Maintaining Interactivity While Exploring Massive Time Series

Sye-Min Chan* Ling Xiao† John Gerth‡ Pat Hanrahan§

Stanford University

ABSTRACT

The speed of data retrieval qualitatively affects how analysts visually explore and analyze their data. To ensure smooth interactions in massive time series datasets, one needs to address the challenges of computing *ad hoc* queries, distributing query load, and hiding system latency. In this paper, we present ATLAS, a visualization tool for temporal data that addresses these issues using a combination of high performance database technology, predictive caching, and level of detail management. We demonstrate ATLAS using commodity hardware on a network traffic dataset of more than a billion records.

Index Terms: D.2.11 [Software Engineering]: Software Architectures—Domain-specific architectures; H.5.2 [Information Interfaces And Presentation]: User Interface—Graphical user interfaces (GUI); K.4.0 [Information Systems Applications]: General

1 INTRODUCTION

There has long been interest in applying visual analytics to temporal data [6] [17] because visualization combined with interaction augments the human's cognitive process leading analysts to discoveries in complex data [27]. Rapidly falling storage costs now permit collection of massive datasets but supporting interactive visual analysis for these datasets is difficult. There are three main challenges. First, analysts do not merely want to view raw data; they need to explore it using *ad hoc* filters, aggregations, and trending. As dataset size grows, the cost of computing these queries also increases. Second, as datasets become too large for the analyst's machine, housing data on remote servers creates problems in load balancing the distributed queries. Third, fetching data from remote database servers is likely to introduce latencies that affect the responsiveness of the visualization system and disrupt analysis.

In this paper we address these challenges by applying technologies from database systems and computer graphics to the field of visual analytics. To support fast query over massive datasets, researchers have developed new methods to more efficiently partition, store, and index data. In the field of computer graphics, real-time rendering systems such as flight simulators use level of detail (LOD) management and pre-fetching techniques to smoothly display large amounts of data at multiple resolutions. Inspired by these approaches, we have created ATLAS, an interactive visualization tool for large temporal databases running on commodity hardware.

2 RELATED WORK

2.1 Visualization

The visualization of time series dates back to as early as the tenth century when the inclinations of the planetary orbits were plotted with lines as a function of time [28]. Since then, numerous visual metaphors have been developed to emphasize specific properties of time series such as periodicity [7]. In recent years, researchers have focused on using interactive visualization to support analytical tasks such as pattern searching [6], motif discovery and anomaly detection [17]. While these applications were built using datasets that fit in system memory, there has been relatively little work on supporting interactive visual exploration over large temporal databases.

Network traffic analysts routinely deal with huge volumes of temporal data which is difficult to visualize *en toto*. In VIAssist, Tesone et al. [26] attack the problem of visual complexity through "smart aggregation" which performs data aggregation either automatically when the cardinality ratio of a field exceeds the threshold or according to user-defined controls. Smart aggregation ensures that the amount of data retrieved will be manageable for visual analysis. ATLAS limits visual complexity by using window displaying a fixed number of time series. However, it must also deal with some kindred aggregation issues, especially when the analyst zooms out to coarse time intervals. Nuance [18] is a system that creates models of the expected behavior of the time series for thousands of monitored systems and displays current data against the model. It handles large datasets in a streaming environment whereas ATLAS is built to directly explore large historical data.

The importance of optimizing data access and filtering times to support fluid visual analysis was recognized by Bethel et al. [2]. Their system used compressed bitmap indexing of flat files to speed up queries, and demonstrated support for interactive *ad hoc* and multi-resolution query on a supercomputer platform optimized for data intensive analysis and visualization tasks. ATLAS aims for similar functionality but uses a column-oriented database and a network of servers running on commodity hardware.

Doshi et al. addressed the problem of visually exploring large datasets by prefetching [12] and discussed several strategies such as Random, Direction, and Focus. In [11], they proposed a strategy selection framework that adapts to user interactions. ATLAS uses a prediction strategy that is similar to their direction strategy, but instead of sending queries only when the system is idle, it predicts when queries need to be sent in order to maintain interactivity.

2.2 Database Systems

Most relational database management systems (RDBMS) are row-oriented, meaning attributes of a record are contiguous in storage, and optimized for throughput performance in on-line transaction processing applications such as order entry and banking. For on-line analytical processing applications, where query throughput is more important, data cubes can be used to accelerate common queries in a large data warehouse [8] by pre-computing aggregations of measures over a set of dimensions. However, with large datasets pre-computed views need to be planned with care [14] because of the large number of possible views.

Column-oriented storage is better for many analytic queries [21] [22] because only the attribute data referenced in the queries are fetched. Equivalent queries can also be computed in row-oriented databases, but their implementations are complicated [20]. Harizopoulos et al. [15] explored the tradeoffs between row and column-oriented architectures in terms of disk bandwidth and CPU performance, and concluded that column stores can almost always achieve better query performance.

In part because of FORTRAN, column storage has been widely used for decades in scientific computing. In current scientific data management the ROOT [5] system from CERN provides an object-oriented store with attribute columns in flat files. In database ap-

*e-mail:sychan@stanford.edu

†e-mail:lingxiao@stanford.edu

‡e-mail:gerth@cs.stanford.edu

§e-mail:hanrahan@cs.stanford.edu

IEEE Symposium on Visual Analytics Science and Technology
October 21 - 23, Columbus, Ohio, USA
978-1-4244-2935-6/08/$25.00 ©2008 IEEE

plications, C-Store [23], MonetDB [4], and the MonetDB/X100 extension [16] are examples of open source column-oriented systems. The last has shown excellent performance in queries over large datasets such as TREC Terabyte Track [9]. Both compress data columns in order to improve effective disk bandwith and employ lazy decompression for better processor cache utilization.

Tesone's VIAssist system uses Sybase IQ [24] , a commercial column-oriented database with specialized indexes to improve performance for low cardinality, grouped, or range data. ATLAS employs kdb+ [29] , a commercial system optimized for distributed time series analysis.

