weaknesses in the system. To improve legibility in a small space we will expand the indices inline.

Let’s start with There are routes from a thing you know to something you don’t (SE3). This is a reasonable summary of the purpose of interactive data analysis! To facilitate this, the user interface can be read from top left (data input) to bottom right (results summary), but this structure is not a rigid choice and certainly be adjusted by trivially repositioning or resizing the elements for emphasis, supporting You can compare or contrast different parts (SE4) and Important parts draw your attention (VE3). The property You can read-off new information (TE2) is supported by the visualisations of the results of the reduce functions, offering highly condensed summaries of large amounts of information.

Beyond this, considerable technical innovation has been applied to meet some of the PUX criteria, such as the use of incremental deployments of compute infrastructure to create a live experience that supports You can change your mind easily (SE2) and correspondingly You can try out a partial product (PE3). This supports commitment correlating with time since something having changed, meaning that You can be non-committal (PE4).

The tool is by no means a panacea and the PUX framework is helpful in elucidating why. The use of the MapReduce paradigm ties the shape of the data to the shape of the distribution, which for some problems can result in poor support for The order of the tasks is natural (PE1). The computational infrastructure requires considerable machinery making You can extend the language (CE1) much more challenging that would be ideal. The information you need is visible (VE1) represents a complex challenge. Ideally when debugging you would have an execution trace of the failure available to show in an immediately live context, however, this would require considerable extensions to the framework.

The system’s primacy of text as an input form means that not all Actions are fluid, not awkward (IE2). Lessons could be learnt here from McDirmid (2013). There are visual design issues that affect the use in performances, such as The purpose of each part is clear (ME2) and You can see the relationship between parts (SE1).

Finally the system offers substantial benefit in reasoning in one direction; from data to results, but offers little support in inferring from results to data. Causal analysis still does not meet You don’t need to think too hard (TE1).

5. Future work

We have motivated and described an experiment in building a live system that operates at TB scale. Such systems are now not only possible, but from practical experience, they are useful. However much work remains to be done. There are open questions in the presentation of such systems especially around distributed debugging, decreasing the ordering and constraint problems of a map-reduce style computation, and providing support for using the system for performances and collaborative usages. We present Sintr as an initial experiment in these directions, showing the plausibility of the approach and the tractability of the problem space.

Acknowledgements

The authors would like to thank Gilad Bracha and Emma Söderberg for conversations about this work.

Mariana is a Vice-Chancellor’s Scholar and is supported by an EPSRC industrial CASE studentship co-sponsored by BT. She is also supported by a Qualcomm European Research Studentship in Technology.

References


