Stat! – An Interactive Analytics Environment for Big Data

Mike Barnett[1], Badrish Chandramouli[1], Robert DeLine[1], Steven Drucker[1], Danyel Fisher[1], Jonathan Goldstein[1], Patrick Morrison*[2], John Platt[1]

[1]Microsoft Research
Redmond, Washington, USA
{mbarnett, badrishc, rdeline, sdrucker, danyelf, jongold, jplatt}@microsoft.com

[2]North Carolina State University
Raleigh, North Carolina, USA
pjmorris@ncsu.edu

ABSTRACT

Exploratory analysis on big data requires us to rethink data management across the entire stack – from the underlying data processing techniques to the user experience. We demonstrate Stat! – a visualization and analytics environment that allows users to rapidly experiment with exploratory queries over big data. Data scientists can use Stat! to quickly refine to the correct query, while getting immediate feedback after processing a fraction of the data. Stat! can work with multiple processing engines in the backend; in this demo, we use Stat! with the Microsoft StreamInsight streaming engine. StreamInsight is used to generate incremental early results to queries and refine these results as more data is processed. Stat! allows data scientists to explore data, dynamically compose multiple queries to generate streams of partial results, and display partial results in both textual and visual form.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems

Keywords

Interactive; big data; analytics; visualization; tool.

1. INTRODUCTION

Big data analytics allows a small number of users to burn a large amount of money very fast. The problem is exacerbated by the exploratory nature of big data analytics where queries are iteratively refined, including the submission of many erroneous (e.g., bad query parameters) and off-target queries. In existing systems, queries must complete before such errors are diagnosed, often after several hours of expensive compute time are used. With the pay-as-you-go paradigm becoming common in the Cloud, there is an increasing need to allow *data scientists* (also referred to as data analysts or users) to get immediate feedback to their ad-hoc analytics queries. In the same setting, data scientists may also wish to track the provenance of their results and maintain context information as they compose multiple queries. We define *interactive analytics* as the generation of results with very short latencies (e.g., seconds). From a data management standpoint, interactive analytics takes two main forms today:

1) ***Full-Data Processing***: Data is stored or cached in (distributed) main memory, and uses efficient organizations such as columnar formats, in order to allow queries over the *entire data* to complete in a very short time. Examples of such systems include Dremel [6] and PowerDrill [7].
2) ***Progressive Processing***: An alternative paradigm that can better fit a low-cost iterative querying paradigm is progressive processing, where the system produces early results based on partially processed data, and progressively refines these results as more data is received; until all the data is read, at which point the final result is produced. Progressive processing allows users to get early results using significantly fewer resources, and potentially end (or reissue) computations early once sufficient accuracy – or an early indication of query incorrectness – is observed. Several systems fall under the umbrella of progressive analytics, including the CONTROL project [3], the DBO system [5], and Map-Reduce-Online [6].

Interactive analytics requires us to rethink how data analysts (the end users) explore and interact with data. We have designed and built *Stat!*[1] – a new workbench for interactive analytics that is built around the use of progressive computations for data processing in the backend. Using *Stat!*, data scientists get a rich and interactive analytics environment that can help them achieve several goals as part of their big data analytics experience:

1) They can explore large data sets (both visually and in tabular forms) as if they fit in main memory; they are shown approximate results that are continually refined based on the amount of time that has elapsed since a query.
2) The data scientist can dynamically compose and adjust progressive queries, and see the results of more complex data workflows as they evolve.
3) They can follow an iterative approach of rapidly building, refining, and testing ad hoc queries.
4) They get a homogeneous environment for loading and handling data and schemas, as well as computations over diverse data sources.

Stat! needs a progressive query processing engine to execute queries in the backend. Instead of building an engine from scratch, we use an *unmodified streaming engine* (Microsoft StreamInsight [9]) to produce incremental results. StreamInsight is backed by a temporal algebra, where events are associated with logical application time [8] and the streaming query is modeled as a relational query over changing relational tables. This makes it possible to use the real-time streaming engine to instead compose and execute progressive queries over offline data.

We describe the Stat! design in detail in Section 2. System and implementation details are covered in Section 3. Section 4 ends

*Work performed during internship at Microsoft Research.

