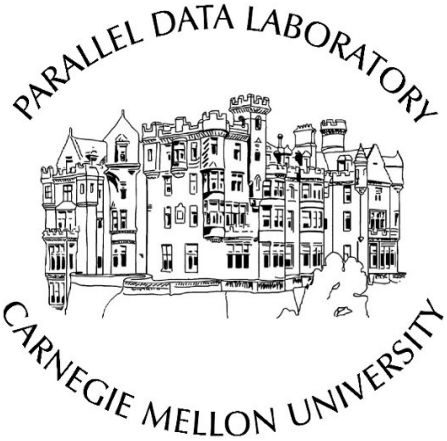


Using Data Transformations for Low-latency Time Series Analysis

Henggang Cui*, Kimberly Keeton†, Indrajit Roy†
Krishnamurthy Viswanathan†, and Gregory R. Ganger*
**Carnegie Mellon University,* †*Hewlett-Packard Laboratories*

CMU-PDL-15-106
August 2015

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

Time series analysis is commonly used when monitoring data centers, networks, weather, and even human patients. In most cases, the raw time series data is massive, from millions to billions of data points, and yet interactive analyses require low (e.g., sub-second) latency. Aperture transforms raw time series data, during ingest, into compact summarized representations that it can use to efficiently answer queries at runtime. Aperture handles a range of complex queries, from correlating hundreds of lengthy time series to predicting anomalies in the data. Aperture achieves much of its high performance by executing queries on data summaries, while providing a bound on the information lost when transforming data. By doing so, Aperture can reduce query latency as well as the data that needs to be stored and analyzed to answer a query. Our experiments on real data show that Aperture can provide one to four orders of magnitude lower query response time, while incurring only 10% ingest time overhead and less than 20% error in accuracy.

This technical report is a full version of the SoCC 2015 paper "Using Data Transformations for Low-latency Time Series Analysis". It is also published as Hewlett-Packard Laboratories technical report HPL-2015-74.

Acknowledgements: We would like to thank Brad Morrey for helping us to set up LazyBase, as well as Haris Volos, Manish Marwah, and Rob Schreiber for fruitful discussions. We also thank our shepherd Jeff Chase and the SoCC reviewers for their detailed feedback. We thank the members and companies of the PDL Consortium (including Actifio, APC, EMC, Emulex, Facebook, Fusion-IO, Google, Hewlett-Packard, Hitachi, Huawei, Intel, Microsoft, NEC Labs, NetApp, Oracle, Panasas, Riverbed, Samsung, Seagate, STEC, Symantec, VMWare, Western Digital) for their insights, feedback, and support. This research is supported in part by the Intel Science and Technology Center for Cloud Computing (ISTC-CC).

1 Time is of the essence

Monitoring of mechanical and natural systems generates a massive amount of time series data. For example, data centers collect information about thousands of servers, and meteorologists have hundreds of years of weather information. In such cases, data is ordered by time, and often it is critical to obtain query results in near real-time [6, 10]. Consider the case of data center monitoring. System administrators typically instrument their servers, network, and applications, and collect information at the granularity of seconds. To detect and alleviate performance anomalies, the administrator needs to interactively correlate new incoming data with historical observations, and pinpoint the cause.

Two challenges make it difficult to achieve near real-time response to time series queries. First, the number of events in a time series can be in the billions, making it difficult to manage and query data. Twitter collects 170 million time series metrics each minute [6], while Facebook's Scuba stores around 70 TB of compressed monitoring data [10]. Even a modest cluster of 100 servers, instrumented to collect a few metrics per second will generate more than 60 million events per week. Due to the data size, it is often only cost-effective to store this data on disk, which unfortunately increases response time when millions of rows have to be scanned during a query. Second, many time series queries are themselves complex, and involve intricate statistical analysis such as correlation. These queries can take hours to run on raw data. For example, our experiments show that a single process may take more than 5 minutes to calculate correlation between two time series with 24 million points. Thus, correlating across hundreds of time series, each representing data from a server, may take hours.

There are many existing options to store and analyze time series data, including traditional relational databases (e.g., MySQL [27]), distributed databases (e.g., HP Vertica [7]), systems built on top of the Hadoop stack (e.g., OpenTSDB [8], HBase [3]), and custom in-memory distributed systems (e.g., Scuba [10]). These systems overcome the challenge of massive data size by scaling up or scaling out, essentially relying on more hardware resources to cope with the demands of time series data. Unfortunately, simply using more hardware resources is insufficient to meet sub-second query latency requirements for complex time series analyses. Without any domain specific optimization, complex queries such as correlation and anomaly detection on large time series still take tens of minutes or more on these systems (Section 5). Although Scuba-like in-memory systems improve response time by avoiding disks, they are limited by the capacity of the aggregate cluster memory. In fact, Facebook has been forced to add new machines every 2-3 weeks to keep up with the increase in data collection [10].

In this paper, we argue that domain-specific ingest-time transformations, such as converting time series data to the frequency domain and retaining important coefficients, can vastly improve query latency. These transformations reduce the amount of data that needs to be stored and processed to answer a query. We describe Aperture, a framework that uses time series specific transformations to answer queries approximately and in near real-time. Aperture consists of two components: (1) an *ingest module* that executes user-defined transformations on batches of input data, and (2) a *query module* that uses transformed data to answer queries while meeting an error bound. Aperture can even process ad-hoc queries (i.e., arbitrary data analysis tasks), by recreating the original data, when needed, from invertible summaries.

The contributions of the paper are:

- We propose the use of ingest time transformations to summarize time series data, trading bounded reduction in query accuracy for lower query latency. The transformations vary from simple sampling techniques to complex wavelet transformations that speed up correlation

queries. For each, we also capture the sensitivity to error.

- We describe how the ingest time transformations and approximate query processing can be implemented in an existing database [17]. We modified the ingest phase of the database to support user defined transformations, and the query phase to use summarized data while adhering to an error bound.
- We have applied our techniques on three real world use cases. Extensive experiments, on up to 800 million rows, show that Aperture can reduce query latency by one to four orders of magnitude, while incurring less than 20% error in accuracy and 10% increase in ingest overhead.

2 Background

Data scientists want the ability to perform a number of analyses over time series data, ranging from simple statistics (e.g., `average`, `min`, `max`) to complex calculations like correlations, convolutions and modeling. In this section, we describe common use cases for time series analysis, highlight existing approaches and articulate the goals that Aperture addresses.

2.1 Use cases

Time series analysis has many applications. We discuss three real-world use cases below.

Correlation search. An important time series analytics task is to find correlations among data series. These correlations can be used to understand the relationships between different data series and to denoise data, predict data, and estimate missing observations. For example, the effect of server utilization on data center room temperature can be understood by calculating the correlation between the server utilization data and the room temperature data. In this paper, we use the commonly employed *Pearson correlation coefficient* to measure the correlation between two data series [9]. In the correlation search task, for a given data series we need to identify all data series in a collection that are significantly correlated with it.

Anomaly detection. Time series analysis is also used to detect anomalies. To detect anomalies, system administrators first train a model on historical data, and then apply the model on each incoming data point. If a data point does not fit well into the model, the event is flagged as an *anomaly*.

Monitoring event occurrences. Another common operation on time series data is to summarize the number of times that a certain event happens in a log. For example, many security monitoring tools log events such as time of login and source IP address. A system administrator can then use techniques such as counting the occurrence of an IP address visiting their web service to detect anomalies, and hence security attacks.

2.2 Related work

Prior approaches for analyzing time series data differ along several dimensions, including the complexity of the supported analyses and queries, whether queries operate on recent data or historical data (or both), and whether queries provide approximate or precise results. Existing database systems, such as HP's Vertica [7] and OpenTSDB [8], are good at managing large amounts of data efficiently, but usually provide only limited (if any) support for complex time series analytics. As a result, practitioners often resort to a combination of tools, including SQL queries and Matlab-like statistical software or other scripts, with the data residing in a database or distributed file

system. Many systems focus on near-real-time analytics on recent data (e.g., Facebook's Scuba [10]) or on streaming data (e.g., Apache Storm [4] and Spark Streaming [29]), rather than permitting analysis of historical data. Conversely, other systems (e.g., Cypress [24]) focus on rich time series analytics for historical data without providing quick insight into recent data. Related work describes techniques for approximate queries that permit tradeoffs between query accuracy and performance (e.g., BlinkDB [12]) and systems that store data in compressed representations that enable limited queries directly to the compressed format (e.g., Succinct [11]). In this section, we describe several key systems in more detail.

