Interacting with large distributed datasets using *Sketch*

Mihai Budiu *  
Barefoot Networks

Rebecca Isaacs*  
University of Edinburgh

Derek Murray*  
University of Edinburgh

Gordon Plotkin*  
University of Edinburgh

Paul Barham*  
University of Wisconsin, Madison

Samer Al-Kiswany*  
University of Wisconsin, Madison

Yazan Boshmaf*  
University of British Columbia

Qingzhou Luo*  
University of Illinois

Alexandr Andoni*  
Simons Institute, UC Berkeley

Abstract

We present *Sketch*, a distributed software infrastructure for building interactive tools for exploring large datasets, distributed across multiple machines. We have built three sophisticated applications using this framework: a billion-row spreadsheet, a distributed log browser, and a distributed-systems performance debugging tool. *Sketch* applications allow interactive and responsive exploration of complex distributed datasets, scaling gracefully to large system sizes.

The conflicting constraints of large-scale data and small timescales required by human interaction are difficult to satisfy simultaneously. *Sketch* exploits a sweet spot in this trade-off by exploiting the observation that the precision of a data view is limited by the resolution of the user’s screen. The system pushes data reduction operations to the data sources. The core *Sketch* abstraction provides a narrow programming interface; *Sketch* clients construct a distributed application by stacking modular components with identical interfaces, each providing a useful feature: network transparency, concurrency, fault-tolerance, straggler avoidance, round-trip reduction, distributed aggregation.

1. Introduction

The goal of this work is to enable the construction of applications for interactive explorations and visualizations of large datasets, ones which do not fit in the memory of a single machine. Visualizing large datasets presents several difficulties: (1) The renderings must adhere to the limits of screen resolution and human perception, which is a challenge when the number of data points vastly exceeds the number of pixels on the screen. (2) Renderings of the data must be computed at interactive speeds (on the order of seconds). (3) The graphical user interface must enable interaction with the data through the displayed renderings (e.g., zooming, scrolling, filtering, searching, and data transformations); these interactions lead to new data views which must also be computed at interactive speeds.

We introduce *Sketch*: a distributed software infrastructure which can be used as the backbone application for interactive exploration of large datasets. *Sketch* uses the collective resources of a computer cluster to manipulate datasets too large for a single machine. We describe in Section 4 three such complex distributed applications:

1. a spreadsheet for billion-row data sets,
2. a distributed log browser,
3. ViewCluster, a distributed systems performance analyzer.

*Sketch’s* design is based on two fundamental principles:

- Visualizations always display bounded views of the input data. In particular, algorithms use the screen resolution limitations to drive computation of bounded views, employing renderings based on aggregates or samples (binned plots, histograms, heatmaps) [27]. This approach is in contrast to almost all analytics engines, which compute aggregates, data cubes or data summaries independently of the screen size.

- Computations are performed using streaming algorithms. Streaming algorithms are a very active branch of research in big data analysis [13, 34]; their main feature is the use memory sublinear in the size of the input data. In this paper we use the term “sketching algorithms” for a particular class

---

* Work performed at Microsoft Research

---

1 The size of a view is bounded by the screen size. Asymptotically the view size must be at least logarithmic in the size of the input data; without such an assumption, a view would be unable to display even the size of the input.
of randomized streaming algorithms which: (a) can perform multiple passes over the data (subsequent passes may depend on results computed by earlier passes), (b) use a sublinear amount of memory (ideally logarithmic), and (c) produce a result insensitive to the input order.

The Sketch runtime combines these two principles: it is a generic runtime for writing sketching algorithms on distributed datasets. In our applications, all data renderings are computed exclusively using sketching algorithms. In general, streaming algorithms compute approximations of the desired results, usually trading off memory or time against precision. Sketch applications derive the target precisions directly from the display resolution, and in consequence, they compute approximate results which are indistinguishable from the exact results. For example, when drawing a histogram the sampling precision is chosen to make error bars smaller than one pixel in size.

The name Sketch reflects a dual nature: (1) a runtime for running sketching algorithms, and (2) a runtime for implementing visualizations with bounded precision data renderings ("sketches" as in "rough drawings").

The central abstraction of the Sketch architecture is a distributed object representing a Partitioned Data Set (PDS). A PDS represents a multi-set of values of an arbitrary type (e.g., a set of rows of a distributed table); each partition of PDS contains one element and may be located in separate address space, possibly on a different machine. A PDS provides a narrow API consisting of exactly three atomic methods (described in detail in Section 3); two of the methods create new PDSs, and the third one computes an arbitrary user-defined sketch.

This software architecture has the following properties:

- It hides the complexity of parallel and distributed programming from the programmer. Programmers write single-threaded code that executes in the client address space and accesses multiple PDSs as local objects.
- PDSs run efficiently on a shared-nothing computer system by exploiting parallelism and bandwidth across multiple cores, machines, and racks. PDS performance automatically scales-out with more computational resources. These are natural features of sketching algorithms.
- Because sketches are small by definition, PDS can provide interactive response even on low-bandwidth networks.

Several software systems have been previously built based on similar principles (see Section 6). While Sketch could be further improved using published techniques, we believe our work makes the following contributions:

- We give a mathematical formalism (based on commutative monoids and linear functions between them) that drives our design (see Section 2).
- We extend prior techniques for computing renderings of big data sets with precision guided by the screen resolution and visual perception limits. Resolution limits are used to speed up computation and to reduce network communication without sacrificing visible result accuracy.
- We describe the PDS software abstraction, which allows the manipulation of large distributed data sets through a simple API. Unlike most other big data frameworks, the PDS abstraction is provided by a stateful service; the data manipulated can also be much richer than a simple relational or nested-relational model. The state is distributed, immutable, and all state transformations are functional and transactional; state is garbage-collected. A PDS is a distributed higher-order generic aggregation network, allowing the distributed execution of arbitrary user-defined aggregation functions (see Section 3).
- We describe a modular implementation of the PDS API. The modular implementation relies on multiple datatypes all implementing the exact same interface, and each resolving a different problem raised by the construction of a distributed system. For example, there are separate datatypes to contain data, encapsulate parallelism (at the rack, cluster, or core level), implement inter-process communication, provide fault-tolerance, or bound the response time. A distributed system is built by mixing and matching the desired datatypes in complex hierarchies.
- We provide evidence through three applications that these abstractions are powerful enough building blocks for constructing significant visualization tools (see Section 4).
- We validate the scalability of PDSs on a computer cluster with 155 machines (1240 cores) (see Section 5).

