AGAMI: Scalable Visual Analytics over Multidimensional Data Streams

Mingxin Lu, Edmund Wong, Daniel Barajas, Xiaochen Li, Mosopefoluwa Ogundipe,
Nate Wilson, Pragya Garg, Alark Joshi, and Matthew Malensek
Department of Computer Science
University of San Francisco
San Francisco, CA, USA
{mlu18,ewwong2,djbarajas,xli149,mogundipe,nmwilson,pgarg,apjoshi,mmalensek}@usfca.edu

***Abstract*—As worldwide capability to collect, store, and manage information continues to grow, the resulting datasets become increasingly difficult to understand and extract insights from. Interactive data visualizations offers a promising avenue to efficiently navigate and gain insights from highly complex datasets, but the velocity of modern data streams often means that precomputed representations or summarizations of the data will quickly become obsolete. Our system, AGAMI, provides live-updating, interactive visualizations over streaming data. We leverage in-memory data sketches to summarize and aggregate information to be visualized, and also allow users to query future feature values by leveraging online machine learning models. Our approach facilitates low-latency, iterative exploration of data streams and can scale out incrementally to handle increasing stream velocities and query loads. We provide a thorough evaluation of our data structures and system performance using a real-world meteorological dataset.**

***Index Terms*—Streaming visualization, big data visualization, in-memory data sketches, online machine learning**

I. INTRODUCTION

Advancements in computing and storage capabilities have enabled organizations to accumulate voluminous datasets with higher resolutions and broader scopes. These datasets are comprised of multidimensional observations with several *features* of interest and span a variety of domains including atmospheric science, epidemiology, finance, marketing, autonomous vehicles, and so on. While extensive research has been carried out on how to manage these datasets at scale [1], [2], [3], [4], extracting insights in a timely manner remains a challenge; data exploration is often an iterative process and practitioners may not be aware of underlying patterns, trends, or properties of the dataset in advance. Consequently, visualization techniques can accelerate the analytics process by striking a balance between information density, interaction, machine learning, and human-in-the-loop.

Our system, AGAMI, was designed specifically for multidimensional streaming data. The system can scale out incrementally as needed based on stream velocity and computational demands, and targets low-latency visualizations to facilitate iterative analytics. We maintain a compact, distributed *sketch* of the data in main memory to avoid latencies associated with secondary storage media such as hard disk drives (HDDs). Additionally, the system monitors incoming multidimensional records to train and create predictive machine learning models to provide visualizations of potential future outcomes. Visualization parameters are specified through an interactive web user interface or via our custom, domain-specific query language. Queries may span both the past and future, in which case results are fused into a single visualization that includes uncertainty measures for predicted events. Given our focus on streaming data, the client user interface is dynamically updated — live and in real time — as new information becomes available.

A. Research Challenges

Resolving visualization queries efficiently at scale can be challenging due to unique data access patterns as well as visualization-specific concerns such as image resolution and perceptual scalability limits [5]. Other challenges include:

1) Streams are multidimensional, unbounded, and each incoming record is comprised of several dimensions.
2) Data exploration is an iterative process and requires low-latency query responses [6].
3) Due to the ever-changing nature of data streams, the system must have the ability to dynamically reconfigure its indexes and re-train predictive models at run time.
4) Given the size of the datasets at hand, the system must be capable of scaling out incrementally.

B. Research Questions

To guide our implementation of AGAMI, we developed the following research questions:

RQ1 *How can we support memory-resident, visualization-driven indexing and storage structures?*
RQ2 *How can we optimize for live, interactive visualization use cases without requiring significant client-side processing?*
RQ3 *How can we train and leverage machine learning models over streaming data, and fuse the past and future into a single, coherent visualization?*
RQ4 *How can we facilitate live, timely updates to the client UI as new data points are received?*

978-0-7381-2396-7/20/$31.00 ©2020 IEEE
DOI 10.1109/BDCAT50828.2020.00020

C. Overview of Approach

AGAMI is modular and based on a microservice architecture. The system consists of several distributed components that can be provisioned based on the use case(s) of the particular deployment (dataset size, ideal response time, hardware capabilities, etc.). Each component can be scaled up or down depending on stream velocity and/or query load. Two distributed, in-memory data sketches are deployed in a clustered setting to provide retrieval functionality: the *SpaceTime Sketch* and the *Predictive Sketch*, providing past and forecasted feature values for visualizations. The client-side user interface is interactive, web-based with live updates, and enables both targeted and iterative exploration over the data.

