VisReduce: Fast and responsive incremental information visualization of large datasets

Jean-François Im*, Félix Giguère Villegas†, and Michael J. McGuffin*
*École de Technologie Supérieure, Montréal, Canada
Email: jfim@jean-francois.im, michael.mcguffin@etsmtl.ca
†Mate1.com, Montréal, Canada
Email: felixgv@gmail.com

Abstract—**Performance and responsiveness of visual analytics sytems for exploratory data analysis of large datasets has been a long standing problem. We propose a method for incrementally computing visualizations in a distributed fashion by combining a modified MapReduce-style algorithm with a compressed columnar data store, resulting in significant improvements in performance and responsiveness for constructing commonly encountered information visualizations, e.g. bar charts, scatterplots, heat maps, cartograms and parallel coordinate plots. We compare our method with one that queries three other readily available database and data warehouse systems — PostgreSQL, Cloudera Impala and the MapReduce-based Apache Hive — in order to build visualizations. We show that our end-to-end approach allows for greater speed and guaranteed end-user responsiveness, even in the face of large, long-running queries.**

Keywords-**incremental visualization; online aggregation; information visualization; MapReduce; columnar storage;**

I. Introduction

Visualization for business intelligence often involves querying large databases to generate plots such as line charts, barcharts, scatterplots, or potentially more exotic visualizations [1], [2], [3] such as parallel coordinate plots [4]. These visualizations provide more insight when the user can interact with them to quickly refine queries, drill down, or choose new paths of exploration. Unfortunately, common back-end systems for processing very large datasets have one or both of the following problems: (1) they exhibit high latency, precluding feedback at interactive rates, or (2) they require expensive pre-computations (such as indices or datacubes) that accelerate restricted classes of queries, but do not accelerate *all* of the common queries involved in data visualization.

For example, systems based on SQL are not designed to run arbitrary code as part of a query[1]. Hence, generating a scatterplot or parallel coordinates plot — which require all data tuples for rendering — of a large dataset with SQL requires running a query that transfers all the data tuples from the database to the client, and then generating the plot on the client. The client becomes a bottleneck, and the work of generating the visualization cannot be distributed over multiple nodes. Scalability is thus severely limited.

[1]Technically, many DBMS support user defined aggregation functions, as proprietary extensions to SQL. These have different caveats depending on the DBMS. For example, MySQL requires them to be written in C, Oracle requires PL/SQL, C, C++ or Java, SQL Server requires them to run on the .NET framework while PostgreSQL supports many languages. Furthermore, the C APIs are incompatible between DBMS. Also, depending on the database and foreign language combination, there may be a significant overhead to calling a foreign function millions of times. Finally, DBMS-internal languages have no intrinsic support for image manipulation.

OLAP datacubes [5] support fast aggregation queries, but only over the dimensions that were included during the construction of the cube. With large datasets involving 20 or more dimensions, constructing a "full" cube with all dimensions is often not feasible [6] due to memory restrictions, and queries involving dimensions that are left out will not be possible.

MapReduce [7] has recently gained attention as a tool for processing large data, but is designed for batch jobs, and has high latency. Just starting up a new job can require several seconds with typical implementations. To quote Heer and Shneiderman, "While popular platforms for large data analysis such as MapReduce achieve adequate throughput, their high latency and lack of online processing limit fluent interaction" [8].

Interactive visualization of large data is an important problem, covering two challenges listed by Johnson ("human-computer interaction" and "scalable, distributed, and grid-based visualization" [9]) and two more listed by Chen ("usability" and "scalability" [10]). Tableau, a leading commercial front-end for visualization of databases for business intelligence, has only limited features for dealing with very large data. However, more powerful solutions would clearly be valuable: "Tableau has users with very large databases who are willing to wait minutes for database queries to run so that they can see a graphical view of their valuable data. However, users do not want interactive experiences that include such pauses." [11]

We present a simple yet promising solution called VisReduce, that (1) allows queries to run arbitrary code on worker nodes (unlike SQL-based solutions); (2) scales up in speed as the number of worker nodes is increased; (3) gives continual feedback to the client about the progress of a query, so that the user knows roughly how long they will need to

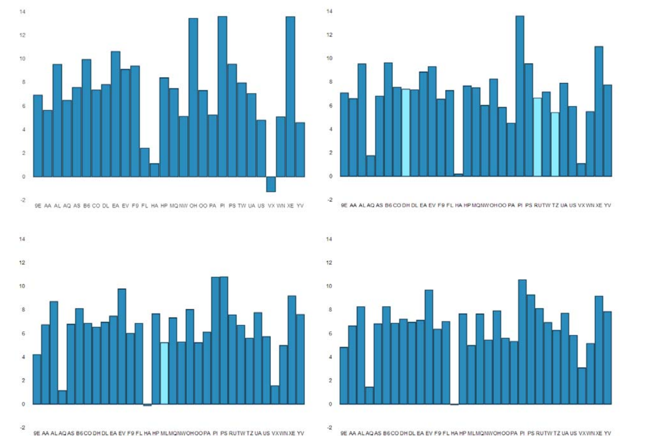


Figure 1. In this example, the output being displayed by the VisReduce client is a barchart. As the query progresses, the barchart is updated incrementally, shown here as a sequence of screenshots, starting at top left and ending at bottom right. Respective times since the start of the query are 270 ms, 1137 ms, 3988 ms and the completed result at 5021 ms shows average delay for all 149.5 million flights. The bars initially oscillate in length, but quickly converge to their final lengths as the query progresses and becomes more accurate. As the query progresses, new bars are occasionally created and inserted (these have been manually highlighted in this figure with a lighter color).