2. Linear Transformations

The PDS design is based on a theory of linear transformations. In this section we provide the intuitions behind this theory, which is described formally in separate paper [7]. This theory has informed the design and implementation of Sketch. In particular, the formalism enables us to give a clear semantics to Sketch computations. Prior work [48] has shown that it is easy to abuse parallel frameworks such as Map-Reduce to write incorrect programs; we also argue in Section 6 that other aggregation network-based systems widely deployed have similar subtle shortcomings. A formally sound design allows us to avoid such mistakes.

Collections as monoids. The core mathematical structure we rely on is a commutative monoid \( M \): a set with a commutative and associative operation \( +_M : M \times M \rightarrow M \) and a zero \( 0_M \) which is the identity element for \( +_M \). A typical example of such a monoid is the set of real numbers.

A collection can often be modeled as an unordered multiset of values (a value can appear multiple times in a multiset). If \( X \) is the type of values, we write \( C(X) \) for the type of collections with values of type \( X \). \( C(X) \) is a monoid when using multiset union; the empty multiset is the zero.

Given a monoid \( M \), we denote with \( M[K] \) the type of key-value dictionaries with keys of type \( K \) and values of
type $M$. (If a key is “missing” in the dictionary we define it to map to 0.) We write the dictionary mapping $k_0$ to $m_0$ and $k_1$ to $m_1$ as \{$(k_0 \mapsto m_0), (k_1 \mapsto m_1)$\}. Dictionaries are also monoids; given two dictionaries $d$, $e \in M[K]$, we define $(d + e)[k] =_{def} d[k] + e[k]$.

In Sketch dictionary keys are used to indicate the location of a data item. For example, a distributed collection of values of type $X$ with partitions on hosts $h_0$ and $h_1$ is modeled as a dictionary $d \in C[X][K]$; $d[h_0]$ is the partition on host $h_0$.

**Linear functions.** A linear (homomorphic) function between two monoids $f : M \rightarrow N$ is a function that “preserves” operations: $f(a +_M b) = f(a) +_N f(b)$, and $f(0_M) = 0_N$. Linear functions are “parallelizable”; they can be applied separately on “pieces” and the results can be combined together. The familiar Map and Reduce operators are linear (Map is between two collections, and Reduce between a collection and a monoid of “reduced” values).

3. Distributed Datasets

Here we describe the core Sketch abstraction, the PDS. We start with the Remote Method Invocation (RMI) layer customized for supporting datasets, and then we proceed to discuss the PDS API.

3.1 Remote Method Invocation

We have built a custom RMI layer tailored to our needs. The RMI layer allows invocation of methods on objects residing in separate address spaces. Although the RMI layer is generic, it is tailored to match the needs of the PDS abstraction. The RMI is designed for relatively infrequent, long-running remote method invocations (the invocation frequency is dictated by the user’s think time, which is on the order of seconds), a small number of clients (e.g., hundreds), and “small” arguments and return values for remote methods (e.g., on the order of kilobytes on average). As transport layer RMI uses Windows Communication Foundation [31] over TCP/IP; WCF also performs authentication.

RMI provides the following services:

- Data serialization\(^2\)
- Cross address-space references to remote objects, dynamic remote object creation, and remote method invocation
- Remote object lifetime management using a combination of leases, keep-alives, and distributed garbage collection

An RMI server maintains a pool of objects referenced by clients through remote references. The server also maintains an ownship map, that maps each object to a list of clients holding references to the object. An object may be shared by multiple clients. The RMI layer assumes that clients do not pass references to objects they own to other clients.

The server maintains a lease for each client; when lease expires, or a client explicitly disconnects, the client-owned state on the server is removed from the ownership map. Objects with no ownership may be garbage-collected. Lease duration is on the order of 10 minutes. Leases are renewed automatically every time a client invokes a server operation. Leases ensure that client crashes do not lead to memory leaks on the server side.

Deterministic computations can have their results memoized, similar to the Nectar approach [21]. Memoizations can be performed both at the client side (saving RMI calls) and at the server side (allowing data structures to be shared among multiple clients who browse the same data). The memoization cache periodically purges objects that have not been accessed for a long time.

The client-side RMI library implements finalizers [3] for remote object references to detect when they are garbage-collected. Periodically a background thread sends lists of unreferenced objects to the servers. Garbage collection works as long as the object graph does not contain cycles that span multiple machines. PDS object graphs are always acyclic.

3.2 The PDS API

A PDS is a generic, distributed, partitioned object. A PDS object implements the IPDS<T> interface\(^3\), shown in Figure 1. One should think of a IPDS<T> object as a distributed tree rooted at the client machine and holding a value of type T in each leaf (see Figure 2). The tree edges are labeled with location information. For example, a path through the tree could be “rack0,machine0,core1”. Using the formalism from Section 2, an object of type IPDS<T> is a dictionary of type T[L\(^*\)], where L\(^*\) is the set of paths.

The IPDS<T> interface comprises only three methods; two of these methods have as arguments user-defined computations (IMap and ISketch) encapsulated in closure objects. These closures and the sketch result of type R must be serializable for transport on the network. The types T and S

\(^2\)Unfortunately WCF does not handle generic datatypes, so we had to implement a custom serialization layer.

\(^3\)The Sketch framework and all visualization applications presented here are written in C#. We use the C# convention of prefixing interface names with I; the angle brackets denote generics, similar to Java or C++. We have slightly simplified the actual interfaces for pedagogical purposes; the actual APIs also support asynchronous execution, error and progress reporting, remote background operation cancellation, and other practical features.