SIGMOD'13, June 22–27, 2013, New York, New York, USA.

[1]stat *adverb*: without delay, immediately (source: Merriam-Webster).

with a detailed set of concrete scenarios that we will demonstrate with Stat! at the conference.

2. THE STAT! DESIGN

In large data systems, an analyst may wait hours for a script to run on a cluster, only to discover a mistake that makes the results unusable. Instead, the *Stat!* environment displays incremental results of long-running commands. In this section, we outline the overall high-level design of *Stat!*

2.1 Unified Scripting Environment

With *Stat!* we take advantage of LINQ, which embeds SQL-like data queries into C#. This allows a single notation to express both data access and aggregation that are typical of "big data" notations and the statistical and mathematical computations that are typical of "small data" notations. LINQ is, by its nature, a parallelizable query language [[4]; as a result, LINQ also offers the advantage of supporting several execution environments: LINQ-to-SQL, in which a subset of LINQ commands compiles into SQL bindings; PLINQ, which run queries in parallel on a single machine; DryadLINQ, which executes LINQ as distributed Map/Reduce jobs; and Microsoft StreamInsight, which executions queries in real time over data streams. The use of C#/LINQ also allows access to the full range of existing .NET libraries.

2.2 Web-Based User Interface

We designed *Stat!* as a web service, with a client based on HTML, Ajax, and Javascript. All script computations are performed on servers, and all data, including small samples, are stored on servers. This allows data storage and script execution to be easily re-hosted (for example, to Windows Azure), without the need for analysts to track the machines where data is stored and to perform the clerical work of copying or migrating data between machines.

2.3 Interactive Analytics and Visualization

The *Stat!* scripting environment provides the user with a *read-eval-print loop* (REPL). In a REPL, such as Python or Matlab, a user types in a single line, which is *read* by the system, *evaluated*, and then a result is *printed*. REPL systems contrast with compiled code environments, in which a user creates a full end-to-end script and then runs it all at once. Of course, even in REPL scripting languages, users can also create stored scripts or loaded libraries; however, the system allows the user to interact with the system.

In most REPL systems, the result of a command is swallowed silently, or printed as a specific side-effect. This leads frequently to a pattern of alternately writing a command, then a print statement, then a command again. *Stat!* is designed to provide visual results after every command.

The *Stat!* scripting environment provides a two-column REPL. In the left column, the user types commands (C# statements or expressions); in the right column, the client displays the command result. In the case where the result is a simple scalar, the client draws a print string, like in a traditional REPL (Figure 1, first script line). However, when the result is a more complex object, the client chooses a default visualization for the data; the user can also interactively switch the visualization type or change its parameters. The visualization column reduces the need for many of the most common scripting commands which are used solely to dump results or create charts.

2.4 Incremental and Cumulative Results

In Figure 1, when the user enters the second script command to read from a 100 GB data file, the resulting table appears after one second and continues to update itself about once per second. The table shows the most recent 10 rows read. Similarly, the histogram appears a second after the third command is entered and updates itself once per second, showing the cumulative results of the commands so far.

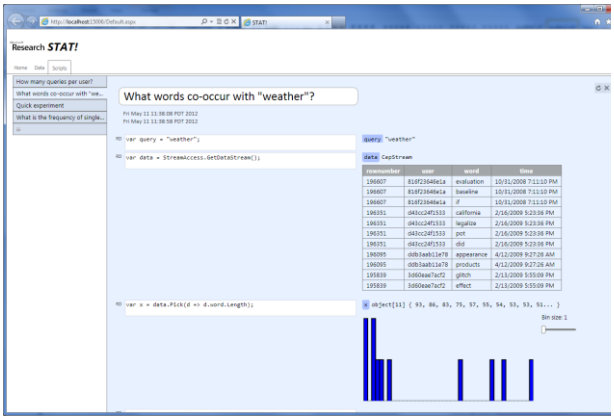

Figure 1. A screenshot of the *Stat!* environment, showing the first three commands of a script titled "What words co-occur with 'weather'?" The user enters C# or LINQ commands in the textbox (left) and sees visualizations of the service response (right).