wait; (4) incrementally updates the result displayed by the client, so that the user sees an approximate visualization of the data processed so-far during the entire query, as illustrated by the bar charts of figure 1. The visualization is displayed within 1 second of launching the query and is updated frequently and continually during the query, gradually converging toward the final result, allowing the user to cancel a query before it has finished if they so desire. VisReduce achieves these properties by avoiding all large transfers of data between nodes, never writing large output files to disk, leveraging compressed columnar storage data formats, and keeping runtime environments on worker nodes "warm" (i.e. persistent) rather than starting up new runtime environments each time a new query is initiated so that processing can start and visual feedback can be displayed within 1 second.

We evaluated VisReduce by comparing its performance with three other popular systems: Apache Hive (built on top of Hadoop MapReduce), Cloudera Impala (a recently-released implementation of Dremel for Hadoop), and PostgreSQL. Tests were performed with the OnTime[2] flight data set, which has approximately 150 million records and over 100 columns. Of these systems, VisReduce is the only one that provides an incrementally updated visualization during the query, and the results of our tests indicate that it also completes queries significantly faster than the other systems. We feel that the design ingredients of VisReduce provide valuable lessons for the design of future interactive visualizations engines for large datasets.

[2]http://www.transtats.bts.gov/Fields.asp?Table_ID=236

II. Related Work

Rapid interactive visual queries on databases were pioneered with Shneiderman's *dynamic queries* [12], which emphasized the value of providing real-time feedback to the user for a tight interaction loop. TreeJuxtaposer [13] is an example of a visualization that achieves a guaranteed constant frame rate during navigation through a large tree structure. It achieves this by progressively rendering data into the video card's front buffer (rather than the usual back buffer), drawing the more important data and landmarks first, and stopping the rendering whenever a new frame must be started. This way, if there is not enough time to draw the full data set in the allocated time for a frame, the user at least sees the most salient information before the next frame is started. VisReduce is designed to scale up to much larger data sets, but still adheres to the idea of displaying a visualization that is updated often (several times per second) and becomes increasingly accurate as the query progresses.

We now survey the most relevant types of backends for large data processing.

SQL-based systems are programmer-friendly because the query language is familiar and easy-to-understand. Examples of systems that expose an SQL-like language include Dremel [14], a query system developed at Google for low-latency querying of large data sets stored on their infrastructure, and Cloudera Impala, a recently-released open-source implementation of the same concept. As explained in the introduction, such systems cannot efficiently generate a scatterplot or parallel coordinates plot, because they would require all the data to be first transferred to the client for rendering.

Of particular note is the work of Hellerstein et al. [15] on implementing online aggregation inside of a database engine, in which they explain how they implemented online aggregration of simple aggregate statistics (sum, count, avg, var and std dev) in Postgres.

OLAP datacubes [5] only accelerate queries on the cubes that have been pre-computed. As mentioned in the introduction, with large data sets, these cubes cannot incorporate all dimensions, otherwise they become too large. The imMens [6] system has a clever workaround for this problem: it only pre-computes small cubes of 3 or 4 dimensions, and can afford to pre-compute many such cubes, each with a different set of 3 or 4 dimensions. This allows the user to perform brushing and linking on overviews of data with very rapid visual feedback. However, business intelligence tasks can require the user to filter along 3 or 4 dimensions (e.g., to examine only the data for a particular country, a particular year, and a particular month) and then explore further, and such filtering along many dimensions means these small datacubes will no longer be useful. In addition, although many small cubes require far less space than