---

interface IPDS<T> {
  IPDS<S> Map<S> (IMap<T,S> map);
  R Sketch<R>(ISketch<T,R> sketch);
  IPDS<Pair<T,S>> Zip<S> (IPDS<S> other);
}

interface IMap<T,S> {
  S Map(T input);
}

interface ISketch<T,R> {
  R Create(T data);
  R Combine(List<R> sketchResults);
}

---

Figure 1. PDS interface.
do not need to be serializable. We discuss multiple implementations of this interface in Section 3.3. Figure 3 illustrates the effects of these methods.

Map transforms an IPDS<T> tree into an IPDS<S> tree with the same “shape” applying the IMap function to each leaf.

Sketch runs a sketching algorithm computing a “small” result of type R. A sketch object s of type ISketch<T,R> contains two user defined-functions: s.Create : T → R and s.Combine : C(R) → R. Given a dataset dictionary d = \{l_0 \mapsto v_0, \ldots, l_n \mapsto v_n\}, define d.Sketch(s) = s.Combine(s.Create(v_0), \ldots, s.Create(v_n)). (Note that the paths through the tree are ignored.)

Zip combines two datasets together into a dataset of cross-products, pairing values with identical keys. Given two dictionaries d, e, the result z = d.Zip(e) is a dictionary z such that z[p] = Pair(d[p], e[p]) for all paths p.

These operations (Map, Sketch, and Zip) are all purely functional: they always create a new result, and they have no side effects (the user-defined functions supplied are required to have no side effects). They are also linear, as defined in Section 2 (Zip is linear in both arguments).

There are natural composition operations on the types IMap and ISketch. For example, the composition of an IMap<T,S> and an IMap<S,U> is an IMap<T,U>. The composition of an IMap<T,S> and an ISketch<S,R> is an ISketch<T,R>.

A concurrent map combines IMap<T,S> and IMap<T,U> into an IMap<T,Pair<S,U>>. Similar concurrent sketches can be defined. Concurrent and composite computation require fewer roundtrips (instead of one for each one of the component functions). As a downside, the intermediate results produced by a composite map are not visible to the application and memoization layers (described below), so they may need to be recomputed if they are reused.

### 3.3 Dataset Implementations

PDSs are abstract objects, accessed solely through the IPDS interface. Sketch provides several basic implementations of this interface, where each solves a different problem in the building of a distributed system. Because all component pieces have the same interface, they can be stacked in arbitrary ways and so by mixing and matching these implementations, users of the Sketch framework can build distributed PDSs with various degrees of parallelism, concurrency, network transparency, and resilience. This modular construction of a distributed system is powerful, elegant, and novel.

- **LocalPDS<T>** is a container for one value of type T.
- **ParallelPDS<T>** contains a list of child IPDS<T> objects. All processing on these objects is performed independently and in parallel. Map returns a new ParallelPDS. Sketch runs recursively on the children and then uses the Combine sketch function on the results.
- **PartialPDS<T>** is a variant of the ParallelPDS<T> which may compute sketch results by aggregating only a subset of the children (the last few returned results are skipped if they are not available within a timeout period, dictated by a policy). This can be used to tolerate child stragglers which take too long to return a result; related functionality is provided by Neptune [12] and Scuba [4].
- **ProxyPDS<T>** contains a single child of type IPDS<S>, and a “frozen” closure of type IMap<S,T>. For a lazy dataset l, l.map(t) = LazyPDS(l.child,Compose(l.map, t)). In other words, the lazy dataset postpones the execution of any Map by composing closures, using of the composition operation for IMap objects, described in Section 3.2. Running a sketch s on a lazy dataset creates a sketch by composing the frozen map and s, and invokes the resulting sketch on the child:

\[
\text{l.Sketch(s) = l.child.Sketch(Compose(l.map,s))}
\]

One subtlety introduced by lazy datasets is the behavior in the presence of exceptions: exceptions caused by a Map call surface only during the execution of subsequent sketches.

- **ReplicatedPDS<T>** is like a parallel dataset, but all children must represent identical values, i.e., they are replicas of one dataset, ideally residing in different fault domains. Like parallel datasets, replicated datasets forward calls to all their children. If a method call on a replicated dataset throws exceptions on some children, the replicated dataset swallows the exceptions and marks the corresponding children as unavailable. (Exceptions that occur on all children are considered deterministic and are returned to the caller.)

Once a child is marked as unavailable there is no protocol to recover it. The recovery is logically part of the underlying storage and transport services on top of which the Sketching framework executes; to handle recovery we would need to devise additional APIs for signaling between the storage, RMI and PDS layers. In all applications we have built, PDS object lifetimes are tied to user interaction sessions, which tend to be on the order of hours. ReplicatedPDS has proven adequate at this time scale for surviving server outages without a recovery protocol.

- **HedgedPDS<T>** is a variant of the replicated dataset, but optimized for speed rather than reliability, implementing hedged requests (i.e., first response wins) [16].
3.4 Putting the Pieces Together

Figure 2 shows a typical tree-shaped distributed PDS object. The leaves of the tree are all LocalPDS objects located on the server machines, holding references to the partitions of the data (of type T) that is being processed. On each server, all leaves have a common ParallelPDS parent. Since the parent invokes all methods on its children in parallel, this server-level ParallelPDS yields multi-core parallelism.

Server 1 contains a second ParallelPDS layer, with two children, both located in Rack 0 (one being on the same machine); the child in Server 0 is referenced through a ProxyPDS. This second-level ParallelPDS provides rack-level aggregation.

Finally, the client machine contains a set of proxy objects, one for each rack where data resides, and a ParallelPDS tying all of them together for cluster-level parallelism.

The client application interacts with the root of this distributed tree in its local address space. In fact, the client cannot tell the difference between a LocalPDS running on the local machine and a distributed data set running on the cluster. Even failures on remote machines (including machine crashes) are caught by the RMI layers and transformed into ordinary C# exceptions in the address space of the client.

3.4.1 Bootstrapping

A Sketch service is a persistent service running on a cluster accessed using the RMI layer. After starting each server hosts a LocalPDS<Empty> object, with a predefined remote object name for the RMI layer. Any authorized client can create a ProxyPDS to refer to this object.