D. Contributions

AGAMI is designed to scale out to handle large datasets and hundreds of concurrent users while providing live-updating, interactive visualizations for analysis. Our SpaceTime Sketch was designed specifically for aggregating and indexing data for visualization purposes. Instead of building an index based on one or more primary keys with records encoded as tabular rows, unique SpaceTime Sketches are maintained for each feature that can be efficiently joined in parallel. This is well-aligned with real-world visualization use cases; instead of retrieving and filtering rows that contain every feature (which could number in the tens or hundreds), only the features of interest are retrieved directly and joined to produce a visualization. Our Predictive Sketch automates the creation of online machine learning models that seamlessly integrate with query results; we allow queries that span from the past into the future to produce a single, integrated visualization with uncertainty measures. Queries are defined either with our user-friendly web UI or with our domain-specific query language. Instead of producing a server-side raster image, a *visualization specification* provides instructions and metadata that the client uses to produce an interactive representation of the data.

II. Related Work

A. Progressive Analytics for Big Data

In the field of data visualization of massive datasets, there have been various research developments that facilitate real-time querying of big data [7], [8], cubes for data processing and interactive visualization [9], [10], and progressive analytics [11]. There have been recent contributions [12] that address challenges related to exploring high-dimensional data using progressive analytics. Turkay et al. [12] also provide a set of design recommendations for high-dimensional online analysis using PCA or clustering.

Interaction and Latency in Progressive Systems: Recently, there has been further exploration of the impact of latency [6] on an analyst's ability to use such progressive analytics-based systems. Zgraggen et al. [13] found that analysts were able to use progressive systems *effectively* when results were presented either "instantaneously" or "progressive." They measured the impact by exploring the "insight discovery rates and dataset coverage."

Incremental Visualization: Fisher et al. [14] presented the "sampleAction" system that facilitates exploration of petabyte sized data through the results of approximate visualizations based on increasingly larger sample queries for fast analysis. Rahman [15] use "online sampling-based schemes" to allow fast exploration of data as it updates visualizations. They find and highlight "salient" features in the data earlier in the process and fill in more details incrementally.

On-the-fly optimizations for interactive exploration: Satyanarayan et al. [16] introduced *Reactive Vega* that uses the declarative visualization paradigm to create dataflow graphs. These dataflow graphs are created and updated at runtime to optimize performance for interactive data analysis on streaming data. Battle et al. [17] demonstrate pre-fetching data "tiles" to facilitate seamless exploration for users with a lightweight client. Their system performs caching based on user exploration traits and data characteristics. Bikakis et al. [18] present a aggregation-based framework that facilitates hierarchical navigation / exploration of large streaming data. They introduce a "lightweight tree-based structured which can be efficiently constructed on-the-fly for a given set of data."

B. Systems and Algorithms

Synopsis [19] distributes hierarchical, tree-based summary sketches as a "forest of trees" across processing resources to enable scalable query-based analysis of streaming data. While the partitioning scheme in Synopsis is well-suited for analyzing arbitrary spatiotemporal bounds, querying individual features in aggregate for visualization requires broadcasting to the entire cluster and traversing each tree to locate relevant data.

Wasay et al. [20] explore the possibility of caching "basic aggregates" to facilitate rapid exploratory statistical analysis. Basic aggregates are stored in an in-memory data structure and are reused for common statistical measures.

Most systems designed for storing large datasets, such as HDFS [1] and Apache Cassandra [2] persist data to secondary storage, i.e., hard disk drives. In-memory, distributed storage platforms including Redis [21] or MongoDB [22] generally represent records using predefined data structures with full resolution; in contrast, AGAMI is an in-memory, distributed store of visualization-specific aggregates. This avoids the high latencies associated with hard disk I/O while conserving memory.