HP's Vertica database provides support for "live aggregate projections" [7], which pre-calculate aggregate functions (e.g., `sum`, `count`, `min`, `max`, `average`, `top-k`) when raw data is initially loaded, and then maintain the aggregate functions as new data is subsequently loaded. Aggregate queries to the pre-computed projection are faster than scanning the underlying table and computing the aggregate at query time. Live aggregate projections provide statistics only for all observed data, rather than permitting queries on a subset of the data. To date, live aggregation doesn't support more complex transformations.

Facebook's Scuba [10] is a distributed in-memory database used for interactive, ad hoc analyses over live data. Scuba provides a SQL query interface for aggregation (e.g., `count`, `min`, `max`, `histogram`) and grouping queries, as well as a GUI that produces time series graphs and other data visualizations. Scuba, which is intended for queries over recent data (a few hours to a few weeks), stores compressed and sub-sampled data in the memory of a cluster of machines; as new data enters the system, older data is aged out, limiting the history that can be queried.

Cypress [24], a framework for archiving and querying historical time series data, decomposes time series to obtain sparse representations in various domains (e.g., frequency and time domains), which can be archived with reduced storage space. Cypress captures high-level trends of signals using FFTs and downsampling in low-frequency trickles, abrupt events in spike trickles, and random projections (sketches) that preserve inter-signal correlations in high-frequency trickles. Cypress supports statistical trend, histogram and correlation queries directly from the compressed data. Unlike Scuba, Cypress is focused on efficiently archiving data, rather than on the analysis of recent data.

BlinkDB [12] is a distributed approximate query engine for running interactive SQL-based aggregation queries; it allows users to trade off query accuracy for response time, by running queries on data samples. Multiple stratified samples are created based on several criteria, including data distribution, query history and the storage overhead of the samples. At query time, BlinkDB uses error-latency profiles to estimate a query's error and response time on each available sample; a heuristic then selects the most appropriate sample to meet the query's goals. BlinkDB focuses on constructing and using samples over stored datasets, rather than on maintaining such samples in the face of dynamically changing datasets, as required in time series analytics.

Succinct [11] stores data using an entropy-compressed representation that allows random access and natively supports count, search, range and wildcard queries without requiring a full data scan. Although not specifically targeted at time series data, this approach is similar in spirit to Aperture, as the compressed representation permits a larger memory-resident data size and speeds queries, due to the ability to directly query the compressed representation. Succinct does not provide an extensible framework for different summarization techniques to enable a wide range of statistical queries.

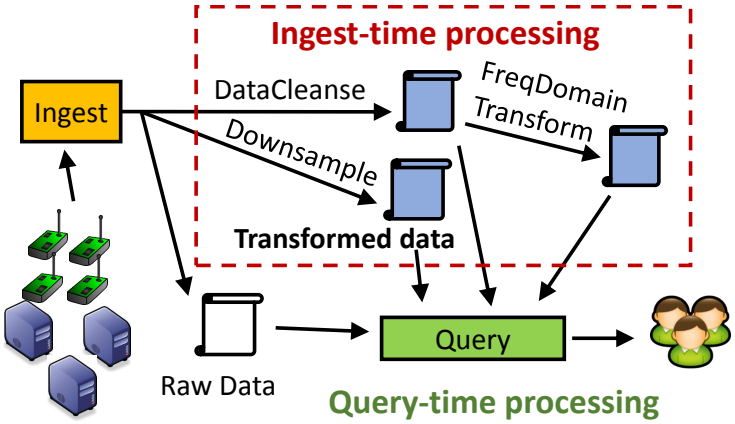


Figure 1: Ingest-time processing and query-time processing. Three transformation outputs are generated from the raw data. `FreqDomainTransform` is chained after `DataCleanse`.

2.3 Goals

Our goal is to support rich time series analytics, including methods to address the case studies described above, for both recent and historical data. Queries should provide interactive response times, and their answers should have acceptable accuracy. The costs of maintaining any additional data representations to answer queries should be low.

There is a tension between processing the raw data in its entirety to provide exact (i.e., non-approximate) answers and the cost of reading and processing all data, especially when queries include historical data. To sidestep this tension, our methodology should permit compaction of the raw input data, so that less data needs to be read and processed at query time. Ideally, this compaction would provide lossless summarization, but the types of analyses we target may tolerate approximate answers, opening the door to a variety of domain-specific analysis techniques with tunable accuracy-overhead trade-offs. Additional benefit is possible if the act of compacting the data would provide some useful proactive analysis of the data, to save effort at query time.

In order for this approach to be viable, maintaining the compact representation should require only minimal additional overhead beyond the baseline ingestion of the raw data. Because time series data is logically append-only (i.e., only adding new data, rather than updating old data), this overhead reduction may be achievable through analyses that can be incrementally performed on the latest incoming data, without requiring re-computation over the data in its entirety. We describe our approach, Aperture, in Section 3, and we discuss our implementation of Aperture as well as alternative implementation options in Section 4.

3 Aperture

The Aperture approach uses time series specific data transformations to reduce query response time. Figure 1 shows the different components of the system. The *ingest module* applies different data transformations on raw input to create summaries, in a per-*window* basis. Aperture supports multiple transformations on the same input, and it supports chained transformations. The *query module* utilizes data transformation by translating queries on the raw data into those on the transformed data.

Example. Consider the example of data center monitoring. Data centers can easily generate 100s of gigabytes of monitoring data each day. However, if we assume that the core characteristics of the data changes slowly, we can use downsampling to reduce the number of events that are retained for query processing. A simple example of downsampling is to retain every 10^{th} item of the input. This downsampling technique will reduce the data to one-tenth of the original. Of course, such naive sampling may impact accuracy, and in Section 5.1 we describe sophisticated transformations that vastly improve upon downsampling. In addition, Aperture allows us to retain multiple transformed versions of the same data. For example, we can retain data downsampled to both one-twentieth and one-tenth of the input. Different transformations on the same data allow the user to trade off query accuracy for performance.

This section describes the design of Aperture, including its ingest-time data transformations and query processing using transformed data.

3.1 Data model and assumptions

Aperture stores data as sorted *rows* in *tables*. Each table has a *schema* that defines the fields of each row. Each row can contain any number of *key fields* for sorting and any number of *data fields*.

While a row can have arbitrary number of fields, we encourage users to define their table schema as *narrow tables*, such as the OpenTSDB [8] table format: {`metric, tags, time, value`}, where `value` is the only data field. This narrow table format reduces data access overhead by ensuring queries read only the metrics that are needed to answer them.

Clients upload data to Aperture in *upload batches*. Each upload batch often represents a group of data in a new time range, instead of updates to old data.

3.2 Ingest-time processing design

This section describes the design of ingest-time processing of Aperture. The transformations to be applied are defined together with the table schemas, and Aperture performs these transformations when the data is ingested. Aperture transforms each upload batch independently, allowing streaming processing and parallelism across different batches. For each upload batch, Aperture first sorts the rows by keys [1] and then applies user defined transformations based on the table schema. The transformed data are written to the database in output tables as specified. Each transformation is a mapping from a set of fields of the input table to a set of fields of the output table. After applying transformations, users can choose to keep or discard the raw data. For example, this feature can be used to pre-process uploaded data by defining a data cleansing transformation (such as interpolation), and storing only the cleansed data, while discarding raw data.

In the rest of this subsection, we will describe the important features of Aperture's transformation framework that enable efficient timeseries data analyses.

3.2.1 Windowed transformation

A key feature of Aperture is its *windowed data transformation*. It allows users to divide timeseries data into windows, and generate one window of transformed data from one window of input.