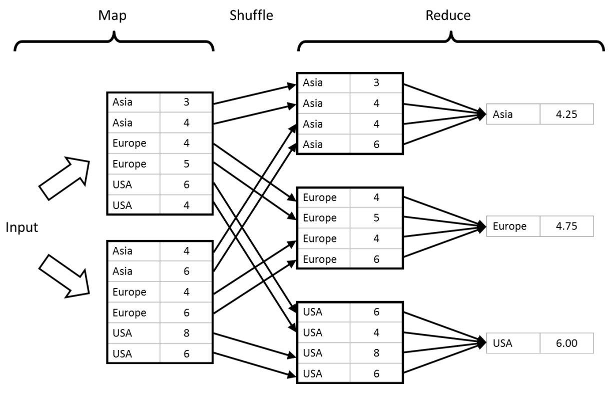
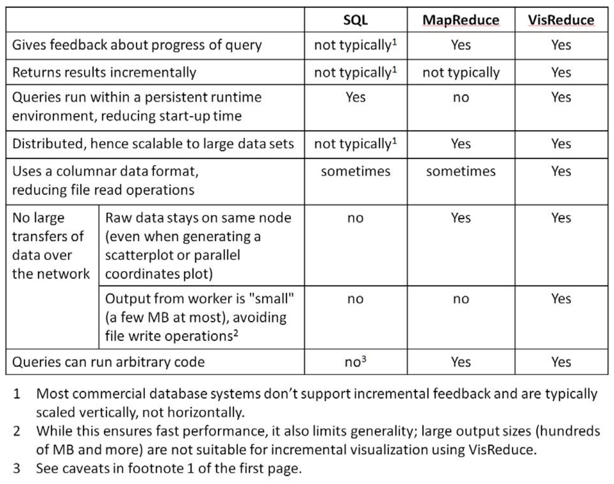


Figure 2. Computing averages with MapReduce. The map, shuffle, and reduce operations are each distributed over multiple nodes. Tables with thick borders may be very large, resulting in slow disk I/O operations and heavy network traffic.



		SQL	MapReduce	VisReduce
Gives feedback about progress of query		not typically[1]	Yes	Yes
Returns results incrementally		not typically[1]	not typically	Yes
Queries run within a persistent runtime environment, reducing start-up time		Yes	no	Yes
Distributed, hence scalable to large data sets		not typically[1]	Yes	Yes
Uses a columnar data format, reducing file read operations		sometimes	sometimes	Yes
No large transfers of data over the network	Raw data stays on same node (even when generating a scatterplot or parallel coordinates plot)	no	Yes	Yes
	Output from worker is "small" (a few MB at most), avoiding file write operations[2]	no	no	Yes
Queries can run arbitrary code		no[3]	Yes	Yes

1 Most commercial database systems don't support incremental feedback and are typically scaled vertically, not horizontally.
2 While this ensures fast performance, it also limits generality; large output sizes (hundreds of MB and more) are not suitable for incremental visualization using VisReduce.
3 See caveats in footnote 1 of the first page.

Figure 3. A comparison of SQL, MapReduce, and VisReduce. VisReduce is the only approach to have all the advantages listed, because it is designed especially for interactive visualizations.

one full cube, they can still require significant space and pre-computation time: in a data set with 20 dimensions, computing all possible 4-dimensional cubes requires storage for $\binom{20}{4} = 4845$ cubes.

MapReduce [7] is a framework used at Google for writing distributed programs that can be run on large clusters of computers. An open-source implementation, Hadoop MapReduce, is a popular approach for handling large datasets. Both are optimized for throughput of large batch jobs, not latency, and thus involve overhead that is a significant barrier to real-time interaction. Just starting up a new job can take several seconds, partly because new Java Virtual Machines (JVMs) must be started on each worker node. Furthermore, running MapReduce on large datasets involves moving lots of data between machines (Figure 2). Finally, common implementations of MapReduce do not return any partial results; the client must wait for a job to complete before seeing feedback.

Some previous work [16], [17], [18] has studied how to structure databases so that queries iterate over data in a statistically random order. This allows confidence bounds on the current result to be computed and displayed throughout the query, so that the user not only sees the current result, but also bounds on what the final result may be, giving the user more information to decide whether they can stop a query before it completes. User studies [19] have shown that such confidence bounds provide end users with valuable information to accelerate decisions by the user. Unfortunately, this approach comes with a significant up-front cost: randomly shuffling all records in the database [16] or constructing an "ACE Tree" [17] structure which itself requires multiple complete passes through the dataset. Our current VisReduce prototype performs no such pre-processing but this also means we cannot provide confidence bounds on the incrementally updated result during a query. Note also that, because VisReduce uses a columnar data format, it can greatly benefit from having data that is *not* randomly shuffled: run-length encoding can greatly compress the columns if there are many repeated values. For example, generating an "average sales by month" barchart of a large dataset with VisReduce would only require reading in two columns, one of which (the "month" dimension) may be greatly compressed because it contains many repeated values, thus saving disk read time.

There have also been efforts to extend MapReduce to allow for online aggregation [20], [21]. These approaches require some time to start up before they can return results. For example, Figure 4 in [21] shows almost 20 seconds elapsing before any progress is made. The latency achieved with VisReduce is much lower (under 1 second) and more suitable for interactive feedback; this faster start up time allows rapid sequences of partial queries to be explored with no perceived waiting time by the user, such as a user executing multiple sequential drilldowns in GPLOM [22] without waiting for query completion.

Figure 3 summarizes some key differences between SQL, MapReduce, and our proposed VisReduce.

III. VisReduce

VisReduce differs from MapReduce by making two assumptions in order to increase the performance to interactive levels:

- the resulting output aggregate is small enough to fit in memory and be transmitted in a reasonable time (less than 250 ms) over the network;
- there always exists an inverse aggregate, so that partial results can be removed from the output aggregate.