To support the incremental update of results, the *Stat!* service automatically rewrites script commands. In particular, when a script accesses an enumeration (`IEnumerable<T>` in .NET), the script engine wraps call in a Microsoft StreamInsight stream, which allows the result to be processed in real time. Every time unit, the service reports a summary of the result back to the client, and reports overall progress. For data sources whose size is known, this can be reported as the percent completed; for other sources, we report the number of items processed. Our use of Microsoft StreamInsight is described in more detail in Section 3.

Since the results update incrementally, the corresponding visualizations need to update incrementally as well. Each visualization may use different strategies to help users keep context when viewing the visualization. Updates to graphs include animation; we are exploring ways to show changes to tables by using color and opacity to indicate volatile or stable rows. When possible in aggregate visualizations, we compute confidence bounds and display these on the visualizations based on dataset statistics.

2.5 Notebook Model

In a typical REPL environment, a session is by default ephemeral, unless the user takes explicit action to save her script to a file or to dump her whole session to a heap file. With *Stat!* we reverse this design choice. We use a notebook model in which every scripting action is implicitly persisted as it happens. As shown in Figure 1, the Scripts tab shows a list of existing scripts (left), with a new button at the end, and the contents of the currently selected script (right). Each script is listed under its title, which could be, for example, the question the script is intended to answer. We organize scripts as a flat list sorted in creation order. All scripting actions—creating and deleting scripts, adding, deleting or editing script contents—are immediately stored in a central database. When a user opens the *Stat!* web site, she sees all users' scripts and script contents up to the last recorded change.

The *Stat!* service also stores summaries of the results of all scripting commands, as they are computed. This means that the results of a script can outlast the REPL session itself. This

The user enters two commands:

then replaces the second command:

Figure 2. A notebook interface to a scripting session has the potential for confusion when commands are edited in place or deleted. The user can refresh the script to give the expected value.

longevity is handy for preserving the interactive session and answering questions about past analyses, but makes the user interface more subtle. First, the web client uses a different background to distinguish scripts where there is an ongoing REPL session ("live" scripts) from those whose REPL session previously ended ("dead" scripts). There is a refresh button to revive dead scripts. REPL sessions time out after 20 minutes of inactivity.

The ability to overwrite and delete commands, combined with the persistence of results, introduces a design subtlety, illustrated in Figure 2. Here the user entered two commands, then overwrote the second command. The resulting pair of commands would baffle anyone who had not seen the previous increment statement, because `y` seemingly has the wrong value. The issue is that replacing or deleting a command does not undo its effects on the REPL session. (However, the user can hit the refresh button to re-run the script and give `y` the expected value of 3.) In this design, we face a trade-off between presenting inconsistent results and restarting potentially time- and resource-consuming commands. Our current implementation chooses the former; in the future, we will maintain the data dependencies needed for consistency, and provide an appropriate user experience.

2.6 Collaborative Scripting

As a consequence of storing scripts in a central database and hosting REPL sessions on a central server, multiple people can work concurrently on the same scripting session from different web browsers. The web service asynchronously contacts each client about changes to the database to keep each user's view up to date. We tag each scripting action with the user id of the person who took the action. For example, in Figure 1, to the left of each command are the initials of the person who last edited that command (RD). This allows the user to understand what her team mates are up to.

3. SYSTEM DETAILS AND DISCUSSION

The *Stat!* prototype uses a conventional client-server implementation; the back-end is implemented in C# using WCF for the REST interface. The service is hosted on Microsoft IIS and uses Microsoft SQL Server for database needs. In this section, we describe the aspects of the implementation that are less familiar.

3.1 The Stat! Frontend

3.1.1 Dynamic C# and LINQ Interpretation

Our prototype uses Microsoft Roslyn[2] to implement the REPL. Roslyn is a language service API, which includes interpreters for C# and Visual Basic. Roslyn's APIs include a Session object to represent a REPL session, a method to compile a string containing C# code to a Submission object, and a method to evaluate a Submission object in a Session to produce a result object. The web service returns this result object as a JSON string, consisting of one or more variable declarations, or an expression result. For each variable declaration in the result object, the JSON string provides (1) the variable's declared name; (2) a print string for the variable's value, which is either the value itself (for simple scalars) or the value's type (for more complex objects); and (3) a description of the data to be visualized.