Dividing transformation windows. Users can specify how the transformation windows are divided separately for each individual transformation. Aperture assumes that the data of each

[1] Although the uploaded data is often already ordered by time, we still need to sort the time series of different metrics by their metric keys.

window are included in a single upload batch, so that each batch of upload could be processed independently. [2] We provide two options for users to specify window boundaries.

The `window-by-count` option groups every k (sorted) rows into one window, where k is the *window size*. This option is suitable for transformations that expect a fixed number of rows.

The more sophisticated `window-by-value` option divides window boundaries based on the key fields. It requires three parameters: name of the time key (typically `time`), starting time offset (in case time does not start from zero), and window granularity. Suppose the table has k key fields $\{key_1, ..., key_k\}$ and $time$ is the i-th key, using this option, each window will contain rows with the same $\{key_1, ..., key_{i-1}, \lfloor \frac{time-offset}{granularity} \rfloor\}$ keys. This option can be used to group the data in the same range of time into one window. For example, suppose the table has key fields {`metric, server, time`}, and we set the window size to one hour. The `window-by-value` option will then group the data of the same metric from the same server on an hourly basis.

Both options have the property that each transformation window is a contiguous range of rows in the sorted upload batch. As a result, we can feed the windowed data to each transformation function with only one pass of the data.

Assigning keys to transformed data. Aperture is able to translate queries on raw data to queries on transformed data. To support this query translation, Aperture provides two options to generate the keys for the transformed data. Both options will use the key fields of the input table as the prefix of the transformed table keys.

The `copy-from-each` option uses the key of the j-th input row as the key prefix for the j-th output row. This option is suitable for one-to-one transformations.

The `copy-from-first` option uses the key of the first input row in the transformation window as the key prefix for all output rows and uses other key fields to distinguish different output rows. This option is often used in combination with the `window-by-value` option. Using our previous example, the input rows with the same $\{key_1, ..., key_{i-1}, \lfloor \frac{time-offset}{granularity} \rfloor\}$ keys will be grouped into one window, and $\{key_1, ..., key_{i-1}, window\_start\_time, other\_keys\}$ will be the keys of the transformed rows. For example, if the input table has keys {`metric, server, time`}, we can use {`metric, server, time, summary-id`} as the keys of the transformed table, where `time` is the window start time and `summary-id` distinguishes different summaries.

Both options have the property that a key range r that aligns with the transformation windows in the input table will be transformed into the same key range in the transformed table. Using our previous example, when we use a window granularity of one hour, the key range "from {`cpu-util,server0,3am`} to {`cpu-util,server0,6am`}" is transformed into the same key range in the transformed table. This property simplifies query translation. Also, by assigning row keys this way, we only need to sort the transformed rows of each window, and rows across different windows are naturally sorted, eliminating additional sorting work.

3.2.2 Transformation chaining

The design of Aperture allows users to chain multiple transformations together. Transformation chaining is useful for many timeseries applications. For example, when the collected data has missing values, one can first perform an `interpolate` transformation to cleanse data, before doing other transformations. For each upload batch, Aperture applies the transformations in the order defined by the users, and the transformation outputs are kept in memory before all transformations are completed. As a result, the users could chain `Transform-B` after `Transform-A` by simply using the output of `Transform-A` as its input.

[2] The window granularity is often much smaller than the upload batch size, making this assumption reasonable.

3.2.3 Multiple versions of transformed data

Aperture allows applications to generate multiple transformations from the same input, by simply defining multiple transformation entries in the table schema. Moreover, Aperture allows users to define multiple transformations using the same function but different parameters. For example, to satisfy the accuracy requirements of different queries, users can downsample data with different downsampling rates and store multiple versions of downsampled data in the database. Users can use the most suitable transformed data to answer each specific query.

3.3 Transformation examples

We describe the details of our transformation framework using the downsampling example. [3] In this example, a client uploads server utilization data to the `Raw` table, which contains three key fields {`metric, server, time`} and one data field `value`. We use Aperture to generate multiple versions of downsampled data with different downsampling rates. Before downsampling, an `interpolate` transformation cleanses the data by interpolating the missing values. `Raw` data is discarded after transformation. The table schemas and ingest-time transformations are defined in an XML formatted *schema file*. Listing 1 shows the transformation entries for this example.

Listing 1: Example transform entries in a schema file

```
<settings discard-raw:"yes">
<!-- Interpolate -->
<transform
  func="interpolate"
  <!--from 0, for every 3600 time-->
  window="by_value:time,0,3600"
  input="Raw:time,value"
  app-params=""
  output="Interpolated:time,value"
  keyprefix="first:metric,server"
>
<!-- Downsample -->
<transform
  func="downsample"
  window="by_value:time,0,3600"
  <!-- chained after interpolate -->
  input="Interpolated:time,value"
  <!-- downsample rate is 8 -->
  app-params="8"
  output="Downsample_1:time,value"
  keyprefix="first:metric,server"
>
<!--Downsample using other params-->
...
```

Each `transform` entry defines a transformation on a specific table, including the transformation function, how windows are created, input and output tables, application-specific parameters, and how transformed data keys are assigned. In this example, we have one `interpolate` transformation and several `downsample` transformation chained after it.

[3] This overly simplistic downsampling example is provided to illustrate the features of our framework. Aperture also supports more sophisticated transformations, as described in Section 5.1.

The `interpolate` transformation divides windows by value with parameters {`time,0,3600`}, which creates hourly windows (e.g., a `time` window of 3600), starting from `time`=0. As specified in the `keyprefix` option, the `metric` and `server` fields of the first input row in the window are used as the key prefix for the output rows of the same window. The `downsample` transformation downsamples the interpolated data with a downsampling rate of 8, as specified as an application parameter.

The actual transformation functions are implemented as C++ functions in application source files. Each of these functions implements an interface that takes the input fields and application parameters as inputs, and writes out the output fields.

3.4 Query processing design

Aperture users write *query programs* to fetch data and run analyses. Our Aperture query APIs enable them to easily run efficient analyses on the most suitable transformed data. Users specify their requirement on data transformations by providing a *utility function*. Aperture automatically finds the transformation that yields the highest utility value and fetches the transformed data for the users.

3.4.1 Query processing procedures

Query processing in Aperture involves choosing which (if any) transformation to use and translating the query accordingly.

Choosing transformations. Aperture reads the transformation information from the table schemas at runtime and calculates the utility of each transformation using the provided utility function. The transformation with the highest utility will be selected. Algorithm 1 gives an example of answering queries using downsampled data. In this example utility function, a downsampling rate of 8 is most preferred. Downsampling rates larger than 8 have lower utilities than those less than 8, meaning that the user prefers only limited downsampling. In our more sophisticated use cases (described in Section 5.1), the utility function often specifies the error bound requirements.

Algorithm 1 Example query using downsampled data.

$transform \leftarrow searchTransform(table, utility)$
$data \leftarrow requestData(table, keyRange, transform)$
Defined *utility* function:
function UTILITY(t)
    if $t.func \neq$ "downsample" **then**
        $utility \leftarrow 0$
    else if $t.downsampleRate > 8$ **then**
        $utility \leftarrow \frac{1}{t.downsampleRate}$
    else
        $utility \leftarrow t.downsampleRate$
    end if
end function

Translating queries and fetching data. Aperture automatically translates the original query on raw data into a query on the transformed data based on the chosen transformation. As described in Section 3.2.1, Aperture's key assignment strategy simplifies query translation, because the key range of the transformed data will be the same as the original key range. Aperture then

uses the translated query information to fetch the transformed data from the database. If the raw query is a range query, which is often the case for timeseries data analyses, the translated query will also be a single range query, allowing efficient data fetching.

Identifying transformation window boundaries. Aperture provides the transformed data to users with window boundary information. This is also enabled by our key assignment options. When `copy-from-each` is used, the keys of the transformed rows are the same as those of the input rows, and the window boundaries can be identified in the same way as we divide windows at ingest-time. When `copy-from-first` is used, the transformed data of the same window will share the same key prefix (copied from the first input row of the window).