An overview of VisReduce running on two worker nodes is shown in Figure 4, which can be contrasted with Figure 2 for MapReduce. One key difference is that MapReduce

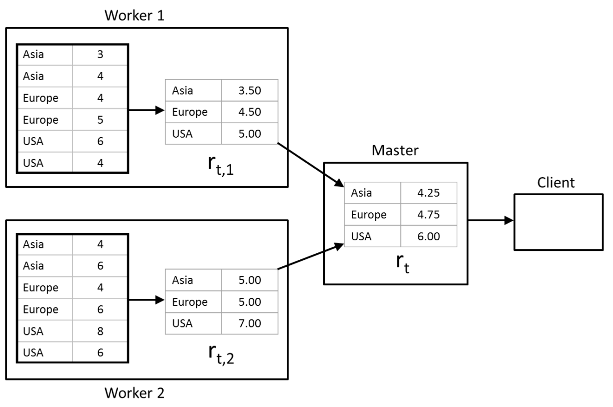


Figure 4. Computing averages with VisReduce over two worker nodes. Tables with thick borders may be very large, but do not leave the worker node they are stored on. Only relatively small amounts of data are transferred between nodes. At time t, each worker w produces result $r_{t,w}$; the master node assembles these into result r_t to be displayed by the client.

doesn't return a result until the entire job is finished. With VisReduce, however, results are computed incrementally, so that at time t, the 2 workers produce partial results $r_{t,1}$ and $r_{t,2}$, respectively. These results are combined by the master node with its previous partial result yielding $r_t = r_{t-1} \oplus r_{t,1} \oplus r_{t,2}$. In the example shown in Figure 4, this partial result r_t might be displayed by the client in the form of a barchart, with the heights of bars gradually converging toward their final heights, i.e., r_t converging toward its final value as t increases.

Another key difference between Figures 2 and 4 is seen by noting which tables are "big" (drawn with a thick border). In Figure 2, there are 5 big tables, requiring large volumes of data to be transferred over the network. MapReduce supports an optional optimization where a combiner is run over the output of the mapping phase before transmitting it over the network, but it still requires writing the potentially large output of the mapping phase to disk. With VisReduce, however (Figure 4), the two large tables remain on their respective worker nodes, and only small results $r_{t,w}$ and r_t need to be transferred between nodes with no disk writes. In Figure 4, the transmitted results are heights of bars in a barchart, and the master node's $\oplus$ operator simply computes average values. Alternatively, if we were using VisReduce to compute a scatterplot or parallel coordinates plot, the partial results $r_{t,w}$ and r_t could be bitmap images, with the master node's $\oplus$ operator performing image compositing.

Because the partial results $r_{t,w}$ and r_t are small in size, VisReduce doesn't need to write results to disk, saving time.

Notice also that, if we increase the number of worker nodes, VisReduce's speed will increase almost linearly, because worker nodes can work in parallel and still only send small results over the network, even if the raw data set grows in size.

Also, no matter how large the dataset is, workers can send frequent partial results to the master (several times per second), to display incrementally updated feedback to the user. This is unlike most database approaches.

As a partial result can be "removed" by inverting it and reducing it into the current state, results can be sent to the client without waiting for completion of an input segment while still maintaing fault tolerance. In contrast, MapReduce's fault tolerance mechanisms require an input segment to be completely mapped before reducing the mapper's output, leading to long delays before the first results are sent to the client in online approaches, as shown in Figure 4 of [21].

Unlike MapReduce, the runtime environments on VisReduce's worker nodes are kept "warm", to decrease start-up time.

VisReduce further saves time by using a simple columnar data store, which is combined with a dictionary encoding (e.g., replacing each string "Dallas, TX" with an integer value) and run-length encoding. Unlike a general database, with VisReduce we care much more about fast agregation performance, and not individual row lookups or updating the data.

Previous work has demonstrated the performance advantages of such columnar approaches. Pavlo et al. [23] compared Hadoop MapReduce with two commercially available massively parallel databases — an unnamed row-oriented database[3] and a column-oriented one (Vertica) — and show that the column database is faster than the other two approaches for aggregation workloads. Abadi et al. [24] discuss how adding various simple compression techniques to a column-oriented DBMS offer significant gains in both query run time and storage size.