2.3 Computer Graphics

Texture mapping is a process which adds surface details by mapping an image to the surface of a computer generated model. Since the observable resolution of a texture varies with viewing angle and distance, computer graphics systems often store a texture at multiple levels of detail. The Mipmap technique [30] does this with a pyramid of images at different resolutions. Flight simulator systems, that must render large number of textures in real-time, extend this idea with the Clipmap [25] which treats the whole texture dataset as a mipmap to fetch only the portions that are visible in the clipped view. To hide system latency due to data access, data is pre-fetched [3] along the flight path. Pre-fetching may be done by biasing mip-level computations to retrieve higher resolutions maps earlier; or to predict the textures that will be needed by tracking the change in the viewpoint.

The task of visualizing a large number of long time series is analogous to the rendering of large terrains except pixels are mapped to temporal space rather than geography. ATLAS uses techniques similar to level of detail and pre-fetching to support smooth navigation in the temporal space. However, unlike graphics systems where textures can be pre-computed, the exploration of temporal database involves *ad hoc* queries and filters that must be computed at run-time.

3 System Goals

We aim to develop a system that allows interactive exploration over large temporal databases. Our design goals for ATLAS are as follows:

1. ***Ad hoc* querying:** In order to support exploratory analysis, it is important to allow analysts to ask *ad hoc* questions about the data. Our goal is to allow analysts to calculate arbitrary aggregates formed by attributes in the database and to apply any filters to the data. This flexibility raises two issues. First, analysts might formulate *ad hoc* queries that are expensive to compute, especially over large databases. Second, permitting filters with queries largely precludes the use of pre-computed aggregates, forcing them to be computed directly from the underlying data for each query.

2. **Load balancing:** As the amount of data analyzed increases, it is essential for the system to be able to expand processing capacity. A common way to handle this problem is by increasing the number of database servers. However, this only works if the system can distribute query load efficiently over the set of servers.

3. **Smooth Interaction:** Exploratory analysis is greatly enhanced by smooth interactions. Specifically, fluid behavior for bread and butter operations such as panning and zooming gives analysts the impression of "flying" through the database. With large databases, this is difficult to achieve due to system latencies caused by query computation and data transfer. Supporting zooming at interactive speed is particularly difficult, since the number of records which must be scanned by the system increases exponentially as the analyst zooms out to examine a longer time period. To address this problem some systems pre-fetch data which creates a new set of questions about what data to fetch and when to fetch it.

ATLAS is composed of three parts, a database cluster, a query distribution server, and a visual interface. In Section 4, we will first give an overview of the architecture of ATLAS, and then describe each of its components in detail. In Section 5, we will evaluate the performance of ATLAS according to the three design goals.

4 System

ATLAS uses a client-server architecture in which the server is a database cluster, and the client consists of a visual interface and a query distribution server. The separation of database server from the client abstracts the details of data storage from the client and allows data to be distributed over any number of database servers. The division of client into the visual interface and the query distribution server partitions the user interface and the caching of data. This allows visualizations and interactions to be conducted smoothly while data are being fetched. It also leverages modern multi-core systems by multiprocessing the visual interface and the query distribution server on separate processors.

The interactions among different components of the system are shown in Figure 1 with green arrows denoting control flow and blue arrows showing data flow. During exploration, the analyst interactively pans, zooms, filters, and groups time series through the visual interface. Based on user interactions, the visual interface predicts data requirements, sending requests to the query distribution server via Java remote method invocation. Upon receiving the request, the query distribution server divides the job according to load and dispatches the queries to the cluster of database servers. As the query results return, the query distribution server writes the data into memory mapped files which are polled by the visual interface as it creates visualizations.

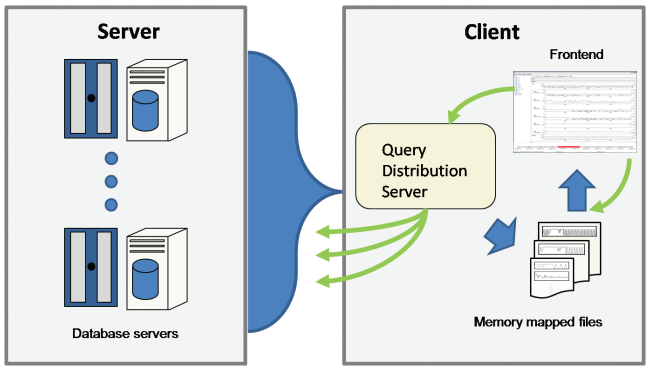

Figure 1: The client-server architecture of ATLAS.

We will use two datasets as examples to discuss the design and performance of ATLAS. The first contains network flows collected over two months in the Computer Science and Electrical Engineering Departments at Stanford University. The database has 1.28 billion rows of data with 25 columns of attributes, which include timestamps, durations, protocols, local and remote ports and IP addresses, number and size of packets etc. taking up more than 100 GB on disk. Data for each date is stored by kdb+ in a separate directory with each attribute column stored in a file sorted by time and indexed by multiple attributes. The second is a financial dataset obtained from the Centre of Research in Security Prices (CRSP) [10] which consists of daily stock prices of over 25,000 companies between 1925 and 2006. The database has roughly 73 million rows of 20 columns consuming 10GB of disk. Its attributes include the CRSP permanent company numbers, exchange ticker symbols, the

standard industrial classification codes, open and close prices, and share volume etc. Data are first grouped by the permanent company number, and then sorted by date. Various fields such as ticker symbol are indexed.

All timing experiments and performance tests were done on a compute cluster using 8 rack-mounted SunFire V40z computer servers, each with 4 AMD Opteron 852 2.6Ghz processors, 32 GB of PC3200 DDR RAM, and a 300GB 10K RPM SCSI U320 disk.