3.4.2 Inverse transformation for ad-hoc queries

In most cases, the transformations are designed such that the query-time data analyses can directly use the transformed data. However, Aperture can also be used to answer ad-hoc queries (i.e., arbitrary data analyses tasks) by reconstructing the original data from transformed data, In order to do that, users can provide an inverse transform function to Aperture, and Aperture will automatically reconstruct the original data from the transformed data using that function. This inverse transformation function does not necessarily need to reconstruct exactly the data ingested. Instead, we often allow some errors and reconstruct data using more compact transformed data to reduce query latency. For example, as we described in Section 5.4, we can use a wavelet transform to generate a compact representation of the timeseries data, and at query-time, we reconstruct the data to answer any queries with bounded errors.

4 Implementation

We have created a prototype of Aperture by extending an existing database system called LazyBase [17]. In LazyBase, the uploaded data is processed through an *ingest pipeline* which consists of three stages: `receive`, `sort`, and `merge`. The `receive` stage collects client uploads and makes them durable. The `sort` stage orders rows in each batch by their keys, and the `merge` stage merges newly ingested updates with the existing data. Each upload batch is the granularity of transactional (e.g., ACID) properties throughout the database. The three stages are run by different worker processes in parallel. The intermediate results, as well the final output tables, are stored in the file system as DataSeries [13] files, a compressed record storage format. For read-only queries, LazyBase provides snapshot isolation, where all reads in a query will see a consistent snapshot of the database, as of the time that the query started.

We implemented Aperture's ingest-time transformation framework by modifying the `sort` stage of the LazyBase ingest pipeline, as shown in Figure 2. The modified `sort+transform` stage first sorts each upload batch and then applies the user defined transformations. Transformation windows are divided based on the specified window options. The transformed data is written together with the sorted raw data (if not discarded) as the output of this stage. We implemented the query processing module of Aperture by adding a layer above LazyBase's query library to select transformations, translate queries, and partition data windows.

Alternative implementation options. Aperture can be implemented in a variety of software infrastructures. One option is a relational database that supports materialized views and user-defined functions (UDFs) that can be incrementally evaluated on newly added data (e.g., something akin to Vertica's live aggregation projections, but with the addition of UDFs). The ingest phase of the system would perform data transformations, and the query engine could be extended to use transformed data. Such a system would look similar to our prototype on LazyBase.

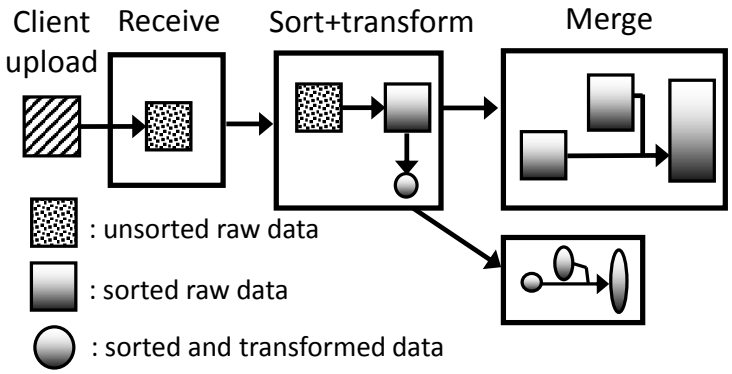


Figure 2: Aperture ingest pipeline.

Aperture could also be implemented in new scale-out systems, such as a deployment of Hive [25] or HBase [19] that receives streaming data from Storm [26]. In such a deployment, Storm would perform ingest time transformations, while Hive could be modified to use transformed data to answer queries. Users would be able to add new data transformations by writing a new Storm program.

5 Case studies and evaluation

We use three real-world use cases to evaluate the benefits of Aperture. Our results show that: (1) using transformations and approximate query processing can reduce query response time by one to four orders of magnitude, (2) the overheads of calculating transformations have minimal impact on the ingest throughput of the database, and (3) the transformed data is small enough that it is practical to maintain different versions of transformed data and trade accuracy for performance.

Experimental setup. Our experiments use a cluster of HP ProLiant DL580 machines, each with 60 Xeon E7-4890 @2.80GHz cores, 1.5 TB RAM, running Fedora 19. They are connected via a 10 Gigabit Ethernet. We use one machine for the case study experiments from Section 5.1 to Section 5.4, emphasizing the performance improvement from ingest-time transformations. For these experiments, we configured Aperture to launch one database worker (a process with one worker thread and several I/O threads) for each of the three pipeline stages on each machine. We use a separate client process to upload data in multiple batches. In Section 5.5, we examine the scalability of Aperture by adding more machines and having more clients uploading or querying data simultaneously.

The raw input data as well as the internal database files are stored in the in-memory file system. The large-memory machines allow us to keep all data in memory, thus demonstrating the benefits of Aperture over an ideal baseline that does not need to use disks. Otherwise, only the compact transformations of Aperture would fit in-memory and the baseline (because of the lack of transformations and large data size) would be penalized for accessing disk. We describe datasets used in the corresponding use case sections. In the case of multiple machines, the intermediate database files are exchanged between workers via network sockets.

Experimental methodology. For the query latency numbers, we run each query three times and use the median latency as the result, showing the minimum and maximum latencies using error bars.

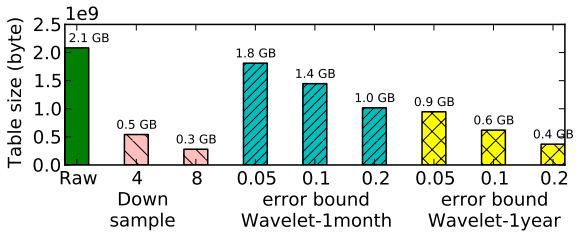
5.1 Correlation search

The correlation search task identifies similarities among data series by measuring their correlation. We use the Pearson correlation coefficient to measure the correlation. Given two data series x and y each of size N, their Pearson correlation coefficient is given by:

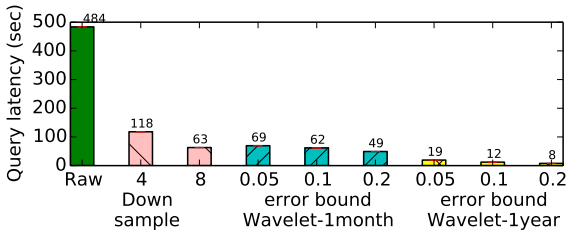
$$corr(x, y) = \frac{\sum_{i=1}^{N}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{N}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{N}(y_i - \bar{y})^2}} \quad (1)$$

where x_i and y_i represent the i-th value in the data series x and y respectively, and $\bar{x}$ and $\bar{y}$ are the mean values of x and y respectively. The correlation coefficient takes values in the range of $[-1, 1]$, and a larger absolute value indicates a greater degree of dependence between x and y. It is common in analytics applications to apply a threshold to the correlation coefficient to decide whether two data series are significantly correlated. In the correlation search task, for a given data series we need to identify all the data series in a collection that are significantly correlated with it.

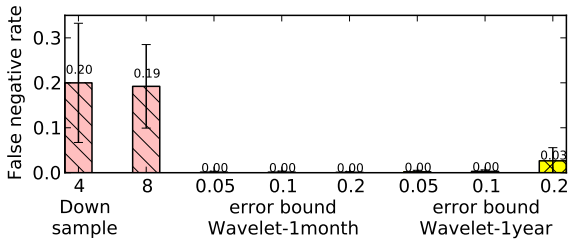
The most straightforward way of identifying pairs of correlated data series is to first fetch the interested data series (sometimes the entire dataset) from the database and then calculate their pairwise correlations using a local program (such as a C++ program or a Matlab script). To compute the pairwise correlation of a pair of length N data series requires $O(N)$ computations, so that if there are k pairs of such data series, the whole computation takes $O(Nk)$ time, which might be infeasible for large tasks. If it were possible to determine if two data series are likely to be correlated by performing significantly fewer than $O(N)$ computations, then this task could be sped up significantly.