Many distributed batch processing systems are well-suited for producing offline visualizations over petascale datasets, such as Hadoop [23], Spark [3], or Flink [4]. Both Spark and Flink also provide streaming capabilities, allowing visualizations to be produced over collections of records (windows) or when certain conditions are met. Combined with a computational notebook and declarative DataFrame API (e.g., Jupyter Notebooks and Spark SQL), users can query and visualize their data interactively. However, a limited amount of full-resolution records can be stored in memory by these systems, and the client hosting the interactive notebook often must perform significant processing to produce the final visualization.

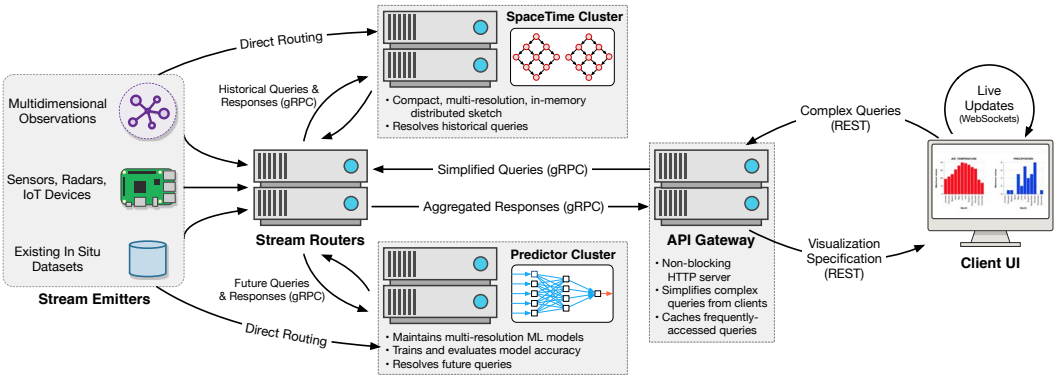

Fig. 1. AGAMI architecture. *Emitters* represent stream sources, which are consumed by *Stream Routers*. Stream routers partition incoming records based on the features they contain, and forward the feature values to their respective *aggregators*. After configuring the stream, emitters can optionally route directly to their destination aggregators to improve performance. Aggregators include the *SpaceTime* and *Predictor* clusters, which resolve historical and future queries, respectively. Clients generate *complex* queries that are *simplified* and forwarded to relevant aggregators for resolution, combined into a single response at the stream routers, and returned as a visualization specification to the client.

III. METHODOLOGY

AGAMI is composed of several distributed components, shown in Figure 1 along with their data and query flows. Streams are produced by *emitters* that drive data collection from sources such as sensing equipment, click stream data, stock prices, IoT (Internet of Things) devices, etc. These records are received by a cluster of one or more *stream routers*, which are responsible for passing the data to appropriate downstream *aggregators* via a configurable partitioning algorithm. Multidimensional data points are split and sent to a subset of the aggregator nodes; for example, all temperature values may be managed by a particular subset of machines.

Our current implementation includes two aggregators: the *SpaceTime Sketch* and *Predictive Sketch*. The SpaceTime Sketch is responsible for indexing feature values, their spatial location, and time of collection. These values are stored in a compact, memory-resident structure that can quickly resolve summarization queries at multiple resolutions. The predictive sketch, on the other hand, builds machine learning models to allow the system to resolve queries that span into the future. Consequently, raw data is not stored on predictive nodes once it has been incorporated into relevant models. Both sketches are deployed in a clustered configuration that can be scaled to meet storage and query demands.

On the client side, our interactive UI sends queries to the REST *API Gateway*, which coordinates backend query operations, with WebSockets used to trigger live updates. Queries are sent to the stream routers and redirected to appropriate aggregator nodes via the same partitioning algorithm used for storage. Queries submitted through the client UI are initially represented as *complex queries*, which are decomposed into multiple *simplified queries* before being processed in parallel by relevant aggregator nodes. The UI displays both historical and future query results on the same chart, shown in Figure 2 with future queries (the bottom three) displayed in varying opacity depending on the uncertainty of the predictions.