4.1 Database System

Interactive visual analysis over any large dataset is impossible without an efficient data management system because accessing and summarizing the data prior to mapping it for visualization is the gating factor in system performance. The cost of moving data through the multi-tiered storage hierarchies of today's systems is so high that it is critical for applications to try and fetch only needed data. In situations where analysts can predict interesting aggregations in advance, datacube techniques are effective performance accelerators. However, one of our goals is to allow analysts to specify *ad hoc* aggregations and filters over the data; either of which renders precomputation ineffective and thus places a premium on being able to perform runtime processing on servers. Finally, time series data are intrinsically ordered and traditional analysis in this field includes algorithms such as the exponentially weighted moving average which are dependent on ordered data. In ATLAS data is stored in kdb+, a database designed specifically to support time series data, which has three major attributes important to achieving our goals: (a) column-oriented storage; (b) partitioning and indexing schemes for ordered data; and (c) a networkable query engine with a programming language.

Column-oriented storage in kdb+ improves query times by improving locality in several ways. First, since many *ad hoc* queries touch just a few columns, only the data associated with them will be moved through the storage hierarchy thus avoiding the expense of bringing in data which will lie unreferenced. Second, when a column is being scanned as part of a filter, its data will lie contiguously making access sequential whereas row-oriented data would be accessed with potentially large stride factors. Such dense locality improves speed from the processor cache out to the page tables. Third, kdb+ accesses data using memory-mapped files wherein a disk file becomes a piece of the application address space. While the entire file contents are logically available, only those portions which are referenced are transferred from disk. Transfer occurs through the highly efficient OS page-fault process. Furthermore, files are implicitly shared for reading across all processes mapping the same file. Thus, the combination of column-oriented storage and mapped files creates granular, shareable, flexible, hardware-assisted access to data under the simple programming model of vector indexing.

The second design element involves partitioning and indexing schemes for ordering data. Stock market data is often partitioned by its ticker symbol because analysts want to compare the performance of specific groups of stocks. Network flow data is more naturally partitioned by date because the large number of network addresses; network devices tend to create traffic every day; and the problems of interest usually involve contemporaneous behavior. While kdb+ stores each column as a separate flat file, it also allows one to partition data according to the values of one column. Temporal partitions can be organized in days, months, or years. Each partition becomes a directory in the filesystem and the directory's name becomes a virtual column in what is logically a single table. The columns of data within each partition can also ordered by sorting on one or more of them. Any column may be indexed in which case a hash table is appended to its data file. Network data is often sorted by time then indexed by its categorical dimensions. Queries of a sorted column can be processed very efficiently with a binary search. Having the hash index appended to the data column means not only that it won't be touched if not in referenced, but also that it will be logically nearby when needed.

The final important kdb+ element is its networked query engine and the *q* query language which has analytic functions for time series analysis such as moving average and grouping functions including ones that expressly bin time values across ar rich set of date and time granularities. Applications can connect via TCP sockets to the server natively, from Java, via ODBC, or as a webserver. Queries may be submitted as SQL92 or in native *q* with results returned in tables, CSV, HTML, or XML.

4.2 Query Distribution Server

The query distribution server runs on the client computer. It is responsible for querying the database cluster and serving the data to the visualization system. The design goal of this component is to minimize both the query time from the database cluster and the transfer time to the visualization system.

The query distribution server fetches data from the database cluster by distributing queries over the servers according to a load balancing algorithm that tries to equalize the workload among them. We determine the workload by the total number of records that needs to be scanned in a query. Depending on the organization and size of the dataset along with the analysis characteristics, the data can either be partitioned by time, or by the series. For example, in the network dataset, we have chosen to partition the load according to ranges of days because data are stored in the resolution of milliseconds, giving orders of magnitude more data points in each series than there are series. In addition, we have found that analysts tend to concentrate on a few suspicious machines but would look at their data over very large time ranges to extract historical trends. To partition the time ranges so that the workload is distributed evenly, the system keeps track of the total number of data points per day for the current set of time series, and estimates workload by computing the total number of data points in the queried time range.

The query distribution server transfers data to the visualization system by creating memory mapped files shared between the two components. Using memory mapped files the query distribution server can run as a completely separate process from the visualization system which allows modern operating systems to more efficiently delegate resources. However, because there is no data duplication, effective disk transfer rates are the same as having the query distribution server running in the memory space of the visualization system. Since the network and financial data has orders of magnitude more records for each series relative to the number of series, ATLAS creates a separate memory mapped file for each series at each aggregation level. To control disk space consumption, the files are implemented as circular buffers in time.

4.3 Visual Interface

The ATLAS visual interface, as shown in figure 2, supports scrolling, panning, zooming, ordering, filtering, and grouping time series displayed as line graphs or bar charts. ATLAS overcomes the latencies associated with querying large datasets by monitoring the analyst's actions and predictively caching the data that will be needed to render subsequent frames. How the predictions are made will be described later. Here, we note that zoom out is the most difficult operation to support, since it entails an exponential increase in the number of records analyzed. This creates two difficulties: first, there may be too much data to be transferred to the local client; and second, there are not enough pixels at current resolutions to show individual records. ATLAS overcomes these difficulties by coupling zooming with level of detail (LOD). When the analyst zooms out, we adjust the visual distance between consecutive data points until a threshold minimum inter-point distance is reached. To zoom out further, we issue new queries to the servers to re-bin the records using familiar temporal intervals such as minutes, hours, and days. In the network dataset the finest level of detail is milliseconds. When zooming out, we aggregate records in intervals

of 1 minute, 10 minutes, 30 minutes, etc. Zooming in requires the inverse of the above operations with a maximum threshold on the visual distance. This approach has the advantage that we can delegate the computation to scalable database clusters, and we transfer the minimal amount of data to the client.

For visualization systems where all data is resident, panning and random access are trivial modifications to the window boundaries. Once data exceeds memory, disk access at speeds far slower than interaction demands must occur. ATLAS hides the disk latency during panning by placing a ceiling on the maximum panning speed. This ceiling is determined by analysis requirements and depends on the size of the dataset and the performance of the database cluster. For example, network analysis using four cluster machines resulted in a ceiling of 600 pixels per second. ATLAS does not support smooth interaction for temporal random-access, instead an analyst using ATLAS zooms out from his current temporal location, and then zooms in to the desired location, similar to Pad++ [1].