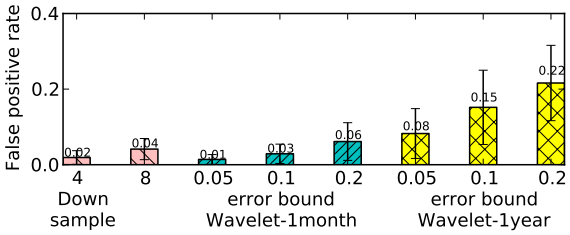
(a) Table size



(b) Query latency



(c) Query false negative rate



(d) Query false positive rate

Figure 3: Correlation search results using transformed data vs. using raw data.

Dimension-reduction using Haar wavelet. There are various dimension-reduction techniques that can be used to speed up correlation search tasks, by transforming time series into alternate more compact representations. These techniques include discrete wavelet transform [15, 28, 20],

discrete Fourier transform [30], singular value decomposition [22], and Piecewise Constant Approximation [21].

All of those techniques can be implemented using our Aperture framework. For this paper, we implemented and evaluated a specific type of discrete wavelet transform known as the Haar wavelet transform [5, 16]. Given a time series x, the Haar wavelet transform represents x as a linear combination of Haar wavelet times series, *i.e.*, $x = \sum_{i=1}^{N} a_i \phi_i$, where $\phi_1, \phi 2, \ldots, \phi_N$ are Haar wavelets, and the a_is are called the *wavelet coefficients* of the time series x. Given the wavelet coefficients, one can reconstruct the original time series exactly and therefore this transform is lossless. Further, given two time series x and y, their correlation can be computed from just their corresponding wavelet coefficients. The wavelet representation is usually sparse for real world data, meaning that many of the wavelet coefficients are zero or very small. Therefore, one can obtain a compact but good approximation of time series by storing only the largest wavelet coefficients and treating the rest as zero. Such compact representations can be used to obtain good approximations of the correlation between two time series. We use a *transformation error bound* parameter to determine the number of coefficients that need to be stored. [4]

Dataset. We evaluate the performance of correlation search with and without transformations on a public dataset collected by National Climatic Data Center [1], which contains the daily climatic data collected at their climatic stations around the globe since 1929. We define the schema of the raw table as {`metric, station-name, time, value`}, which is similar to the OpenTSDB [8] schemas. The data we use has 350 million rows in total, with three metrics: temperature, wind speed, and dew point. We upload the entire dataset to Aperture in 132 batches.

Transformation. In Aperture, we define a `wavelet` transformation, which uses the `window-by-value` option to divide the data of the same {`metric, station-name`} for every range of *granularity* in `time` into one window. It then calculates the wavelet coefficients of each window and keeps the necessary coefficients to satisfy the given error bound. In our implementation, the wavelet coefficients of each window are represented as an array of {`coef-id, coef-val`} pairs and are stored as a byte array in a single row of the transformed table.

In this experiment, we generate six `wavelet` transformations with different window granularities (1 day and 1 hour) and error bounds (5%, 10%, and 20%). In addition to the `Raw` data, we use two `downsample` transformations as another baseline, with downsampling rates of 4 and 8. The original raw data has missing values, so we use an `interpolate` transformation to cleanse the data before `wavelet` and `downsample` are applied. The raw data is discarded after being transformed, and the `Raw` bars in Figure 3 refers to the interpolated data.

Results. Each test query compares 10 years of temperature data collected at one station with the temperature, wind speed, and dew point data collected at all other stations over the entire history. It calculates the correlation of all such pairs by sliding the range on a yearly basis (2005 to 2015, 2004 to 2014, and so forth), and reports those pairs with correlations larger than 0.8.

Figure 3(a) and Figure 3(b) summarize the sizes of different tables and the latencies to perform one correlation query. The table size and query latency decrease when larger downsample rates, larger error bounds, or larger window granularities are used, because of the vast reduction in the amount of data accessed and processed to answer the query.

Figure 3(c) and Figure 3(d) summarize the accuracies of the query results, in terms of false negative rates and false positive rates. We randomly picked 5 queries to run and calculate the mean and standard deviation (shown in error bars) of the accuracies. The error increases when we use less data to answer the query (larger downsample rates, larger error bounds, or larger window granularities). Using a `wavelet` transformation with a window granularity of 1 year and an error

[4] More details about correlation calculation using wavelet coefficients can be found in Appendix A.

bound of 20%, for example, Aperture provides a query latency of 8 seconds, which is only 1.7% of the baseline latency of 484 seconds. The false negative rate and false positive rates are only 3% and 22%, respectively.

The false positives can be eliminated by validating the detected pairs using the raw data. For the cost of doing such validation, suppose the number of true correlated pairs is nc, the number of all pairs is n, the false positive rate is fpr, and the time of comparing one pair is t_{raw} for raw data and t_{trans} for transformed data, the total query time with validation is $t_{trans} \times n + t_{raw} \times nc \times (1 + fpr)$, compared to $t_{raw} \times n$ using raw data. When $t_{trans} \ll t_{raw}$ and $nc \ll n$, it's much more efficient to use the transformed data, and the only errors are false negatives, so it's more important to have a low false negative rate. From Figure 3(c), we find the false negative rates of using `wavelet` transformations are much lower that those using `downsample`.

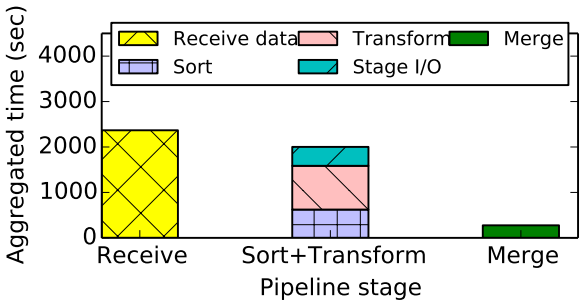


Figure 4: Aggregated time spent on each pipeline stage for correlation search.

In order to satisfy different query requirements, we calculate and maintain multiple transformations with different parameters. For example, we need `wavelet` transformations with a granularity of one month to answer queries on specific months, but those with a granularity of one year are more efficient for answering longer ranged queries.

An important question is whether ingest-time transformations limit the throughput of ingesting data. In our experiments, the total time to ingest all the uploads (350 million rows) without doing any transformations is 2629 sec (133 thousand rows per second). When we do one `interpolate`, two `downsample`, and six `wavelet` transformations, the ingest time is 2723 sec (only 4% overhead). We explain the low overhead by summarizing the time that each ingest pipeline stage spends on ingesting the data in Figure 4. Recall that the three stages in our ingest pipeline are executed by separate worker processes, so the total time for the pipeline to finish ingestion mostly depends on the stage that takes the most time. As shown in the figure, our low ingestion overhead is majorly because of two reasons. First, the cost of doing all these transformations is quite low, just a little more than the sorting time, because wavelet transformation is only $O(N)$ complexity. Second, even with the additional transformation work, the `sort+transform` stage still spends less time than the `receive` stage, meaning that most of the transformation calculation overlaps with data reception.

5.2 Anomaly detection

ARMA model. The anomaly detection task detects anomalous events based on historical observations. The common approach to doing this is to train a model based on historical data, and then flag incoming data points as anomalies based on how well the point fits into the model. For example, the autoregressive moving average (ARMA) model [14] is often used in time series modeling. In this model, the observation x_t at time t is expressed in terms of the past observations and the *residual*

errors ε in modeling past observations:

$$\hat{x}_t = c + \sum_{i=1}^{p} \varphi_i x_{t-i} + \sum_{i=1}^{q} \theta_i \varepsilon_{t-i}.$$

Here p is the number of autoregressive terms, and q is the number of moving-average terms. The residual of x_t is defined as its difference from the predicted value $\varepsilon_t = x_t - \hat{x}_t$. Conventionally, to detect anomalies, the data of interest is fetched from the database and then based on the pre-calculated ARMA model, and points whose residuals exceed a certain threshold are labelled anomalies. This query can take a long time to run when the data is large, and the ARMA residual calculation will be redundant when the same query is asked multiple times.