Stat! supports several forms of data description and visualization, with more being added regularly. *Stat!* chooses defaults based on the dataset, or allows users to interactively select appropriate representations. These can include *table samples* (e.g. the variable `data` in Figure 1); *simple vectors*, which provides a list of numbers (the top part of the variable `x` in Figure 1); and various visualizations. These include *frequency histograms* (Figure 3); *line charts*; and *2D histograms* for examining correlated arrays.

3.1.2 Interactive Query Composition

Cascading interactively implies that a user can enter line 1 (`x = readfile()`) and see partial results instantly, but later enter line 2 (`y = x group by …`) after a long delay. If an earlier query Q's result is a large memory structure (e.g., a histogram of user counters) then attaching a new query to Q can take a long time since it would need to consume these results prior to accepting new data. Further, the system may need to keep state in memory just to support the possibility of someone else attaching to it later. We manage these tradeoffs by allowing the possibility of discarding such state and instead re-executing an earlier step in the pipeline if necessary.

3.2 The Stat! Backend

3.2.1 Processing SQL Queries Incrementally

While relational engines are well-suited for set-oriented offline analysis, they are unable to produce incremental results as data is processed. For example, many database operations such as sort and merge-join employ algorithms that are fundamentally non-incremental and require multiple passes over the entire data. Streaming engines, on the other hand, enable applications to issue long-running *continuous queries* (*CQs*) over real-time data that arrives as a stream or sequence of *events*. For example, in case of Microsoft StreamInsight, users write queries using LINQ, with extensions for time-oriented computations.

Streaming engines work entirely in main memory and produce incremental results to CQs. CQs usually operate over *bounded windows* of data (e.g., a CQ may compute a 5 minute running average). Internal state is cleaned up when it is no longer needed, e.g., it cannot contribute to new results. All operators in relational algebra, including selects, projects, and joins, have their equivalent streaming counterparts.

We use Microsoft StreamInsight to generate interactive SQL results for Stat!. This is done by reinterpreting the notion of application time in StreamInsight to denote query progress. Briefly, each tuple is given a coarse-grained timestamp corresponding to the order in which we would like to process tuples. The window size is set to ∞ to denote that tuples never expire. Issues such as scheduling, synchronization across multi-input operators, memory management and state cleanup, and incremental maintenance of internal state are handled by the streaming engine, allowing us to focus on core issues such as query semantics, interactivity, and dynamic composition.

[2] See http://msdn.microsoft.com/en-us/vstudio/roslyn.aspx.

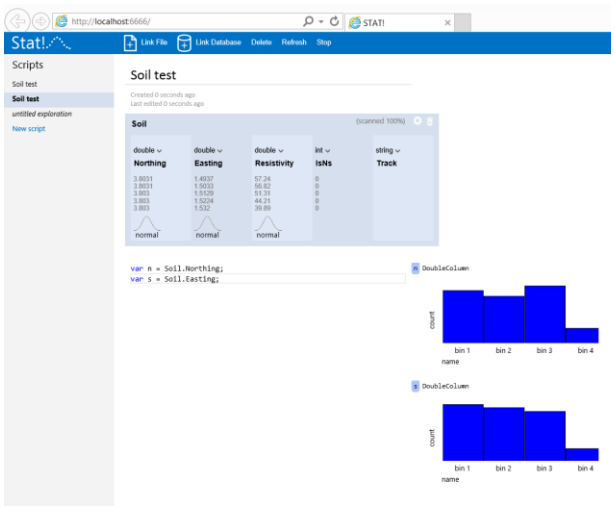

Figure 3. A screenshot of *Stat!* showing the incremental visualization of histograms computed in the backend, as part of the interactive querying environment.

3.2.2 Memory Management

A consequence of using an in-memory streaming engine to process offline data without enforcing bounded windows is that we can no longer clean up internal state in many cases.[3] As a result, memory usage can monotonically increase over the lifetime of the query. In some cases, one can leverage sort orders inherent in the data to avoid this memory blowup (e.g., when the data is ordered by join key). Fortunately, we do not expect interactive queries to run to completion and thus find that this is not an issue in practice. We can alleviate the memory problem by using a *distributed DSMS*, or by using scaled-out analytics platforms such as TiMR [8] with MapReduce Online [6].