A client is bootstrapped by constructing an initial PDS from the root down. For example, to create the tree in Figure 2, the client creates one ProxyPDS for each rack representative and hooks these proxies as children of the root ParallelPDS. Invoking the Map method of the root, the client recursively instructs each rack-level node to create a subtree referencing the machines in the local rack. The shape of the aggregation tree is decided by the client itself.

The resulting distributed tree object is the seed PDS for this client. As long as the seed PDS is alive (i.e., it has not been garbage-collected), it uses a timer to execute a periodic “keep-alive” Sketch, used only for its side effect of renewing the leases for all server-side objects held by the client. To load a useful IPDS<T>, the client calls a Map<Empty,T> method on the seed PDS.

Note that the Sketch service does not have a central coordinator; each server provides the service completely independently of its neighbors. Each client acts as a separate control point for the computations it initiates.

To provide fault tolerance, the input data must be replicated (replication is performed by the storage layer, and thus is outside the scope of the Sketch framework). To load a replicated dataset the client must insert ReplicatedPDS nodes in the appropriate places in the seed PDS. Interestingly, ReplicatedPDS nodes can be used at any level in the PDS tree: either just above leaves, above the rack-level nodes, or even as the root of the complete hierarchy.

The client maintains for each PDS its lineage: the sequence of computations (Map, Zip, Sketch, with their arguments) that led to the PDS creation starting from the seed object. Since all PDS computation arguments are serializable, the lineage itself is serializable, and it can thus be persisted in files and shared between users.

3.4.2 PDS invariants

Figure 3 shows how computations operate on PDSs. Running a Map on a PDS produces another PDS with the exact same shape and with the corresponding components located in the corresponding address spaces. Running a Sketch produces a scalar result aggregated from leaves up the tree. Executing a Zip on two PDSs with the same shape produces as a result with the same shape. Thus, by induction, all PDS that descend from the same seed have the exact same shape.

From the client point of view, all method invocations on a distributed dataset are single-threaded and atomic, and seem to be executed in the local address space; in reality all computations run concurrently in a distributed system.

Any exception thrown during the execution of a PDS is propagated up the tree, across proxies, all the way to the client. If an exception occurs in tree node, the partial results produced at the other tree nodes remain unreferenced, and are eventually removed by the garbage collector. Returning
a root reference to the client is an atomic operation that indicates the termination of a distributed Map or Zip computation. Sketches complete if and only if they return a value.

4. Applications

The IPDS API may seem overly restrictive: it contains only three operations: Map, Zip, and Sketch. Of these, the first two create new datasets. Thus the only way to “extract” some information out of a dataset is through Sketch calls. Moreover, the LazyPDS construction implies that any sequence of Map and Zip computations ending with a Sketch is equivalent to a Sketch.

A lot is known in the theory literature about the expressive power of sketches [13]. For example, it is impossible to decide using sublinear memory whether a stream of items contains no duplicates. Despite these limitations, we claim that the IPDS API is rich enough to implement useful applications. In particular, we are interested in visualization applications, which need to render visual results of a size independent of the data size — bounded by the screen resolution. We describe now three applications of significant complexity that were built on top of the Sketch framework, by exploiting the API limitations to our advantage.

4.1 A Spreadsheet for Large Tables

Our most complex\(^2\) application is a spreadsheet for browsing distributed datasets. The data model is a database-like table, with a known schema (a list of typed columns). A table can have many rows (billions) and few columns (hundreds). The table data is manipulated through an IPDS<Table> object, where Table is a simple (non distributed), immutable, relational table object. Our Table implementation contains references to a schema, an array of columns, and a bitmap with 1 bit per row, used for filtering operations.

The spreadsheet application does not depend on the actual storage substrate. Our current implementation can ingest data from text files, from a set of SQL Server databases (one for each server), from the Cosmos [10] distributed storage system, and also from a simple custom column-store; it should easily accommodate other distributed storage systems. The Table lazily loads the browsed columns in RAM when using the column store\(^3\). To use multi-core parallelism, after loading each Table may be further split into smaller Tables containing ranges of consecutive rows, referenced through a ParallelIPDS<Table>.

Figure 4 displays several screenshots showing different views of a dataset comprising 4.1 billion tweets with 12 columns. Users can filter (using arbitrary per-tuple predicates), search, add a column (computed using a user-defined function applied to each row), sample, find heavy hitters (most frequent tuples) using the Misra-Gries algorithm [33], sort lexicographically on arbitrary column combinations, draw charts (multidimensional histograms and heatmaps), and perform set operations between different views of the same dataset (intersection, union, difference, etc.)\(^6\).

Each spreadsheet window maintains a reference to an IPDS<Table> and shows a rendering of the dataset. User actions can either change the rendering (e.g., scroll, chart), or change the dataset itself (e.g., filter), causing a recomputation of the rendering. All renderings presented by the spreadsheet are computed using sketches.

Filtering operations compute a new bitmap, which is packaged into new Tables with the column data of the filtered table (the data itself is never copied or modified). Zooming and sampling are special forms of filtering. The set operations also use bitmaps, and they are implemented using a Zip followed by a Map.

Sorting is the main operation performed on tabular data displays. Users can choose which columns to show or hide, but the visible columns are always sorted in some lexicographic order. Sorting computes only the rows visible on the screen, using a TopK Sketch, which includes the following:

- Filter out (ignore) all tuples that are smaller (in the current sort order) than the first tuple shown on the screen,
- Out of the remaining tuples keep the \(k\) smallest\(^2\) distinct tuples, together with their multiplicity (i.e., the number of times each tuple appears in the complete distributed dataset).

For \(k\) in practice we choose 10 times the number of rows displayable on the screen \((k \approx 1000)\). This allows the user to scroll without requiring a new round-trip for each user click. Additional rows are retrieved if the user scrolls too close to the boundary of the available data, by invoking a TopK Sketch using as first tuple the last displayed tuple.

The cost of TopK (in the absence of indexes) is \(O(n \log k)\), where \(n\) is the size of the dataset. TopK.Create uses a priority heap storing the smallest \(k\) values; TopK.Combine performs mergesort and truncation of the result to \(k\) values. Readers can convince themselves that the set of \(k\) smallest tuples endowed with this Combine operation does form a commutative monoid. (“Zero” is a set with no tuples).