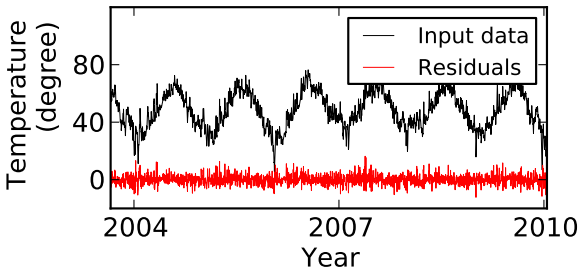
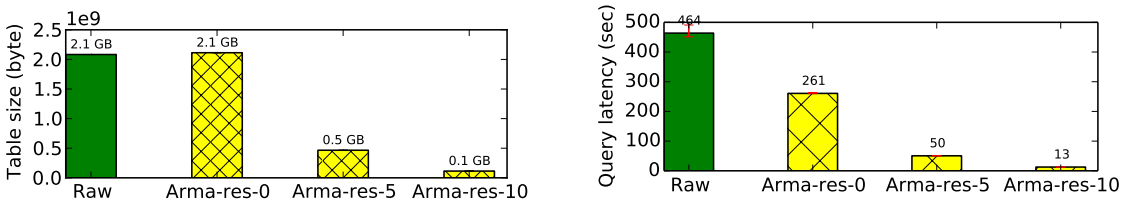


Figure 5: Temperature data collected at one station with their residuals calculated from the ARMA model.

Transformation. Instead, we can calculate the ARMA residuals using the `arma-res` transformation at the ingest-time. The transformation fits a window of data into a given ARMA model and writes the residuals as the output. The ARMA model parameters can either be specified as transformation parameters or be stored as a file in the database server, allowing updating the model at runtime. In order to reduce the size of the transformed data and improve query performance, the `arma-res` transformation takes in a `threshold` parameter and only writes residuals larger than `threshold` as the output.

Dataset and results. We evaluate the anomaly detection use case on the the same climatic dataset as the correlation search use case. We use the same table schema {`metric, station-name, time, value`} and interpolate the raw data in the same way. Before ingesting the data, we train one ARMA model for each of the metrics for every station, and store the model parameters in an in-memory file system. Each ARMA model is configured with 2 autoregressive terms and 2 moving-average terms, trained from two years of data. Figure 5 is an example of the residuals calculated on the temperature data from one station. We define anomalies as the data points with $abs(residuals) > 10$, and the testing query asks for all such anomalies. In this experiment, we generate three `arma-residual` transformations, with the threshold being 0, 5, and 10 respectively, and compare their performance with the raw data baseline.

Figure 6 summarizes the sizes of different tables and the latencies of doing the test query using them. The size of `Arma-res-0` is very close to the size of `Raw`, because we store the residuals of all data points, so both tables have the same number of rows. However, the latency of the query using `Arma-res-0` is lower than that using `Raw`, because the query using `Raw` needs to do the additional work of calculating the residuals. With larger thresholds used, both the data sizes and the query latencies drop, because fewer residual rows are kept as the transformed data. The query program can pick the transformation with the largest residual threshold that satisfies the application requirement.



(a) Table size

(b) Query latency

Figure 6: Anomaly detection results using transformed data vs. using raw data. `Arma-res-0`, `Arma-res-5`, and `Arma-res-10` are transformed data with residual threshold being 0, 5, and 10 respectively.

For example, when we use `Arma-res-10` to find all points with residuals larger than 10, there will be no query error, but the query latency is only 13 seconds, 2.8% of the baseline of 464 seconds.

The ingest time without doing any transformations is still 2629 seconds. With the transformation performed, the ingest time is 2848 seconds, which is 8% overhead.

5.3 Monitoring event occurrences

In addition to numeric data analysis, we show that Aperture provides orders of magnitudes speed up for non-numeric data analysis, by presenting the event monitoring use case.

One common approach of detecting anomalous events (e.g., security attacks from an IP address) is to monitor the occurrences of events in logs. For example, a suspicious burst of visits from one IP address is likely to be an attack. A natural way of counting events is to upload the log files to a database and use SQL queries to count the number of times a certain event occurs:

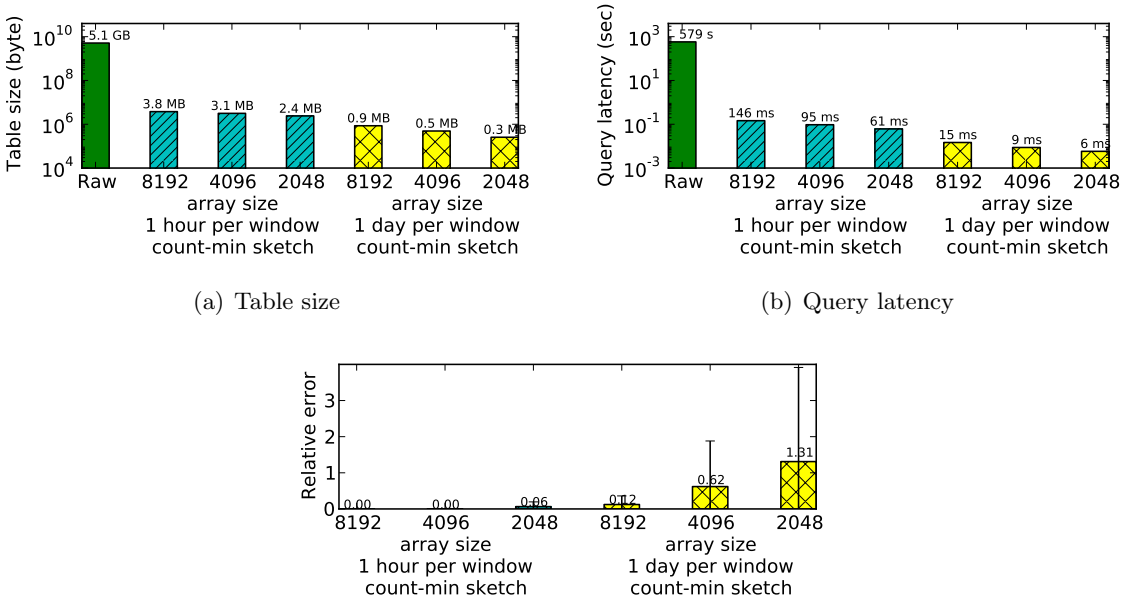
```
SELECT COUNT(*)
FROM event-log
WHERE ip='the-suspicious-ip'
```

However, this query can take a long time to run if the log contains billions of entries. Instead, Aperture uses approximate methods, such as count-min sketch [18, 23], to store data in a compact form, reducing query response time.

Count-min sketch. Mathematically, count-min sketch stores the counts of all items in d counter arrays $\{C_i\}_{i=1}^{d}$, each with size n. All counters are initially set to zero. For each incoming item s, it calculates d hash values $\{hs_i\}_{i=1}^{d}$ using independent hash functions $\{f_i\}_{i=1}^{d}$, and increases the corresponding hashed counters $\{C_i[hs_i \mod n]\}_{i=1}^{d}$ by one. To find the count of a given item t, it calculates the hash values $\{ht_i\}_{i=1}^{d}$, and the result is the minimum count among the hashed counters $\min_{i=1}^{d} C_i[ht_i \mod n]$. The count calculated from the sketch is not 100% accurate because of hash collisions, but is an upper bound to the true count.

Dataset. We evaluate Aperture on a public dataset collected by Measurement Lab [2]. This dataset is an event log that records the IP addresses visiting the Measurement Lab service. We used a log with 800 million entries spanning 8 days. We extract the time and IP address fields from the dataset and make a row schema of {`time, IP`}. We group log entries in the same day as one database upload batch, for a total of 8 batches.

Transformation. A `count-min-sketch` transformation is defined in Aperture with configurable number of sketch arrays d and sketch array size n. The transformation windows are created using the `window-by-value` option, grouping events in the same time range into one transformation



(a) Table size

(b) Query latency

(c) Query error

Figure 7: Occurrence count results using transformed data vs. using raw data.

window. The sketch of each window is stored as a byte array in a single row in the transformed table. In this experiment, we generate six `count-min-sketch` transformations by varying the window size (1 day and 1 hour) and sketch array length (2048, 4096, and 8192). We set the number of sketch arrays to 4 for all six transformations.