Large datasets often contain thousands of time series; however, the number an analyst can simultaneously examine is limited by the vertical resolution of the display. ATLAS supports vertical scrolling by fetching only the series that are currently visible or predicted to be visible soon in fashion similar to panning. The vertical ordering of series is determined by the analyst's choice of an ordering attribute. The analyst can change the ordering to juxtapose related series and thus affect the focal series and future predictions. For example, the series in the network data can be ordered according to local IP, remote IP, local Port, etc. ATLAS helps the analyst maintain visual context during vertical scrolling with two panels, one above and one below the main display area showing as an overlaid time series those that are immediately above or below the focal series. This approach is similar to fisheye views [13]. ATLAS can also show grouped series overlaid as one plot. This can be used reveal trends in related series, e.g. the stock price for all auto manufacturers. Since a group of series might have many members, ATLAS only shows the top five series in a single group as determined by the analyst-selected ordering attribute for the group.

ATLAS is designed to hide the query latency associated with analysis. Ideally, visual analysis tasks can always be performed on data that is cached. In reality, it is possible that the data is not ready when it is needed. For example, the database could be unexpectedly slow because of expensive queries from other analysts. In such cases, ATLAS does not stop and wait, instead, it highlights the ranges of any unavailable series in red and allows the analyst to continue his analysis on the data that is available. Meanwhile, the front-end polls the mapped files for new ranges or series and displays them as they arrive.

4.4 Predictive Caching

ATLAS hides the latency associated with querying large datasets by predicting and preemptively caching required data. The predictive algorithm is based on observing that there is a sense of momentum associated with the direction of exploration - e.g. an analyst panning to the left at time t is likely to continue panning left at time $t+1$. As seen in figure 3, ATLAS has 6 directions of exploration (pan left, pan right, scroll up, scroll down, zoom in, and zoom out) with different interaction triggers fetching in different dimensions– panning right (left) leads to the fetching of the next (previous) time window for the same set of series that are currently visible, scrolling down (up) leads to the fetching of the same time window for the set of time series that will become visible next, and zooming in (out) leads to the fetching of a narrower (wider) time window for the same set of series.

Once the fetch direction is determined, the system needs to decide when to issue the query, and what data to fetch. These decisions are governed by a set of system parameters and constraints, which will be discussed in section 4.4.1. We create a timing model for each dataset as a preprocessing step to estimate query processing time. These models will be discussed in section 4.4.2.

Table 1: Variables

Action	Variables
Initialization	• Size of cache file (S_{cache}) • Number of series to be fetched (N_{init}) • Time range to be fetched (T_{init})
Panning/ Scrolling/ Zooming	• Number of series to be fetched ($N_{pan/scroll/zoom}$) • Time range to be fetched ($T_{pan/scroll/zoom}$) • Time at which the pre-fetch query is issued in terms of the amount of time before the end of cache is reached ($t_{pan/scroll/zoom}$)

Table 2: Fixed variables and Constraints

Constants
• Maximum panning speed (V_{pan}) • Maximum scrolling speed (V_{scroll}) • Window width (W_w) • Window height (H_w)
Functions
• NumDataPoint(T) = number of data points in time range T • TimeRange(W)= time range spanned by width W • NumSeries (L)= number of time series visible in length L • TimeToFetch (T, N)= the amount of time taken to fetch time range T for N series
Constraints
• $S_{cache} \geq NumDataPoint(T_{init} + T_{pan})$ • $N_{init} = N_{pan} = N_{zoom} = NumSeries(H_w) + 2 \times NumSeries(t_{scroll}/V_{scroll})$ • $T_{init} = T_{scroll} = T_{zoom} = TimeRange(W_w) + 2 \times TimeRange(t_{pan}/V_{pan})$ • $t_{pan/scroll/zoom} > TimeToFetch(T_{pan/scroll/zoom}, N_{pan/scroll/zoom})$ • $T_{pan} \geq TimeRange(2 \times W_w)$ • $N_{scroll} \geq 3$

4.4.1 Constraints

A major question to answer in predictive caching is "what to fetch". A query consists of various variables: the aggregates, the aggregation level, the filters, the set of series, and the time range. The first three are defined by users, while the last two are determined by the system according to the current time window, the set of visible series, the user interactions, and the parameters governing the number of series and the time range to pre-fetch. To determine when queries should be issued, the system needs to take into consideration how long the query would take to complete, as estimated by the timing model, and when the data is needed, i.e. how long it takes to reach the end of the currently cached data under the maximum panning, scrolling, or zooming speed. Table 1 shows the list of system parameters.

The values of the system parameters are chosen to ensure that data will always be retrieved in time to support smooth interactions. We formalize this with a set of constraints that need to be satisfied by the parameters (Table 2). For example, the cache size needs to be big enough so that newly fetched data will not overwrite the data used in the current frame. The number of series fetched during panning and zooming has to be large enough so that when the analyst decides to start scrolling, there will be enough time to fetch the next set of time series that become visible. Similarly, the time range queried during scrolling and zooming should accommodate a sudden panning action. In Table 2, the last two constraints ensure that queries are not issued too frequently. The solution space of the constraints is not a unique point, but a region that contains differing tradeoffs. For ATLAS we prefer a solution that queries data just-in-time. This minimizes the load on the client machine, the load on the database cluster, and the overhead associated with the analyst changing directions during panning / zooming / and scrolling. Our objective is not to pull the maximum amount of data possible, but rather to get the minimum amount of data to sustain smooth interactions. If the query speed were instantaneous, we would only