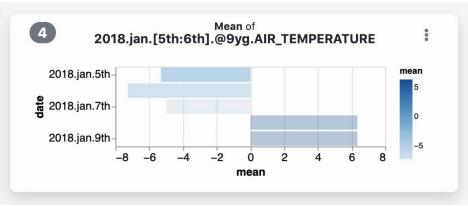

Fig. 2. Mean air temperature from Jan 5th to 9th, 2018 with predictions 3 days into the future, shown with reduced opacity to reflect uncertainty.

The AGAMI network operates over TCP sockets and can be configured to use either JSON or Protocol Buffers [24] for its data interchange format. Both the stream router and the SpaceTime Sketch cluster support partial deserialization functionality backed by Protocol Buffers. When serialized messages arrive, only the features used for data partitioning and routing are deserialized. While this enforces a strict structure to the messages sent over the wire, it speeds up deserialization, especially with datasets that have a large number of individual features. Queries managed by the API Gateway are handled by the gRPC framework. The system is implemented in Python 3, and each component uses a pool of processes to allow for concurrent processing of requests.

A. Stream Emitter

Stream emitters ingest data into the system concurrently from multiple sources. Emitters can be any form of data sources: sensing equipment, IoT, or even Apache Kafka [25] topics. AGAMI standardizes input streams by requiring two metadata: The UTC timestamp when the observation was recorded, and its location expressed as $< longitude, latitude >$ tuples. Any number of features can be included in the data source, but only those that are configured to be consumed by the stream router will be captured and forwarded to downstream aggregators. Note that AGAMI does not capture the time when the system receives the record; only the UTC timestamp will be used as the event recording time. This property is particularly important when dealing

with a variety of geographically-distributed data sources that may include a range of different devices, and implies that stream sources do not have to emit records chronologically; an in situ dataset stored on disk could be emitted in reverse chronological order and the end result would be the same as if it was emitted chronologically.

After initial stream configuration has taken place with a stream router, emitters can optionally employ *direct routing* to transmit records to aggregator nodes. This requires more state information to be maintained by the client, but improves performance by avoiding bottlenecks at the stream routers. Changes to the routing table tend to be infrequent, and aggregator nodes will only accept direct transmissions from clients that are operating on the same version of the routing table. Clients with outdated routing tables or hardware that is unable to maintain network state efficiently (such as IoT devices) can fall back on using the stream routers.

B. Stream Router

The stream router handles distribution of incoming data points and client queries. Upon receiving a new record, the stream router deserializes it, extracts its individual features, and forwards the feature values to their destination aggregator(s). The stream router must deserialize and interpret the features in the input stream because we allow sparse records with only some of the features present, and we also support adding new features and aggregator nodes at run time. Multiple stream routers can be deployed at larger scales, in which case the routing table is synchronized across the cluster.

Data Partitioning: the stream router maintains a routing table that maps features in the input stream and their ranges of values to a cluster of aggregator nodes. When a new aggregator starts and registers itself with the stream router, it is assigned a feature with the smallest cluster size to maintain load balancing. Partitioning is based on numeric values or spatial coordinates.

Multi-Process Stream Handlers: input streams are handled by a pool of processes, with incoming features mapped to an outgoing queue associated with each destination aggregator. Mappings are updated and transmitted to the stream handlers each time a new aggregator registers with the stream router.

Query Processing: queries are forwarded to their appropriate clusters based on the routing table. Complex queries generated by clients often translate to large amount of simple queries, so the stream router pre-computes a list of queries for each aggregator node and submits them in batches. This improves performance because only a single trip is required to collect query results, making the query latency less sensitive to network overhead.

C. SpaceTime Sketch

A cluster of SpaceTime Sketch instances provides indexing functionality in AGAMI. The SpaceTime Sketch maintains aggregated statistics of incoming records at multiple resolutions. The sketch is structured as a directed acyclic graph (DAG) where each vertex in the DAG holds spatiotemporal metadata and an online running statistics object calculated via Welford's method [26]. The spatiotemporal metadata is encoded from the UTC Timestamp, Longitude, and Latitude in each input record, and it is structured hierarchically as year, month, day, hour, minutes, and Geohash [27]. Geohash is hierarchical Base-32 encoding scheme that represents two-dimensional $< longitude,\ latitude >$ pairs as one-dimensional strings. Longer strings refer to higher-precision spatial bounding boxes, and hashes that share similar prefixes are spatially proximate.