Results. Figure 7(a) compares the sizes of the transformed data with raw data (in log scale). Using count-min sketch, the size of each transformed table is less than 0.1% that of the raw table, so it does not incur significant storage overhead to store multiple versions of transformed data.

To measure the effect on query latency and query accuracy, we use the following workload: each query requests the total number of occurrences of one IP address over all 8 days. Figure 7(b) shows the response time of running one such query (in log scale). The query latency is greatly reduced when we use the transformed data, because of the vast reduction in the amount of data accessed and processed to answer the query. The query latency increases when smaller window granularities or longer sketch arrays are used.

Figure 7(c) summarizes the average error of the query results (count of one specific IP). We consider the query result from the raw data as ground truth, and define *query error* as $\frac{|count - true\_count|}{true\_count}$. We randomly pick five different IP addresses to query, and measure the mean and standard deviation (the error bars in the graph) of the query errors. The results show that the query error decreases with more data being used (smaller windows or longer sketch arrays). When we use a window size of one day and a sketch array length of 8192, for example, the query error is only 12%, but the query latency is reduced from 579 seconds to an impressive 15 milliseconds (40,000× faster). An error of 12% is totally acceptable for this use case, because one can confidently identify IP addresses with heavy visits (e.g., more than a thousand times within the last hour) using this approximation.

We also measure the ingestion overhead of calculating the sketches. In our experiments, the total time to ingest 8 batches (800 million rows, 100 GB in total), without transformations, is 6249 sec (128 thousand rows per second). With six `count-min-sketch` transformations, the total ingest time increases only to 6482 sec (4% overhead), Since we have only 8 uploads, we are able to

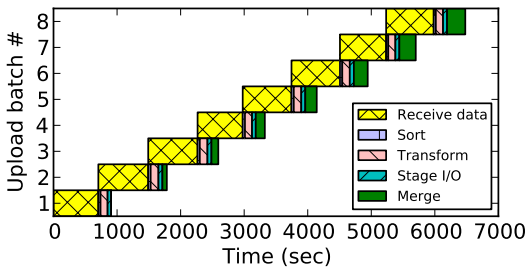
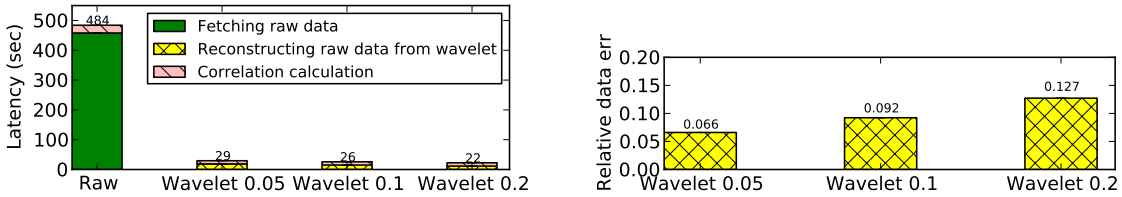


Figure 8: Ingest pipeline profiling for the occurrence count use case. This figure shows the amount of time that each upload batch spends at each pipeline stage, when we generate six `count-min-sketch` transformations.



(a) Query latency

(b) Relative Data error

Figure 9: Correlation search using reconstructed data.

summarize the amount of time that each upload spends at each pipeline stage in Figure 8, justifying the low ingestion overhead. The `receive` worker needs about 700 seconds to receive one batch from the client, while the `sort+transform` worker needs only about 200 seconds to sort and transform one batch, which means the `sort+transform` worker can finish its work before the reception of the next batch is completed.

5.4 Ad-hoc queries

Aperture can also process ad-hoc queries, i.e., arbitrary data analyses tasks. Whenever a query cannot be directly answered using transformed data, Aperture can automatically reconstruct original data from invertible transformations. To demonstrate this flexibility, we run experiments on the same wavelet transformations and climate dataset which was used in correlation search (Section 5.1). We execute a query on the full raw data (350 million rows) to calculate an arbitrary analysis. The wavelet transformations is calculated with a window granularity of 1 year and varying error bounds (from 5% to 20%).

Results. Figure 9(a) shows query latencies. We compare the time taken to load or re-create the raw data before the query can run. Raw data can either be fetched directly (green bar) or reconstructing from the transformed data (yellow bars). Even though reconstructing data from wavelet coefficients requires inverse wavelet transformation, it is much faster than fetching raw data. If we allow an error bound of 20%, we can reduce data preparation time from 458 seconds to 9.7 seconds (mere 2% of fetching raw data).

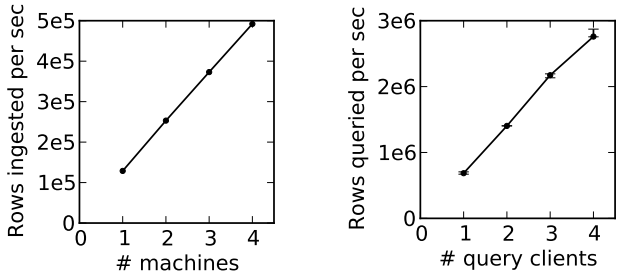
Figure 9(b) shows the error of the reconstructed data compared to using the original raw data. The error of timeseries a compared to timeseries b is defined as $\frac{\|a-b\|}{\|b\|}$. The result show that the combined error of all windows is close to (often smaller than) the error bound.

To measure the end-to-end query latency, we run the correlation search query (same as Section 5.1) on the reconstructed data. Since the reconstructed data is of the same size as the raw data, the time taken to calculate correlations is same in both cases, as shown by pink bars in Figure 9(a). After including the cost of calculating correlations, we notice considerable reduction in the overall query execution time, which decreases from 484 seconds when using raw data to 22 seconds when using reconstructed data. While reconstructing the raw data provides more flexibility, especially for ad-hoc analysis, it is slower than the case when a query can be directly run on transformed data (such as Figure 3(b)).

5.5 Scalability experiments

We examine the scalability of Aperture on the same datasets and transformations (one `interpolate`, six `wavelet`, and two `downsample`) as used in correlation search (Section 5.1). To evaluate performance scalability, we let Aperture run on N machines and launch three workers on each of them (one worker per pipeline stage). The machines are connected with 10 Gigabit Ethernet.

We first measure the scalability of our ingest processing. We duplicate our dataset into N copies and launch one client on each machine to upload one copy of the data. Each client uploads data to a separate table, and each table is transformed in the same way as the correlation search use case described in Section 5.1. This setup simulates the case where the data is uploaded from multiple sources. We change N from 1 to 4, and measure the ingest throughput of Aperture in terms of the number of rows ingested/transformed per second. Figure 10(a) shows that Aperture scales almost linearly with more machines added. For example, we get 3.8× more throughput using 4 machines, because Aperture transforms each batch of uploaded data in parallel.



(a) Ingest throughput (b) Query throughput

Figure 10: Aperture scalability.

We also measure our query processing scalability by having multiple clients doing queries concurrently. For this experiment, we run Aperture on four machines and change the number of query clients from 1 to 4. Each client runs on one of the four machines and issues a range query to fetch all the `Raw` data from one table. We measure the query throughput by dividing the total number of fetched rows by the time it takes for all the clients to finish. Figure 10(b) shows the result. For each setting, we do three runs with clients querying different tables. The markers and error bars show medians, maximum values, and minimum values. The result shows that the query throughput scales almost linearly when we run multiple query clients.

6 Conclusions

In this paper, we present Aperture, a framework that leverages ingest-time transformations to enable efficient time series analytics queries. Users define transformations on the data, which operate at ingest time to create succinct summaries of incoming data. At query time, user queries are automatically translated into queries on the most suitable transformed data.

This approach provides numerous benefits. Ingest-time transformations permit query-time speedups for important queries, by leveraging the fact that transformation outputs are typically much smaller than the raw data and that they pre-compute some of the query-time work. Additionally, the transformation outputs are compact enough that the summaries for many windows can be maintained in memory, thus avoiding the costs of accessing slow storage at query time, and permitting efficient queries to a longer history. Transformations can be calculated without significant overhead beyond the ingestion of untransformed data. Finally, due to the characteristics of many time series metrics, queries on transformed data often don't sacrifice query accuracy to obtain good performance.