A. Theory

Formally, if R is the set of possible results (e.g., key-value pairs for barcharts, or bitmap images of scatterplots), then the $\oplus$ operator used to combine partial results is a binary operator from $R \times R \mapsto R$. We furthermore require that $(R, \oplus)$ be an Abelian group:

Closure	$\forall a, b \in R \Rightarrow a \oplus b \in R$
Associativity	$a \oplus (b \oplus c) = (a \oplus b) \oplus c$
Commutativity	$\forall a, b \in R, a \oplus b = b \oplus a$
Identity element	$\forall a \in R, \exists \mathbf{0} \in R : a \oplus \mathbf{0} = a$
Inverse element	$\forall a \in R, \exists -a \in R : a \oplus -a = \mathbf{0}$

Associativity and commutativity mean that the master node can combine partial results from W workers in any order with $r_t = r_{t-1} \oplus r_{t,1} \oplus \ldots \oplus r_{t,W}$. The existence of inverses means that the master node can also remove a partial result $r_{t,i}$ from r_t if the master decides that worker node i is

[3]Many commercial database vendors prohibit publishing benchmarks in their licensing agreements. In the case of Pavlo et al., the database is known as DBMS-X, "a parallel DBMS from a major relational database vendor."

malfunctioning and the partial result needs to be recomputed, either by a different worker node or by the same worker node.

We further define an operator $\odot : R \times D \mapsto R$ which combines a single data element $d \in D$ with a partial result $r \in R$ to produce a new partial result. This operator is used by the worker nodes to construct their partial results. Each worker node w begins with an empty result $\mathbf{0} \in R$ (e.g., $\mathbf{0}$ could be a blank bitmap image) and combines it with data elements to generate $r_{t,w} = \mathbf{0} \odot d_1 \odot \ldots \odot d_n$.

The $\odot$ operator in VisReduce is analogous to the map operator in MapReduce. However, with MapReduce, the map operator must map to a list of key-value pairs, whereas our $\odot$ operator can map to any object of reasonable size, including key-value pairs and bitmap images.

VisReduce jobs can be implemented by defining the four side effect-free functions listed below. These could be the basis for four methods in a Java interface or an abstract base class in C++, that must be implemented by the programmer.

- a $\oplus$ function that combines two partial results
- a $\odot$ function that combines a partial result with a data element
- an identity function which returns the empty result $\mathbf{0}$
- an inverse function which returns the inverse $-r$ of a partial result r

Those four functions, or methods, can be packaged together to form a work-object, which can then be distributed across a cluster (as a single `.class` file, in the case of Java) for parallel processing.

B. *Implementation*

Our VisReduce prototype has been implemented as a web application using Play[4] for serving the web content and Akka[5] for clustering and actor-based inter-node communication. Each worker node has a copy of the entire data set which is saved on disk in a compressed and bit-packed read-only column oriented format, split in several segments called *tablets*. Upon receiving a request from a client, the master node gathers a list of tablets for a particular table, then sends a list of tablets and a work-object to each worker node. Worker nodes then run the work-object on the tablet, sending the resulting state from the processing to the master node and requesting additional work.

The master node aggregates the results of tablet processing on the cluster as they are received and pushes back the aggregate onto the client at an appropriate speed for the client, which updates the data visualisation shown to the user. To ensure interactivity, the master node can request a partial result from the processing of a tablet by a worker, so that slow tasks running on large tablets still display partial aggregates.

[4]http://www.playframework.org/
[5]http://akka.io/

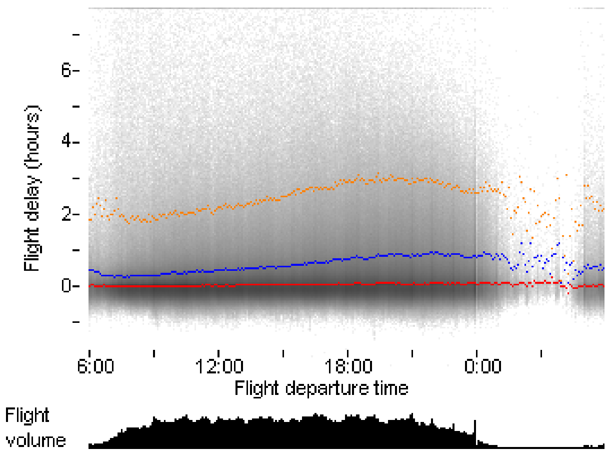


Figure 5. Example of a heat map and accompanying histogram of arrival delay by time of day, with the median, 90th and 99th percentiles of the arrival delay highlighted.

Complete node failures are detected using the φ accrual failure detector [25], while exceptions caught by VisReduce are reported to the master node. Should a node fail during tablet processing with its partial state sent to the client, the partial state can be removed by inverting it and sending the negative delta to the client, while scheduling the execution of the failed tablet on another node, thus ensuring a consistent final state.

C. *Examples*

The bar chart in figure 1 was generated using simple operations for each operator:

$\mathbf{0}$	Initialize an empty associative array of grouping keys to sum and count pairs
$-$	Invert all sums and counts in the associative array
$\oplus$	Combine both associative arrays, removing values with a count of 0
$\odot$	Add or update the value in the associative array

The resulting associative array is then turned into an animated bar chart by the VisReduce client. The heat map of figure 5 was generated using the following operators:

$\mathbf{0}$	Initialize an empty array of bins, each containing a count of 0
$-$	Invert all counts of the array
$\oplus$	Sum both arrays together
$\odot$	Increment the count for the appropriate bin

The bins are then turned into an image and the appropriate percentiles for each time block are highlighted.