Scrolling by dragging the scroll-bar is the most complex operation on a tabular view\(^8\). Scrolling to a specific offset first determines the first tuple to display on the screen; once known, this tuple used as “first tuple” argument in a TopK sketch. To determine the first tuple the algorithm computes a quantile; for example, in a window 200 pixels tall, dragging the scroll-bar 3/4 from the top must first locate

\(^2\) The total size of the spreadsheet program including the Sketch framework is around 65k lines of C# code, including comments.

\(^3\) We have also started exploring offloading the Table implementation to a relational database, which can use indexes to efficiently perform computations directly on external storage.

\(^6\) We plan to add other operations amenable to sketching-based implementations, such as clustering and linear regression.

\(^7\) If there are fewer than \(k\) distinct tuples, all of them are kept.

\(^8\) Scrolling by paging up/down is efficient; “jumping” within the table is harder.
some tuple between the 75% and 75.5% percentiles of the dataset according to the current sort order (each pixel represents 0.5% of the data, i.e., \(2 \times 10^7\) rows in a \(4 \times 10^9\) row dataset). There are well known sketching algorithms for computing such approximate quantiles (e.g. [14]) whose cost is logarithmic in the required precision. The Sketch spreadsheet takes advantage of the limited screen resolution to use an efficient approximate algorithm.

Charts are graphical displays of the data in a set of 2, 3, or 4 columns. As noted previously [26, 27], when displaying very large datasets, with many more points than pixels on the screen, charts must show aggregate data; charts such as line plots do not scale to billions of data points. The Sketch spreadsheet supports the types of charts suggested by prior work: all these charts are essentially multidimensional histograms. Formally a histogram is a dictionary value of type \(R[B]\), where \(B\) is the set of buckets and \(R\) is the set of real numbers; histograms with a fixed bucket set thus form a monoid. Heatmaps are dense 2D histograms.

Histograms and heatmaps are computed in 3 steps: (1) a Sketch to compute the range of the data, (2) a client-side computation to determine the histogram buckets, and (3) a Sketch to compute the histograms proper. Similar to Map-Reduce, all histograms are obtained using GroupBy-Aggregate computations; unlike Map-Reduce, the set of groups is always finite, and independent on the data size. Histograms are computed on sampled data; in this case the sampling rate is chosen so that the error bars are smaller than 1 pixel in the worst case\(^{10}\). (In general the data range must be computed on the complete data, since otherwise it may lose rare outliers.) Even the cumulative distribution function (CDF) plot is computed using a 1-dimensional histogram, where each bucket has the width of a screen pixel.

Unlike most other systems for data visualization (e.g., Tableau [44]), the Sketch spreadsheet uniformly treats all values as continuous, including categorical values. This is necessary because the number of buckets in a histogram is always bounded by the window width; if there are too many

\[\text{Figure 4.} \text{ Spreadsheet screenshots of a dataset of tweets with 4.1 billion rows and 12 columns, manipulated on a cluster with 155 machines (1240 cores). Clockwise from left-top: tabular view sorted on 4 columns, 2D histogram with CDF showing tweet length distribution colored with tweet time, 3D histogram (array of histograms): showing SpamScore/AdultScore/TweetLength, and heatmap of tweet length vs creation time.}\]

\(^{9}\text{An array of histograms as in Figure 4 is also a histogram.}\]

\(^{10}\text{Interestingly, assuming that sampling can be performed in linear time in the result size, the work to compute a histogram is constant, irrespective of the dataset size, and is only a function of the display resolution!}\]
categorical values to fit on the screen, the user will have to zoom in to discover the structure of individual categories.

Each window maintains a history of the rendered data together with the renderings themselves. These are, by construction, small objects. The history enables the user to navigate back and forth instantly.

Observe that the actual Table data is never moved from its home machines (in contrast to the shuffle phase of Map-Reduce). The only values that cross the network are the sketch results used in renderings.

4.2 Browsing Distributed Logs

One of the most useful tools for debugging distributed applications is a simple, reliable, and inexpensive distributed log browsing service. Our second Sketch application is such a distributed log browser. The tool shares the same GUI as the Spreadsheet; it differs in the handling of data loading. In this case the data source is a set of distributed logs generated by some arbitrary cloud services. The logs are stored in plain text files; every hour a new log file is created. The local filesystem storing the logs provides a simple form of indexing on the time dimension. The index enables the use of sublinear time algorithms (e.g., cost linear in the size of the logs explored by the user, and not of the complete logs).

When loading logs, the user has to specify a time window of interest. The data-loading code parses only the log files overlapping this interval. After parsing each log file is loaded into an object subclassing Table from Section 4.1. This allows the logs to be browsed using the Sketch spreadsheet GUI. Some of the results presented in the experimental section were extracted using this tool from the Sketch RMI servers logs.

4.3 ViewCluster: Distributed System Perf. Analysis

An ideal tool to understand and debug the performance of a distributed system allows the user to visualize performance at multiple resolutions (e.g., cluster/machine/thread and second/millisecond/microsecond). In this section we describe the ViewCluster tool, which can be used for offline performance analysis based on event traces.

In a typical example, a global view indicates that the entire distributed system is stalled because of a straggling machine; zooming in identifies the cause as a heavily contended lock affecting the straggler. Timescales in this scenario vary from minutes to microseconds, and the analysis subject changes from a coarse-grained representation encompassing tens of machines and their communication patterns to the detailed interactions of a small number of threads on a single machine. Fine-grained tracing can produce on the order of gigabytes per minute.

There are a number of challenges to overcome to realize this vision: (1) voluminous traces collected on many machines must be synchronized and the events within must be correlated across multiple machines; (2) zooming and panning at various resolutions requires scanning and processing large numbers of events; (3) fast response requires the use of in-memory data structures with appropriate indexes.

The original version of ViewCluster predated the Sketch framework, and was a single-machine implementation; it copied traces from all analyzed machines to the client machine for analysis. By refactoring the tool around the Sketch framework we have obtained a distributed application which browses and analyses the traces in place. The rewriting effort took around one person-week.