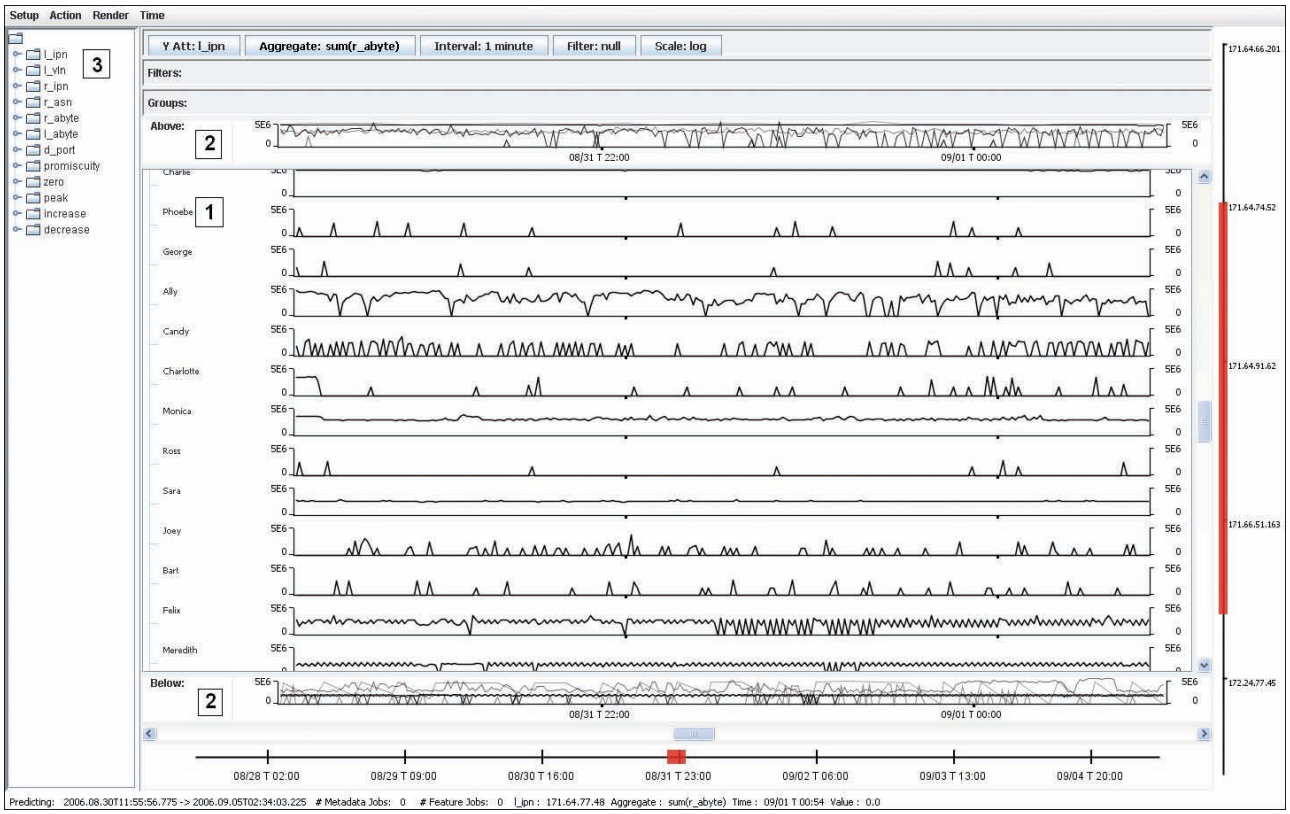

Figure 2: Fontend interface of ATLAS. 1: The main panel where time series in focus are plotted as line graphs. 2: Context panels showing the time series outside the focal point. 3: List of attributes that can be used for grouping and filtering.

need to fetch data in the current frame. On the other hand, if the query speed is extremely slow, everything would need to be pre-computed and stored on the client machine. ATLAS will function with any solution in the solution space with the tradeoff between client and server load adjusted depending on available resource.

To show how ATLAS uses the parameters to support smooth interactions, consider the scenario where an analyst pans to the right. When the motion is detected by the scrollbar, the system will check the cached files to determine when new data will be needed. If the minimum time taken to pan to the end of the cached data is smaller than t_{pan}, a query is issued for the N_{pan} series that are closest to the visible window over T_{pan} after the last time point cached. Since t_{pan} is greater than the estimated time taken for the query to complete, we expect the data to arrive before they are needed for visualization.

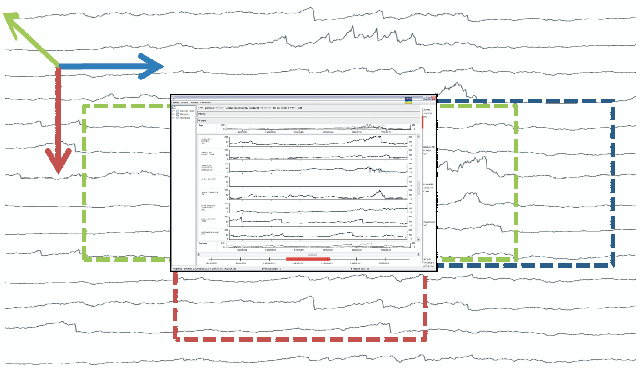

Figure 3: The three dimensions for caching, as defined by panning, scrolling, and zooming.

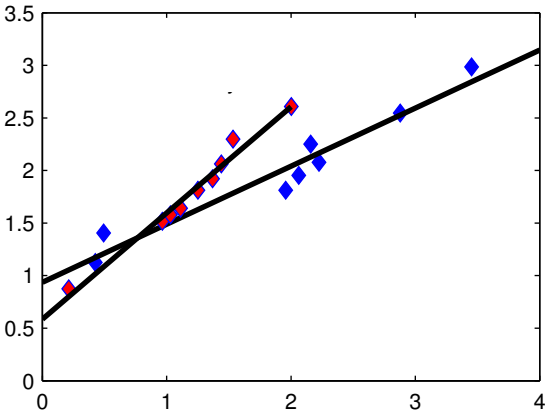

Figure 4: The processing times for queries that span 5 and 10 days in the network database.