IV. Evaluation and Results

VisReduce is designed to ensure that query performance is comparable to other non-incremental systems; it would not be very useful to see incremental results while having to wait significantly longer for any given query, compared to a non-incremental approach.

We evaluate our approach using a dataset of domestic US flights by major carriers from October 1987 until February 2013 and their associated delay information, for a total of 149,598,920 records with 109 columns. The dataset comprises 305 CSV files for a total of 65.74 GB. The size of the data set and its number of records are consistent with the findings of Rowstron et al. [26] for typical analytical workloads on clusters of computers.

The query performance of VisReduce was compared with PostgreSQL, Apache Hive and Cloudera Impala. All systems were benchmarked using queries that spanned one, two and three columns. The one column query counted the number of flights by carrier, which resulted in 32 rows of output across all 149.5 million flights. The two column query calculated simple aggregate statistics (min, max, count and sum) for the arrival delay of flights grouped by carrier, which also resulted in 32 rows of output. Finally, the three column query calculated the same aggregate statistics for the arrival delay but grouped by origin and destination airport pairs, resulting in 8431 pairs of airports and their associated arrival delay information.

Queries were run on a five node cluster used for production analytical workloads at a commercial dating website during times when no other jobs were running. The cluster was comprised of heterogenous nodes equipped with either one or two Intel Xeon processors ranging from 2.13GHz to 2.66GHz, memory sizes ranging between 8 and 16 gigabytes and Western Digital hard drives ranging in size from 160GB to 600GB spinning at 7200 RPM. All nodes ran Debian 6.0 "Squeeze," had Cloudera's CDH 4.2.0 Hadoop distribution installed and used the vendor-supplied patches for Apache Hive 0.10.0 and Apache Hadoop 2.0. We used Impala 1.0, as it was the most recent version available at the time of writing. PostgreSQL was tested using version 9.1.3 on a desktop computer running Windows 7 equipped with a hard drive with a rotational speed of 7200 RPM as well as a laptop computer running PostgreSQL 9.1.9 and Ubuntu 10.04.4 LTS equipped with a 120GB Intel 330 series solid state drive.

The data was loaded directly from the set of CSV files in the case of PostgreSQL, generating a large table with 109 columns. For Apache Hive, the data was loaded from CSV file using CSV SerDe and put into a text table as well as a table encoded in record columnar format, also known as RCFile [27]. As Impala supports an efficient column-oriented file format called Parquet natively, we also copied the data from the text table into its preferred Parquet format. For VisReduce, data was imported from CSV and written as its native columnar format, each tablet being the columnar representation of an input CSV file.

To minimize the effect of disk caching, the operating system's read cache was flushed between runs of Apache Hive, Impala and VisReduce; this was done to ensure that the data for each query was loaded from disk. Neither Impala nor VisReduce keep data resident in memory, but both benefit from the operating system cache in the case where data from a previous run is re-read on the same node. As the PostgreSQL database size was larger than the available memory, this step was not deemed necessary for PostgreSQL. Queries on Impala and VisReduce were run 20 times each, five times for Apache Hive and PostgreSQL on SSD and three times for PostgreSQL on a standard hard drive. As is common for benchmarks on the JVM, queries for VisReduce were ran several times before the actual timing runs as a warm up phase as to avoid JIT compilation during benchmarking.

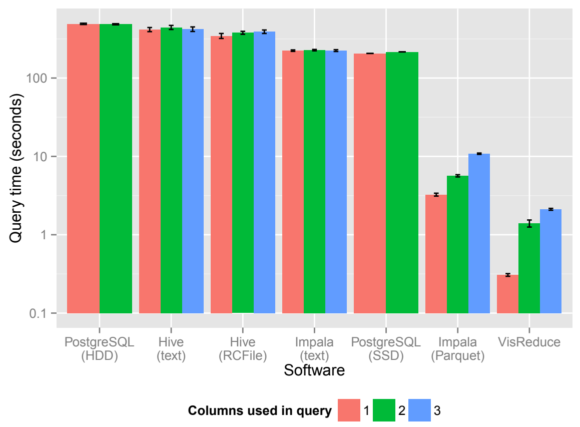


Figure 6. Log_{10} plot of mean query completion time by query type. Error bars represent a 95% confidence interval for the mean. Queries on three columns for PostgreSQL were aborted after twenty minutes.

The performance of VisReduce is significantly better than row-oriented databases, as shown in figure 6. As column-oriented databases only transfer the columns required to answer a query, they have much higher effective I/O utilization than row-oriented approaches for large aggregation-oriented workloads on a subset of columns. Column compression further increases the performance advantage in the case of columns containing highly redundant data, such as the carrier name column in our dataset, which has a very low cardinality compared to the number of records. Finally, as the data in VisReduce is static, there is no need for locks, row versions or other forms of concurrency control, as would be the case in a general purpose database where data can be written at any time, such as PostgreSQL.