Using Sketch provides immediate benefits: (1) data remains distributed, so no copying is necessary; (2) the work performed by each machine is constant, irrespective of the size of the monitored system: trace parsing and data navigation are performed in parallel on all machines, and scale with the size of the monitored system. We have even (ab)used Sketch to start and stop distributed trace collection, by executing sketches that run scripts with side-effects.

Visualization is a powerful tool for performance debugging, but providing a good user experience requires interactive response times. Data loading is a relatively slow process (linear in the size of the collected traces, usually in the order of a few minutes); loading requires parsing on-disk trace files to construct a complex in-memory representation of system activity on each server, including multiple indexes used to navigate the data structures quickly on user requests. After parsing, data exploration can be done at interactive speeds: the response time is on the order of seconds for clusters with tens of machines.

The ViewCluster application uses a data model significantly more complex than just a simple table. This highlights the power and flexibility of the Sketch framework to handle even complex datatypes.

Trace collection and visualization. Activity on the analyzed systems is recorded using Event Tracing for Windows (ETW) [38]. ETW is a well-engineered, low-overhead tracing infrastructure built into Windows. Any application, including the OS kernel, can post events using standard interfaces. For fine-grained tracing our tools collect both low-level events (context switches, inter-thread signals, deferred procedure calls, interrupts, network packet sends and receives), as well as high-level activity (memory allocations, garbage collection, and application-specific events). In addition, tracing captures full stack traces for a subset of these events. ETW is used to collect traces individually on each machine, and ViewCluster is used to “stitch” all traces into a visualization of the complete distributed system.

Figure 5 shows 3 screenshots of ViewCluster, with increasing degrees of detail from left to right. The X axis is always time — the first two views cover 0.1 second, while the rightmost view covers 6 milliseconds. The Y axis represents hierarchically the usage of computational resources; each “row” shows the activity on one computer. Colors indicate the type of activity (user-level, system-level, garbage-
collection, interrupt handler, user-specified code region, etc.). The left-most image in the figure displays the usage of each machine; the middle one shows average CPU utilization using stacked bars in each time interval. The right-most picture shows CPU utilization for each of the 8 cores of each machine; the yellow lines are network messages exchanged between machines.

The user can scroll on both axes, and zoom in/out; she can also change visualization details or can mouse-over to find out more details about a specific event. When she focuses attention on a single machine, a separate window is created to browse just events local to that machine (not shown).

**Sketches.** All trace renderings are computed using sketches. Each sketch is parameterized by the time range visible on the screen and by the window resolution in pixels. For example, in the middle view, average CPU utilization is computed and displayed for each time interval corresponding to 5 pixels, which is about 1/2 millisecond in this image. Zooming in would cause a recomputation of the average CPU utilization for a shorter time interval. Currently a sketch fetches data for all machines; this decision should be altered when analyzing very large clusters, by using scrolling on the vertical axis to trigger the computation just for the visible machines. The Sketch computation takes advantage of the index data structures constructed during trace parsing to touch only those events visible in the displayed rendering.

Computing the set of messages to render is a challenge for the PDS API, because the send and receive events are on different machines. This effectively requires a Join computation, where data is not partitioned on the join key. We employ several techniques to solve this problem: (1) we offer the option to display only messages that have long latencies; these can be computed using the hashing-based scheme in [36] to find vectors with large $l_1$ norm in a stream given by the vector endpoints, (2) when zoomed-in we filter out messages that are not sent or received within the displayed interval, reducing the computation to an intersection between two small sets, and (3) when zoomed-out, we aggregate all messages close to each other in time and display a single line for a set of messages. The lines are computed using two rounds of sketching: first, the “receive” events are processed, aggregating the ones that are received within a pixel-sized time-interval (the list of aggregated events thus produced cannot exceed the screen resolution). The second Sketch is essentially a broadcast-join followed by further filtering (of messages send outside the displayed interval) and aggregation (of messages with both endpoints very close to each other).

5. Evaluation

In this section we evaluate properties of the Sketch framework using the Spreadsheet application. We measure first interactivity and scalability (Section 5.1), next we compare performance with VisReduce [26] using a common dataset (Section 5.2) and finally we test fault tolerance (Section 5.3).

**Hardware platform.** We run our experiments on a 155 node cluster. All machines use Window Server 2012 R2. Each is equipped with an 8-core 2.1GHz AMD Opteron 2373 EE processor, and at least 16GB of RAM (some machines have more). Each machine has two volumes: the OS is separate from the data; data is on a logical volume striped over 3 1TB 10K RPM disks using RAID0. The machines are organized in 5 racks of roughly the same number of machines, each with a top-of-rack switch. The network is 1Gbps Ethernet. In our evaluation there is only one client connected to the service, located outside the cluster network, but on the same LAN (network latency < 2ms).

5.1 Interactivity and Scalability Evaluation

In the first set of experiments each server processes the same amount of data. The dataset is a table with 12 columns representing a set of tweets; the full dataset, shown in Figure 4, contains 4.1 billion tweets. Total data size is about 1TB. Data is partitioned almost uniformly on machines, for 26.45 million rows per machine, further divided into 8 parts on each machine, of 3.3 million rows in each part. The underlying storage is a simple column store.
Figure 7. End-to-end time on client to perform various sketches. X axis is number of server machines, Y axis is time in milliseconds. Note that the Y axis scale differs in these plots, for better readability.

Browsing speed. We measured the end-to-end execution time for each operation for an interactive user-controlled data-browsing session on the full 4.1B rows dataset (histograms, tabular views with scrolling, and heatmaps). The average latency between the user initiation and the final rendering is 560ms; with a maximum of 7.6s. Loading data is the most expensive step, in particular non-compressible string columns require expensive I/O\textsuperscript{11}.

Network utilization. We also measured network message sizes during this browsing session. In our traces the largest sketch results were generated by Topk sketches required to render Spreadsheet tabular views; each was around 80KB. As described in Section 4.1, such sketches transport around 1000 table rows. A concurrent histogram+CDF sketch uses 20KB. The largest requests had size 3.8KB.