4.4.2 Timing Model

The timing models are used to estimate the time required for a query to complete. They are specific to each database because of the differences in size and organization of data; however, they are all based on the premise that query processing times increase linearly with the number of records contained in the query range. This is a reasonable choice since most query times are dominated by disk and memory access, not computation. As the number of records depends on the number of time series to be fetched as well as the time range spanned by the query, we measure the change in query processing times as these two variables change. During this prepro-

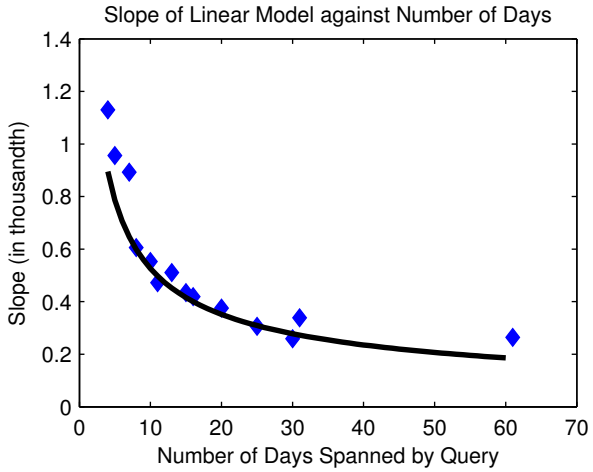

Figure 5: Change in slope of linear model in the network database.

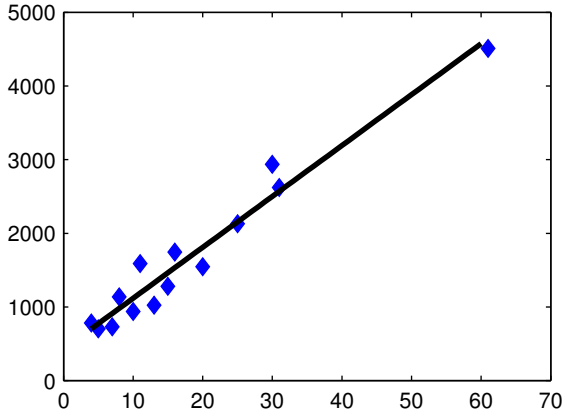

Figure 6: Change in intercept of linear model in the network database.

cessing step we also vary the number of database servers and use a load balancing algorithm to simplify the model by measuring the number of records read and the amount of time taken for the server with the most work.

For the stock price database we observed that changing the number of series and the time range of the query have the same effect on the query processing time. We therefore concluded that the two variables are redundant and model the query time using the number of records as the only variable. The timing model was created by simply running linear regression on the number of records contained in a query range and the response time.

Query speed over the network flow database was highly dependent on the number of days spanned by the query because the network dataset is partitioned by date. However, we did observe a linear relationship between the query time and the number of records fetched when when the number of days spanned is held constant. Figure 4 shows the change in response times for queries that span 5 and 10 days as the number of records increases and their linear regression lines. Figure 5 and 6 show the changes in the slope and the intercept of the regression lines as the number of days spanned by the query changes. As shown, the slope decreases exponentially while the intercept increases linearly with the number of days. Therefore, to estimate the time it takes for a given query to complete, the number of days spanned by the query is first used to estimate the parameters of a linear model. The resultant linear model is then used to compute the query time given the number of records.

5 Performance

In this section, we evaluate the performance of the main features of ATLAS: support of *ad hoc* queries, load balancing, and smooth interaction. Examples of ATLAS in use are shown in the accompanying video, in which we demonstrate system interactivity and perform a case study of network intrusion detection. In the case study, we follow an analytical process inspired by US CERT analysts [19] but ATLAS allows the analysis to proceed without having to create the dramatically reduced sample that they needed to extract because of limited data capacity of their tools.

In ATLAS, data are stored in a column-oriented database system designed and optimized for temporal queries. In Section 5.1, we will briefly compare the performance of this to a row-oriented database system to justify our design choice.

As the amount of data being collected and analyzed increases, it is important for systems to expand their analytic capacity. In Section 5.2, we will evaluate our load balancing algorithm by assessing its performance as more database servers are added.

ATLAS supports smooth interactive explorations via predictive caching. In Section 5.3, we will measure the performance gained by the use of predictive caching.

5.1 Database Performance

The query performance of the database server system is critical if ATLAS is to meet its design goals. To demonstrate that the column-oriented kdb+ 2.4 system used by ATLAS is a good choice, we compared it to MySQL 5.0.22, a widely-used, row-oriented RDBMS with a good reputation for performance in read-only queries.

The queries were run against one of the SunFire servers with the 1.28 billion row dataset of network flows stored on the local disk. Each row consists of 25 columns of attributes. As is customary with this type of data, it is organized in tables by date. For both kdb+ and MySQL, the overall database size including indices is 128GB. MySQL stores the table data (82 GB) separate from the indices (46 GB). Performance measures of the raw disk read on the 64-bit Linux 2.6 system showed that it could sustain data transfer at a rate of 59 MB/sec when the read data was being discarded.

Scanning Queries Scanning queries occur when indices cannot be used or when the analyst is zoomed out so far that the entire time span of the dataset must be summarized. In our first test, we measured the performance of such a scanning query involving summing one column at initialization when no data has yet been transferred from disk. MySQL took 2134 seconds whereas kdb+ finished in 113 yielding a speedup of 19 for kdb+. This is near the ratio of the byte width of a full row in MySQL to the byte width of the selected column. kdb+ read data at 45 MB/s from disk to MySQL's 40 MB/s indicating efficient use of filesystem operations by both.

We then reissued the same query to measure the effect of the operating system's file cache. In this test, MySQL took 2074 seconds while kdb+ finished in 28 for a speedup factor of 74 for kdb+. The change is probably due to the fact that the 5GB of data in the 60 files comprising the single column kdb+ reads can be cached in the RAM of the server whereas MySQL's retraversal of 82GB of data overwhelms it.

Simple Indexed + Partitioned Queries Next, we sought to understand the performance of queries which look at indexed data in restricted date ranges since experienced analysts will make sure that important attributes are indexed and for the most likely queries the date range will be limited to days rather than months. For these tests, we again summed a single column, but now only interdomain traffic grouped hourly by network address spanning a few days to a week. When selecting two relatively quiet addresses over a four day range, the kdb+ advantage dropped to only a factor of 3 over MySQL (11.95 versus 3.98 secs). However, adding a third active

address exceeded the configured 8G limit for MySQL's temporary RAM table resulting in a disastrous 2709 sec query versus 6.7 sec for kdb+. The problem here is that while kdb+ will take advantage of date partitioned tables, MySQL has only the limited functionality of the MERGE table (essentially a listed of concatenated tables) requiring the application to create temporary MERGE table definitions for the specific date ranges of the query. Thus, while kdb+ indexing may be less sophisticated, it is more predictable and arguably more effective.