In the case of Apache Hive, there is a significant per-job overhead due to the query being translated into a MapReduce job before being deployed onto the cluster. Once the job has been deployed, it requires several MapReduce iterations, with each iteration incurring fixed start up time costs and the need to write to disk between each iteration. While very general — for example, Hive can join arbitrarily sized tables, which neither Impala nor VisReduce can do — there is a significant performance cost to this generality, which makes Hive unsuitable for exploratory visual analytics if the data can be processed by faster systems. At the time of testing, Hive lacked stable support for the more efficient Parquet file format.

As for Cloudera Impala, it is possible that its query planning phase limited its performance relative to VisReduce; the current implementation of VisReduce has no query optimizer and naively processes all tablets across the cluster. Furthermore, Impala has a pluggable storage architecture and supports multiple input formats, while the current implementation of VisReduce only supports its own native column store format.

For the bar charts displayed by the client, our VisReduce implementation attempts to send JSON-formatted data every 250 milliseconds to the browser via WebSocket, which is then turned into an animated chart using JavaScript. We experimented a little with changing this parameter, but it seemed a reasonable compromise between perceived end-user latency and the capabilities of current web browsers to ingest data at a fast rate while animating many SVG elements without any perceived choppiness.

Increasing the rate at which data is sent to the browser also has another unfortunate tradeoff; as the data displayed to the user rapidly converges at the beginning of the computation, showing a visualisation right after the user clicks to start a job means that the user will see a quickly updating and "jumpy" visualisation. Our first iteration on VisReduce simply iterated through tablets in chronological order, which caused the bars in the resulting bar chart to rapidly shift for several seconds as new carriers that did not operate in the year 1987 were introduced into the bar chart and pushed the other bars around while the vertical axis changed its scale to accomodate the fluctuating bars. Shuffling the tablet processing order greatly reduced this shifting. We believe that pre-populating bars with cardinality information gathered from the column store metadata would make the resulting visualisation more aesthetically pleasing, as it would prevent new bars from being introduced.

V. Limitations

As we mostly focused on the technical aspects of computing the underlying data for an information visualisation in an incremental fashion, most of the human interface aspects were ignored. For example, adding error bars as the query processes more information seems like an obvious improvement, which has been explored by Fisher et al. [18], [19].

Another limitation is the fact that programming a VisReduce job is not as simple as writing a SQL query. The endurance of SQL as a query language is a testament to how user friendly and useful it is to answer a wide range of queries. However, in many visualisation systems, such as Tableau [28], SQL is merely an implementation detail that is hidden from the user. We believe that it would be possible to provide built-in VisReduce jobs that compute aggregates in an online fashion and offer a more familiar interface just as Apache Hive provides a SQL-like abstraction on top of MapReduce.

VisReduce is also not as general as MapReduce, which can handle arbitrarily sized outputs and enormous input data sizes that would simply be too large to visualize in an interactive fashion; this is by design. In VisReduce, we trade generality for performance and quick feedback. VisReduce is simply not a good match for batch processing or processing of arbitrary data, just as common implementations of MapReduce are not a good match for interactive processing.

We also do not currently address algorithms that require multiple passes over the input data. For example, it is not possible to compute the standard deviation in a single pass as it requires knowing the mean of the input data, which is unknown at the start of a job. Algorithms that depend on a global ordering, such as computing the exact median of the data, are also impossible to express in a single pass.

VI. Conclusion and Future Directions

VisReduce is a novel approach for interactive visualization of large data sets, that is scalable, distributed, achieves low latency, returns incremental feedback to the user multiple times per second, and was found to be significantly faster than three competing readily available solutions.

The main drawback with VisReduce is that it currently has no way of computing confidence bounds on the partial results it displays to the user over the course of a query. As mentioned in the previous section, adding estimation of error bounds of partial aggregates would be helpful for analysts to determine if they should stop a query or wait for its completion. Jermaine et al. [16] and Joshi et al. [17] suggest ways of doing so on relational databases and, while VisReduce isn't a relational database, similar approaches could be used to provide online estimates of error.

An additional direction for future work would be to modify VisReduce to allow pre-loading all data in memory in the case of smaller data sets, as is done by [29]. Further work is also needed to evaluate VisReduce with much larger data sets and cluster sizes to identify potential performance bottlenecks.

Acknowledgments

The authors wish to thank Hisham Mardam-Bey and Mate1.com for their support.

References

[1] P. C. Wong and R. D. Bergeron, "30 years of multidimensional multivariate visualization," 1997, chapter 1 (pp. 3–33) of Gregory M. Nielson, Hans Hagen, and Heinrich Müller, editors, Scientific Visualization: Overviews, Methodologies, and Techniques, IEEE Computer Society.

[2] U. C. Georges Grinstein, Marjan Trutschl, "High-dimensional visualizations," in *Proc. International Workshop on Visual Data Mining*, 2001, pp. 7–19.

[3] D. A. Keim, "Information visualization and visual data mining," *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, vol. 8, no. 1, pp. 1–8, 2002.