Procedure. The remaining experiments in this section were scripted (there was no user think time). Each experiment is repeated 7 times, and error bars show variability after the two extreme values are dropped. The file caches are warmed, so disk I/O cost is not a factor in these results. Result memoization is disabled, because it would provide instantaneous answers when running repeated computations.

Communication latency. To get a baseline for the communication costs we measure the end-to-end execution time for a Null sketch; the Null sketch Create and Combine function just return true. (The Null sketch is executed periodically by the client application as a keep-alive, as described in Section 3.4.1.) Figure 7(a) shows 2 sets of measurements: with (red/continuous line) and without (blue/dashed line) per-rack aggregation.

Without rack-level aggregation, the Null sketch latency increases linearly with the number of servers, as expected, due to contention on the single client network interface communicating with all servers. The rack-level aggregation layer provides a flat response time over this set of machines. The two lines cross at 16 servers. With similar constants we expect that a second-level intermediate aggregation layer is useful for clusters with more than 16*16 = 256 machines. Facebook’s Scuba [4] uses an aggregation fanout of 5, which is much smaller.

Scale-out. The next experiment measures the time to compute the histogram of the CreatedAt column, which contains the DateTime when each tweet was created. Figure 6 shows the timeline of the computation when there is no rack-level aggregation. Displaying a histogram requires 3 sketches, as explained in Section 4.1. (1) The first sketch computes the range and precision of the data. The precision is computed using heuristics: the code checks whether all numbers are integers, or all dates have an integer number of minutes\textsuperscript{12}. After receiving these results, the client decides the histogram bucket boundaries and initiates 2 more sketches concurrently (concurrent sketches are described in Section 3.2): (2) the histogram proper, and (3) the CDF. After receiving all these results the client renders and displays the histogram on the screen (a display such as the one in Figure 4 top right).

The end-to-end execution time for a sketch is the maximum of the network time and server compute time over all servers (see Figure 6). Since data is uniformly partitioned, ideally the server compute time should be independent of the number of machines if servers are not overloaded.

\textsuperscript{11}The tweet “Text” column contains the bulk of the data, and loading it requires 22 seconds — this data was not explored in the user session measured.

\textsuperscript{12}The precision tests are quite expensive when applied to DateTime values — which is the case in our benchmark; computing the range and precision for a numeric column is substantially faster.
Figure 8. Out-of-core scaling performance for computing a histogram using SQL Server.

Figure 7(b) shows the execution time \(t_r\) in Figure 6) of the first sketch, that computes the data range and precision. Figure 7(c) shows the end-to-end time \(t_a\) in Figure 6) to compute all 3 sketches. The data-range computation dominates at 65% of the end-to-end histogram time. The last 2 sketches are computed on sampled data, as explained in Section 4.1. This suggests an improvement for future work: the data precision of a column should be precomputed and saved as part of the column metadata. (The data range needs to be recomputed every time, because the range changes with the filtering operations applied.)

Single-threaded implementation. As a reference, we also measured the cost of a histogram when run on the local client, without using the Sketch framework. On the client the code is single-threaded. We used the same dataset present on a single server. The end-to-end local time is 5.5 seconds. Comparing to the leftmost point in the graph in Figure 7(c), we conclude that, despite network overheads, Sketch on an 8-core server is 5.9 times faster than a local implementation, because the Sketch uses all cores on the server effectively, without requiring any support from the programmer.

Out-of-core scalability. To understand the limitations imposed by in-memory data structures, we explored scaling up the workload on a single server machine. We use SQL server to implement the histogram as a pair of SQL queries (one for range and one for histogram proper). The SQL server is configured to use an index on the histogrammed column. SQL server’s range computation is simpler (computing just the min and max of the dataset, but not the precision), and runs in constant time due to the index.

Figure 8 shows the time to compute histograms for data sets of increasing size. At 25 million rows, due to the lack of a column store, SQL Server starts computing out-of-core, showing a steeper slope. These results suggest that, with the aid of indexes and using efficient sequential I/O, Sketch may be viable even with datasets too large to fit in RAM.

5.2 Visualizing OnTime Flight data

In order to provide a comparison with related work, we reused the dataset from VisReduce [26]: the OnTime flights database. The dataset consists of 321 files, each around 250MB, for a total of 65G uncompressed. There are 159M tuples, with 109 columns each. We distribute the files in a round-robin fashion on our machines. We experiment with 5 machines to match [26], having then 64 files/machine, but also with 155 machines, having about 2 files/machine (3 files on 11 machines). The files on a machine are all loaded as children of a single ParallelPDS.

Since it is difficult to make a direct comparison, we have carried several experiments using the full dataset. Table 1 shows the measured end-to-end execution time for various scenarios; [26] gives several numbers, and we could not understand the difference in experimental setup, so we list a range. When caches are cold, the first pass over the data triggers data loading; in this case [26] wins due to better column store compression. Histogram computation follows the range sketch, and always find the data in memory and thus is much faster.

5.3 Fault Tolerance

We have also exercised the fault tolerance provided by the ReplicatedPDS using a simple experiment. Here we use a cluster of 4 identical servers; each server runs a SQL Server 2014 instance containing a copy of the same AdventureWorks database [32]. The root PDS is a ReplicatedPDS, with 4 ProxyPDS children, one for each machine.

Our failure injection (1) shuts down instances of SQL Server or (2) kills the Sketch service running on a server. Failures are detected by timeouts; in case (1) there is a timeout in the database connection, and in case (2) a timeout in the WCF communication layer. Timeouts are translated by the runtime into exceptions, which are caught by the ReplicatedPDS. In all cases, the ReplicatedPDS successfully switches to an alternate replica, allowing the client to continue computing from the current state after a short interruption in service. The system successfully tolerated the failure of 3 of the 4 replicas.

We have also run an experiment crashing client applications. We have verified that server-side client-referenced objects are reclaimed when the client leases expire (we know because servers periodically log the total amount of memory used by the managed runtime).