We demonstrate the benefits of our approach using three case studies: event occurrence monitoring, correlation search, and anomaly detection. We observe reductions in query latency of one to four orders of magnitude for transformations with up to 10% ingest overheads and 20% query error.

A Wavelet Transform for Correlation

As is discussed in Section 5.1, the wavelet transform decomposes a signal x of length N into the linear combination of N wavelet basis functions. We denote the wavelet coefficients of x by $X = dwt(x)$. The wavelet transform is an orthonormal transform, i.e., the inner product of the wavelet basis functions is 0. Therefore, the inner product of two signals x and y is the same as the inner product of their wavelet coefficients:

$$\sum_{i=1}^{N} x_i y_i = \sum_{i=1}^{N} X_i Y_i \tag{2}$$

The correlation of two signals x and y can be calculated from their wavelet coefficients. Let $\tilde{x} = x - \bar{x}$ and $\tilde{X} = dwt(\tilde{x})$, and likewise for y. Then following (1) and (2), it can be verified that

$$corr(x, y) = \frac{\sum_{i=1}^{N} \tilde{X}_i \tilde{Y}_i}{\sqrt{\sum_{i=1}^{N} \tilde{X_i}^2} \sqrt{\sum_{i=1}^{N} \tilde{Y_i}^2}} \tag{3}$$

we can achieve a compact representation of the signal by retaining only the large wavelet coefficients.

From among the many choices of wavelet basis functions, we use the Haar wavelet [5, 16] in our paper. The Haar wavelet has a property that for any constant offset c, the wavelet coefficients of x and $x + c$ differ only in the first coefficient, which is determined only by the mean and the length of the signal (Equation 4).

$$X_1 = \sqrt{N}\bar{x} \tag{4}$$

Using the above two properties, we can calculate $\tilde{X}$ from X as

$$\tilde{X}_i = \begin{cases} 0 & i = 1 \\ X_i & i \neq 1 \end{cases} \tag{5}$$

Using Equation 5 and 3, we can calculate the correlation coefficient of x and y using their Haar wavelet coefficients X and Y.

Retaining only the large coefficients. Since the wavelet transforms of many real world signals are sparse, we can approximate the original signal using just a few large coefficients. The number of retained coefficients is controlled by a *transformation error bound* parameter. For any set $S \subseteq \{1, 2, \ldots, N\}$, let $X^{\{S\}}$ be the wavelet representation, where only the coefficients corresponding to indices in S are retained. Formally,

$$X_i^{\{S\}} = \begin{cases} X_i & i \in S \\ 0 & else \end{cases} \tag{6}$$

The *transformation error* of $X^{\{S\}}$ in retaining the coefficients specified by S is

$$err(X^{\{S\}}) = 1 - \frac{\sum_{i=2}^{N} (X_i^{\{S\}})^2}{\sum_{i=2}^{N} X_i^2} \tag{7}$$

For each window of time series data x, the `wavelet` transformation with an *errbound* will retain the minimal number of wavelet coefficients such that $err(X^{\{S\}}) < errbound$. The sparse $X^{\{S\}}$ vector can be stored as an array of {`coef-id, coef-val`} pairs in the transformation output.

Combing multiple windows. Our query program can combine the wavelet coefficients of multiple contiguous windows to answer correlation queries over windows whose length is a multiple

of the transformation window size. Suppose the transformation window size is N, and the queried data series x has length KN, consisting of K windows: $x = [x^{(1)}, ..., x^{(K)}]$. The correlation of two such signals $corr(x, y)$ can be computed from the windowed wavelet coefficients $X^{(j)} = dwt(x^{(j)})$ and $Y^{(j)} = dwt(y^{(j)})$. Let $\tilde{x} = x - \bar{x}$ and $\tilde{x}^{(j)} = x^{(j)} - \bar{x}^{(j)}$ The inner product between $\tilde{x}$ and $\tilde{y}$ can be written as

$$\tilde{x} \cdot \tilde{y} = N \sum_{j=1}^{K} \left(\frac{X_1^{(j)}}{\sqrt{N}} - \bar{x} \right) \left(\frac{Y_1^{(j)}}{\sqrt{N}} - \bar{y} \right) + \sum_{j=1}^{N} \sum_{i=2}^{N} X_i^{(j)} \cdot Y_i^{(j)}.$$

We can similarly calculate $\tilde{x} \cdot \tilde{x}$ and $\tilde{y} \cdot \tilde{y}$ and thus the correlation coefficient of x and y.

References

[1] https://cloud.google.com/bigquery/docs/dataset-gsod.

[2] https://cloud.google.com/bigquery/docs/dataset-mlab.

[3] Apache HBase. http://hbase.apache.org/.

[4] Apache Storm. http://storm.apache.org/.

[5] Haar wavelet. http://en.wikipedia.org/wiki/Haar_wavelet.

[6] How Twitter monitors millions of time series. http://radar.oreilly.com/2013/09/how-twitter-monitors-millions-of-time-series.html.

[7] HP Vertica Live Aggregate Projections. http://www.vertica.com/2014/07/01/live-aggregate-projections-with-hp-vertica/.

[8] OpenTSDB. http://opentsdb.net/.

[9] Pearson's product-moment correlation coefficient. http://en.wikipedia.org/wiki/Pearson_product-moment_correlation_coefficient.

[10] L. Abraham, J. Allen, O. Barykin, V. Borkar, B. Chopra, C. Gerea, D. Merl, J. Metzler, D. Reiss, S. Subramanian, et al. Scuba: diving into data at Facebook. *Proceedings of the VLDB Endowment*, 6(11):1057–1067, 2013.

[11] R. Agarwal, A. Khandelwal, and I. Stoica. Succinct: Enabling queries on compressed data. NSDI, 2015.

[12] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 29–42. ACM, 2013.

[13] E. Anderson, M. Arlitt, C. B. Morrey III, and A. Veitch. Dataseries: an efficient, flexible data format for structured serial data. *ACM SIGOPS Operating Systems Review*, 43(1):70–75, 2009.

[14] P. J. Brockwell and R. A. Davis. *Time series: theory and methods*. Springer Science & Business Media, 2009.

[15] K.-P. Chan and A. W.-C. Fu. Efficient time series matching by wavelets. In *Data Engineering, 1999. Proceedings., 15th International Conference on*, pages 126–133. IEEE, 1999.

[16] C. K. Chui. *An introduction to wavelets*, volume 1. Academic Press, 2014.

[17] J. Cipar, G. Ganger, K. Keeton, C. B. Morrey III, C. A. Soules, and A. Veitch. LazyBase: Trading freshness for performance in a scalable database. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 169–182. ACM, 2012.

[18] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[19] L. George. *HBase: the definitive guide*. " O'Reilly Media, Inc.", 2011.

[20] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *VLDB*, volume 1, pages 79–88, 2001.

[21] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra. Locally adaptive dimensionality reduction for indexing large time series databases. *ACM SIGMOD Record*, 30(2):151–162, 2001.

[22] F. Korn, H. V. Jagadish, and C. Faloutsos. Efficiently supporting ad hoc queries in large datasets of time sequences. *ACM SIGMOD Record*, 26(2):289–300, 1997.

[23] S. Matusevych, A. Smola, and A. Ahmed. Hokusai-sketching streams in real time. *arXiv preprint arXiv:1210.4891*, 2012.

[24] G. Reeves, J. Liu, S. Nath, and F. Zhao. Managing massive time series streams with multi-scale compressed trickles. *Proceedings of the VLDB Endowment*, 2(1):97–108, 2009.

[25] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[26] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al. Storm@ Twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.

[27] M. Widenius and D. Axmark. *Mysql Reference Manual*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 2002.

[28] Y.-L. Wu, D. Agrawal, and A. El Abbadi. A comparison of DFT and DWT based similarity search in time-series databases. In *Proceedings of the ninth international conference on Information and knowledge management*, pages 488–495. ACM, 2000.

[29] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, pages 10–10. USENIX Association, 2012.

[30] Y. Zhu and D. Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 358–369. VLDB Endowment, 2002.