[4] A. Inselberg, "The plane with parallel coordinates," *Visual Computer*, vol. 1, pp. 69–91, 1985.

[5] S. Chaudhuri and U. Dayal, "An overview of data warehousing and olap technology," *ACM Sigmod record*, vol. 26, no. 1, pp. 65–74, 1997.

[6] Z. Liu, B. Jiang, and J. Heer, "imMens: Real-time visual querying of big data," in *Proceedings of EuroVis 2013*, 2013.

[7] J. Dean, S. Ghemawat, and G. Inc, "MapReduce: simplified data processing on large clusters," in *In OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*. USENIX Association, 2004.

[8] J. Heer and B. Shneiderman, "Interactive dynamics for visual analysis," *Queue*, vol. 10, no. 2, pp. 30:30–30:55, Feb. 2012. [Online]. Available: http://doi.acm.org/10.1145/2133416.2146416

[9] C. Johnson, "Top scientific visualization research problems," *IEEE Computer Graphics and Applications (CG&A)*, vol. 24, 2004.

[10] C. Chen, "Top 10 unsolved information visualization problems," *Computer Graphics and Applications, IEEE*, vol. 25, no. 4, pp. 12 – 16, July-Aug. 2005.

[11] J. D. Mackinlay, P. Hanrahan, and C. Stolte, "Show me: Automatic presentation for visual analysis," *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, vol. 13, no. 6, pp. 1137–1144, 2007.

[12] B. Shneiderman, "Dynamic queries for visual information seeking," *Software, IEEE*, vol. 11, no. 6, pp. 70–77, 1994.

[13] T. Munzner, F. Guimbretière, S. Tasiran, L. Zhang, and Y. Zhou, "TreeJuxtaposer: scalable tree comparison using focus+context with guaranteed visibility," *ACM Transactions on Graphics (TOG)*, vol. 22, no. 3, pp. 453–462, 2003.

[14] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: interactive analysis of web-scale datasets," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 330–339, Sep. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1920841.1920886

[15] J. M. Hellerstein, P. J. Haas, and H. J. Wang, "Online aggregation," in *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '97. New York, NY, USA: ACM, 1997, pp. 171–182. [Online]. Available: http://doi.acm.org/10.1145/253260.253291

[16] C. Jermaine, A. Dobra, S. Arumugam, S. Joshi, and A. Pol, "The Sort-Merge-Shrink Join," *ACM Transactions on Database Systems (TODS)*, vol. 31, no. 4, pp. 1382–1416, 2006.

[17] S. Joshi and C. Jermaine, "Materialized sample views for database approximation," *IEEE Transactions on Knowledge and Data Engineering*, vol. 20, no. 3, pp. 337–351, 2008.

[18] D. Fisher, "Incremental, approximate database queries and uncertainty for exploratory visualization," in *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, Oct. 2011, pp. 73 –80.

[19] D. Fisher, I. Igor Popov, S. Drucker, and M. Schraefel, "Trust me, I'm partially right: Incremental visualization lets analysts explore large datasets faster," *CHI 2012*, May 2012.

[20] J.-H. Böse, A. Andrzejak, and M. Högqvist, "Beyond online aggregation: parallel and incremental data mining with online Map-Reduce," in *Proceedings of the 2010 Workshop on Massive Data Analytics on the Cloud*, ser. MDAC '10. New York, NY, USA: ACM, 2010, pp. 3:1–3:6. [Online]. Available: http://doi.acm.org/10.1145/1779599.1779602

[21] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce online," in *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, 2010, pp. 21–21.

[22] J.-F. Im, M. J. McGuffin, and R. Leung, "GPLOM: The generalized plot matrix for visualizing multidimensional multivariate data (in press)," *IEEE Transactions on Visualization and Computer Graphics (TVCG) (Proceedings of InfoVis)*, vol. 19, no. 12, 2013.

[23] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *Proceedings of the 35th SIGMOD international conference on Management of data*. ACM, 2009, pp. 165–178.

[24] D. Abadi, S. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM, 2006, pp. 671–682.

[25] N. Hayashibara, X. Defago, R. Yared, and T. Katayama, "The φ accrual failure detector," in *Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on*. IEEE, 2004, pp. 66–78.

[26] A. Rowstron, D. Narayanan, A. Donnelly, G. O'Shea, and A. Douglas, "Nobody ever got fired for using Hadoop on a cluster," in *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing*. ACM, 2012, p. 2.

[27] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu, "RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems," in *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. IEEE, 2011, pp. 1199–1208.

[28] C. Stolte, D. Tang, and P. Hanrahan, "Polaris: A system for query, analysis, and visualization of multidimensional relational databases," *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, vol. 8, no. 1, pp. 52–65, 2002.

[29] S. Shenker, I. Stoica, M. Zaharia, R. Xin, J. Rosen, and M. J. Franklin, "Shark: SQL and rich analytics at scale," 2012.