6. Related Work

There is a broad body of work related to the Sketch system.
Parallel computation and visualization. It is impossible to do justice to the immense amount of work on parallel visualization and rendering, starting with [37]. There are sophisticated parallel rendering and visualization toolkits available, for example ParaView [43]. These tend to have much richer data models and to provide much more sophisticated functionality than Sketch. The Sketch design decouples the parallel-execution framework from the actual data model and rendering. It would be an interesting exercise to see how many of the operations available in fully programmable systems can be recast in a framework such as Sketch.

Sketch is a sort-middle type of rendering pipeline [15], because the data is arbitrarily partitioned, and the rendering is computed on the client side from aggregated data summaries. Initial versions of Sketch attempted to do rendering sort-last, by overlaying actual bitmaps rendered on a transparent background. The examples in [9] show why such an approach is incorrect; fundamentally it is because bitmap overlays do not form a commutative monoid.

Sketch is to some degree related to Zoomable User Interfaces, introduced by the Pad system [39]. ZUIs tend to have multiple distinct semantic layers (e.g. [23]). The idea of Responsive Design [5] is to adapt a web site to the screen resolution; Sketch extends this idea to data visualization (but Sketch is not the first system to propose this approach).

Polaris [44], commercialized under the name “Tableau” is a successful visualization platform for tabular data. Sketch is a complement to tools such as Excel or Tableau, designed to work in the regime where there are many more data points than pixels on the screen.

The MPI Reduce [42] primitive is strongly related to the aggregation model of Sketch, but it is a lower-level, non generic interface.

Large-scale analytics. The PDS Sketch operation is functionally equivalent to the Neptune Data Aggregation Call architecture [12]. Unlike Neptune, Sketch provides a stateful object model (the PDS) with distributed garbage-collection. Our work is also geared towards visualization applications, tying screen resolution to computations using sketches.

Several systems including Dremel [29] (commercialized as BigQuery), PowerDrill [22], Apache Drill [24], Druid [50], and Scuba [4] are oriented towards efficient execution of aggregated queries and visualization of large distributed databases. Sketch decouples the aggregation layer from the application, and allows manipulation of very rich datatypes and displays. In particular, applications such as ViewCluster show that Sketch is applicable to rich data types, beyond simple relational data models. On the other hand, Sketch could benefit from many of the optimizations from these systems, including better column storage and indexing. Some of these system do not provide a clear semantics when results are very large (e.g., Scuba limits results in leaf aggregators to 100K rows to “avoid memory issues and rendering problems”, and Dremel computes TopK only approximately).

[18] presents a theory of aggregated information visualization. The idea of restricting visualizations to aggregates for large data is proposed by imMems [27]; imMems pre-computes cubes for faster rendering; this idea and the use of indexes could be adapted by our Spreadsheet. VisReduce [26] also uses a system architecture for computing user-defined aggregation functions, as well as incremental renderings. MapReduce [17] is adapted for complex big-data rendering in [47], foregoing interactive response. [9] identifies several problems with traditional visualizations, but proposes heuristic solutions. Sketch uses the screen resolution to select rendering precision by restricting errors bars to be below one pixel in size, computes CDFs as histograms with pixel-sized bins, computes TopK based on the number of displayed rows, and scrolls quickly using approximate quantiles, all new ideas in this context, to our knowledge.

A fair amount of work focuses on incremental visualizations. Early work [25] synthesizes incremental queries on a database. [20] evaluates user interfaces for incremental visualizations; Tempe [30] uses a generic query language; [11] uses special user annotations on data; Stat! [8] uses a programmable streaming engine. The problem of distributed aggregation is one of the core problems of sensor networks [41]. Sketch is built on top of a distributed, garbage-collected partitioned object model, and is not tied to a database. It would be an interesting exercise to adapt the Sketch model to incremental visualizations.

There is also a large amount of work on sampling-based data visualization; recent work related to big data is BlinkDB [6] and [49]. These techniques could further accelerate the Sketch Spreadsheet.

Splunk [1] is a log-analysis system for distributed logs, related to our log-browser application; it has a similar query architecture, where the “search head” corresponds to our client, and the “indexers (search peers)” correspond to the Sketch servers. Sketch seems to provide better scalability than Splunk, but could certainly benefit from the rich set of log-parsing heuristics and tools. IBM BigSheets [2] also provides a spreadsheet user interface for big-data manipulation.

The Fay log-analysis system [46] is related to ViewCluster by collecting ETW events from a large cluster; however, our viewer is optimized for browsing, and not for analytics. Data virtualization [45] is a well-known technique for building large spreadsheets, and closely related to paginated queries in databases and to the display of results in search engines. The architecture of search engines also relies on aggregation trees. All these influenced the Sketch spreadsheet.

The modular architecture of Anvil [28] is similar to the various implementations of the PDS interface in our dis-

\[\text{\textsuperscript{14}}\text{From the Splunk documentation: “To avoid negatively impacting web browser performance, the Splunk Enterprise charting library places a limit on the number of points that can be plotted for an individual chart.”}\]
tributed objects. Several systems are stateful; for example: Grappa [35] keeps state local and delegates computation to remote locations, but the computational model is quite different; [19] is geared towards batch processing, and Piccolo [40] uses a distributed key-value store.

7. Conclusions

The core observation driving this work is that visualizations render data with precision necessarily bounded by the screen resolution. To render a large dataset on a small screen one has to aggregate information in some way. We addressed the following question: “What if aggregation is the only thing one is allowed to compute on a large dataset?”

To answer this question we built a generic system for distributed aggregation and used it to build three useful and complex data visualization applications: a spreadsheet, a log browser, and a tool for debugging distributed application performance. The aggregation interface endows these application with desirable properties, such as simple parallelization, low-bandwidth requirements and natural scalability, which we have quantified up to 1240 cores.

We finally described Parallel Distributed DataSets, or PDSs, a distributed stateful object architecture for building distributed aggregation networks. PDS objects hide the complexity of programming distributed systems from clients. They offer a divide-and-conquer approach to the construction of distributed systems by using simple modules with identical interfaces that independently address problems such as remote communication, parallelism, fault tolerance, bandwidth reduction, and straggler avoidance.

References