4. DEMO WALKTHROUGH

4.1 Top-k Search Correlation

We start the demo with a canned query called "*top-k search correlation*" that operates over a sample of a 10TB search log dataset. This query was created by users working on a feature discovery task for a search engine. While the query is prepared in advance, it is run as a series of interactive LINQ expressions in the Stat! environment.

The user provides a keyword (e.g., "music") as input parameter, and the sequence of queries returns the top-k words most closely correlated with the provided term. This is a two stage process. The first stage uses the search data set as input and partitions by user. For each user, we compute a histogram that reports, for each word, the number of searches with and without the input term, and the total number of searches. The second stage job groups by word, and aggregates the histograms from the first stage, computes a per-word *goodness metric*, and applies a top-k operation to report the *k* highly correlated words to the input term.

The query is expressed in LINQ and executed incrementally by Microsoft StreamInsight in the backend. In Figure 4, we show how this query converges (in terms of top-k precision) as we process a large dataset using StreamInsight; we see that we quickly get a very accurate answer without having to process the entire dataset.

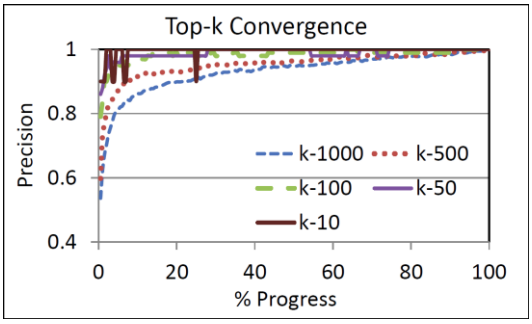

Figure 4. Convergence speed for top-k correlation query.

4.2 Twitter Data Exploration and Fusion

The second demo is based on a large sample of Twitter data. This query was created with researchers exploring how users interact with Twitter. During the course of the demo, we will ingest one file containing sentiment keywords, and then start a continuous stream from a second file containing tweets. We will dynamically create a histogram of distributions of sentiment on the tweet stream, and will show several visualizations based on this dataset.

As in the top-K search, the demo will be written in LINQ and executed incrementally on the back-end. Each command can be entered at the prompt; we will also use this demo to show the shared notebook and stored query facility.

4.3 Visitor Interactivity

Both of these demos are fully interactive. Visitors to the demo can broadly update or modify the demonstrations. At simplest, they can change parameters to the top-k search correlation query and visualize these results interactively. They can also tweak the query—altering its functions or its calls—to learn about both the iterative exploration process and the debugging experience. Similarly, users will be able to alter the Twitter experiment, including choosing new visualizations, adding or removing keywords, and otherwise tweaking the query. We will bring several other data sources so that users can issue and visualize their own interactive queries.

5. REFERENCES

[1] B. Babcock et al. Models and Issues in Data Stream Systems. In *PODS*, 2002.

[2] J. Hellerstein et al. Interactive Data Analysis: The Control Project. *IEEE Computer*, August, 1999.

[3] D. Fisher, I. Popov, S. Drucker, and m c schraefel. "Trust Me, I'm Partially Right: Incremental Visualization Lets Analysts Explore Large Datasets Faster". In *CHI,* 2012.

[4] E. Meijer. 2011. "The world according to LINQ". Commun. ACM 54, 10 (October 2011), 45-51.

[5] S. Melnik et al., "Dremel: Interactive Analysis of Web-Scale Datasets". In *VLDB*, 2010.

[6] C. Jermaine et al. Scalable approximate query processing with the DBO engine. In *SIGMOD,* 2007.

[7] T. Condie et al. MapReduce Online. In *NSDI*, 2010.

[8] A. Hall et al. Processing a trillion cells per mouse click. In *VLDB*, 2012.

[9] B. Chandramouli et al. Temporal analytics on big data for Web advertising. In *ICDE*, 2012.

[10] Microsoft StreamInsight. `http://aka.ms/stream`.

[3] To see why, consider the example of a join, where the last data item from the left-side input needs to join with the first item on the right.