Other Queries There are also analytic queries which can be made in *q* which are difficult or inexpressible in SQL. For example, the "moving average" or "weighted moving average" queries, which are common in time series analysis and supported by *q*, are typically not performed in a relational system like MySQL because they required ordering semantics which cannot be expressed directly in the language. This is not to say that the functions cannot be provided, rather that they would have to be explicitly coded in the ATLAS client or elsewhere.

5.2 Load balancing

ATLAS uses a client-server architecture to offload analytic query processing from the analyst's workstation so that as the size of datasets grow, more database servers can be added to maintain or improve response time. To manage query processing the ATLAS client contains a query distribution server that aims at dividing workload evenly among the available database servers. This step is essential in optimizing the performance gain when adding servers. To evaluate the quality of the load balancing algorithm, we measured query performance while varying the number of database servers used.

In this experiment, the network flow dataset was used because of its larger size. We chose from the database the 120 network addresses with the largest overall number or flows and computed the sum of application bytes by hour over 60 days. This calculation traverses about half of all rows (600 million records) and so represents a severe test of processing capacity. We recorded query time as we increased the number of database servers. The time consists of the elapsed time for the query, excluding the time for writing the data to cache files or for displaying the data since that time is negligible compared to query computation time.

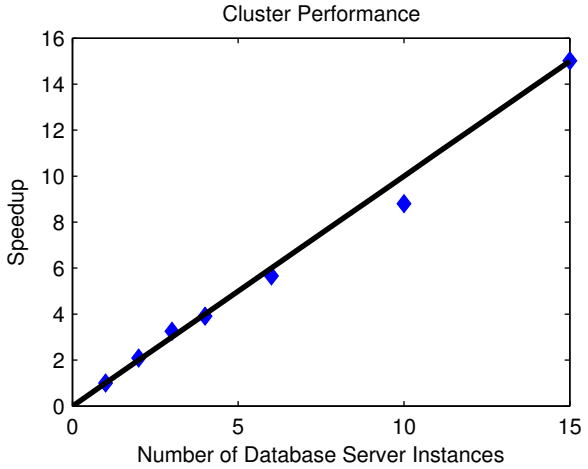

Figure 7: Speedup of query with the number of database servers

As shown in Figure 7, the speedup, as calculated by the proportional gain in query throughput when compared to the base case where only one database server was used, increased linearly with the number of database servers. The trend was consistent as the number of database servers increased to 15. As we further increased the number of servers, we observed a decrease in performance gain. This is due to the current load balancing algorithm which can only divide queries by date and time bounds. Since data are partitioned by date in the network database, it is more efficient to divide queries by time than by lists of series. However, a combined approach that divides query both by time and the list of series might prove beneficial as we further develop the system to accommodate datasets of different size and organization.

5.3 Predictive Caching

Predictive caching is necessary to support fluid exploration over large databases. To demonstrate the performance gain, we compared the smoothness of panning in systems with and without predictive caching.

In this experiment, we measured the time taken to pan over two months of data for 15 network addresses at an aggregation level of 10 minutes. The maximum panning speed was 600 pixels per second. Each frame was about 900 pixels wide and spanned approximately 1.6 days. The selected set of network addresses generated about 2 million flows per day. Thus, each frame requires the traversal of approximately 3.2 million records and a data rate of 2.1 million records/sec is needed to support panning at the maximum speed. With predictive caching, the system fetched 15 days of data as an initialization step, and when it was 2.5 frames away from the end of the cache, it issued a query to pre-fetch 5 more days of data. In the version without predictive caching, the system also fetched 15 days of data as an initialization step, but only fetched data when necessary, i.e. when data for the current frame were not available, the system issued a query for 2 more days of data.

At the maximum panning speed, an analyst should be able to examine two months of data in 57 seconds. With predictive caching, we were able to achieve a performance of 65 seconds including the time to initialize the cache and to pan over all the data. In contrast, the system without predictive caching took 96 seconds because it had to pause and wait for data to be fetched on more than 20 occasions. This not only slowed down analysis, but also disrupted the process even though each pause took only a second or two. Thus predictive caching can significantly improve user experience by enabling smooth and fluid interactions.

6 Discussion

In this paper we used ATLAS to explore challenges to visualization created by the dramatic increase in dataset sizes. This involved a largely straightforward combination of clustered database servers with load balancing and predictive caching algorithms in a client-server architecture. The contribution is in specifying the decomposition of function and the distribution of tasks. Our design is premised on the notion that one must push as much analysis as possible onto the database servers because that is the component which will most naturally scale with data growth. Doing the bulk of analytic computation on the server reduces the amount of data transmitted to the client thereby lowering its disk and bandwidth requirements. The analyst's workstation acts as a cache for the computed results which are then visualized with new analysis requests generated by tracking user interactions.

To push as much analysis as possible onto the database system, we used kdb+ because it parallelizes queries easily and features a rich query language oriented to analysis of time series data. However, for any analytic queries which are expressible by a traditional row-oriented database, we saw that MySQL could also be scaled to provide interactive performance. For most queries, we observed that what matters is the effective working set of data that must be moved through the server's memory hierarchy. Thus, the principal advantage for column-oriented databases is that their memory footprint is smaller.

There is much room to improve the timing and load balancing algorithms. Our current models estimate query time based on the number of server instances available; the time range spanned by the

query; and the number of series to be fetched. The model is precomputed for each type of dataset and is static in nature. One way to make the system more robust would be to adjust the model according to measurements taken at run-time. Furthermore, the model makes no attempt to optimize workload across multiple queries or track query history by server. Such tracking could be use to direct future queries at servers which may have already cached the needed data.

We would like to support more visual analytic tasks. For example, in large datasets, analysts want to reason at higher abstraction levels. Instead of the values of a time series at specific points in time, analysts are more interested in patterns across time periods. Being able to efficiently search for interesting patterns and classify them would be of great value in visual analysis. The ATLAS architecture could easily support pattern searching as described in TimeSearcher [6] in large datasets by distributing the search over a cluster of database servers. However, predictive caching will not be as effective due to the difficulty in predicting the pattern of interest and the algorithm's reliance on inertia in user actions. More sophisticated methods of identifying potentially interesting features in datasets will be needed.

The one thing that is certain is that dataset sizes will continue to grow. Since the value of visual analytics is often critically dependent being able to freely explore data, designing systems to maintain interactivity over massive datasets is now vitally important.

ACKNOWLEDGEMENTS

This work is supported by Max Planck Center for Visual Computing and Communication (MPC-VCC) and the Stanford Regional Visualization and Analytics Center (RVAC). MPC-VCC was jointly established by the Max Planck Society for the Advancement of Science and Stanford University. Stanford RVAC is supported by the National Visualization and Analytics Center (NVAC), a U.S. Department of Homeland Security program operated by the Pacific Northwest National Laboratory (PNNL), a U.S. Department of Energy Office of Science laboratory.

REFERENCES

[1] B. B. Bederson and J. D. Hollan. Pad++: a zooming graphical interface for exploring alternate interface physics. In *UIST '94: Proceedings of the 7th annual ACM symposium on User interface software and technology*, pages 17–26, New York, NY, USA, 1994. ACM.

[2] E. W. Bethel, S. Campbell, E. Dart, K. Stockinger, and K. Wu. Accelerating network traffic analytics using query-driven visualization. *Visual Analytics Science And Technology, 2006 IEEE Symposium On*, pages 115–122, Oct. 2006.

[3] J. Blow. Implementing a texture cache system. Game Developer Magazine, 1998.

[4] P. A. Boncz, F. Kwakkel, and M. L. Kersten. High performance support for oo traversals in monet. In *BNCOD 14: Proceedings of the 14th British National Conference on Databases*, pages 152–169, London, UK, 1996. Springer-Verlag.

[5] R. Brun and F. Rademakers. Root – an object oriented data analysis framework. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 389(1-2):81–86, April 1997.

[6] P. Buono, A. Aris, C. Plaisant, A. Khella, and B. Shneiderman. Interactive pattern search in time series. volume 5669, pages 175–186. SPIE, 2005.

[7] J. V. Carlis and J. A. Konstan. Interactive visualization of serial periodic data. In *UIST '98: Proceedings of the 11th annual ACM symposium on User interface software and technology*, pages 29–38, New York, NY, USA, 1998. ACM.

[8] S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. *SIGMOD Rec.*, 26(1):65–74, 1997.

[9] C. Clarke, N. Craswell, and I. Soboroff. The trec terabyte retrieval track. *SIGIR Forum*, 39(1):25–25, 2005.

[10] CRSP. Center for research in security prices. http://www.crsp.com/.

[11] P. R. Doshi, G. E. Rosario, E. A. Rundensteiner, and M. O. Ward. A strategy selection framework for adaptive prefetching in data visualization. In *SSDBM'2003: Proceedings of the 15th international conference on Scientific and statistical database management*, pages 107–116, Washington, DC, USA, 2003. IEEE Computer Society.

[12] P. R. Doshi, E. A. Rundensteiner, and M. O. Ward. Prefetching for visual data exploration. In *DASFAA '03: Proceedings of the Eighth International Conference on Database Systems for Advanced Applications*, page 195, Washington, DC, USA, 2003. IEEE Computer Society.

[13] G. W. Furnas. Generalized fisheye views. *SIGCHI Bull.*, 17(4):16–23, 1986.

[14] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 205–216, New York, NY, USA, 1996. ACM.

[15] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden. Performance tradeoffs in read-optimized databases. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 487–498. VLDB Endowment, 2006.

[16] S. Heman, M. Zukowski, A. P. de Vries, and P. A. Boncz. Efficient and Flexible Information Retrieval Using MonetDB/X100. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, USA, January 2007. (Demo Paper).

[17] J. Lin, E. Keogh, S. Lonardi, J. P. Lankford, and D. M. Nystrom. Visually mining and monitoring massive time series. In *KDD '04: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 460–469, New York, NY, USA, 2004. ACM.

[18] W. A. Pike, C. Scherrer, and S. Zabriskie. Putting security in context: Visual correlation of network activity with real-world information. In *VizSEC 2007: Proceedings of the Workshop on Visualization for Computer Security*, pages 203–220, Berlin, 2008. Springer-Verlag.

[19] L. Rock and J. Brown. Flow visualization using ms-excel. http://www.cert.org/flocon/2008/presentations/rock_FloCon2008.pdf, 2008.

[20] R. T. Snodgrass and C. S. Jensen. *Developing Time-Oriented Database Applications in SQL*. Morgan Kaufmann Publishers, 2000.

[21] M. Stonebraker. One size fits all: A concept whose time has come and gone. http://www.databasecolumn.com/2007/09/one-size-fits-all.html, 2007.

[22] M. Stonebraker. Supporting column store performance claims. http://www.databasecolumn.com/2008/03/supporting-column-store-perfor.html, 2008.

[23] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented dbms. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.

[24] Sybase. Sybase iq. http://www.sybase.com/, 1997.

[25] C. C. Tanner, C. J. Migdal, and M. T. Jones. The clipmap: a virtual mipmap. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 151–158, New York, NY, USA, 1998. ACM.

[26] D. R. Tesone and J. R. Goodall. Balancing interactive data management of massive data with situational awareness through smart aggregation. *Visual Analytics Science and Technology, 2007. VAST 2007. IEEE Symposium on*, pages 67–74, Oct. 30 2007-Nov. 1 2007.

[27] J. J. Thomas and K. A. Cook. *Illuminating the Path- The Research and Development Agenda for Visual Analytics*. IEEE Computer Society Press, 2005.

[28] E. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, second edition, 2001.

[29] A. Whitney. Kdb+. http://www.kx.com, 2003.

[30] L. Williams. Pyramidal parametrics. *SIGGRAPH Comput. Graph.*, 17(3):1–11, 1983.