Structuring temporal metadata as year, month, day, hour, minutes provides a natural way to organize multiple levels of temporal resolution. Similarly, encoding longitude and latitude with the Geohash algorithm provides multiple spatial resolution by using different Geohash lengths. The topological ordering of the DAG requires that a vertex can only be pointed to by its parent vertex if and only if it inherits the parent's coordinate and expands its resolution either spatially or temporally by one level. Figure 3 shows a DAG built from two records (left side, white vertices were inserted first, followed by the right side, green vertices). Here, the two records share temporal nodes to conserve memory.

During insertion, our algorithm extracts spatial and temporal data from the input record and traverses the DAG to update statistics of the relevant vertices. Specifically, the algorithm starts from the spatial and temporal root of the DAG, updates child vertices that are within the spatial temporal scope of the new records, and creates new vertices and edges if they do not exist. The DAG is implemented using an adjacency list to store all vertices in the form of spatiotemporal coordinates. See Algorithm 1 for the recursive insertion algorithm.

Algorithm 1: Recursive Insertion

Input: Insertion value val, Insertion Geohash gh, Insertion time t ($year$, $month$, day, $hour$), and current vertex v (gh, t, and s for statistics)
let $under_t(t_1, t_2)$ = a function that return whether t_1 equals and extends resolution of t_2
let $under_{gh}(gh_1, gh_2)$ = a function that return whether gh_1 equals and extends resolution of gh_2
$v.s \leftarrow update(v.s, val)$
if $t \neq v.t$ ***and*** $under_t(t, v.t)$ **then**
  $t_{new} \leftarrow v.t$ with 1 more level of resolution from t
  if $(gh, t_{new}) \notin$ *DAG* **then**
    add vertex (gh, t_{new}) to DAG
  $v_{new} \leftarrow (gh, t_{new})$
  $Insert(val, gh, t, v_{new})$
if $gh \neq v.gh$ ***and*** $under_{gh}(gh, v.gh)$ **then**
  $gh_{new} \leftarrow v.gh$ with 1 more level of resolution from gh
  if $(gh_{new}, t) \notin$ *DAG* **then**
    add vertex (gh_{new}, t) to DAG
  $v_{new} \leftarrow (gh_{new}, t)$
  $Insert(val, gh, t, v_{new})$

This approach was selected due to several beneficial prop-

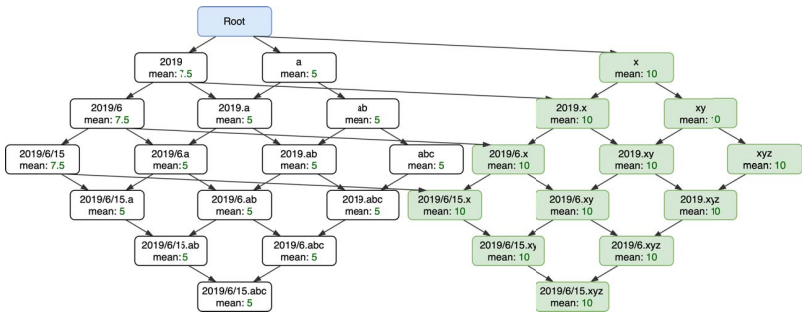

Fig. 3. DAG constructed from two records on Jun 15th 2019, one with a value of 5 (left side, white vertices) and a second record with a value of 10 (right side, green vertices) at two different Geohash locations: "abc" and "xyz." Notice the shared temporal nodes are updated with both statistics.

erties. Expanding a single record into multiple pre-aggregated spatiotemporal nodes avoids the need to perform aggregations on queries, which reduces latency. The edges between parent vertices and child vertices specify all spatiotemporal coordinates with one level of resolution higher than the parent vertex. This information is attached to each query result to provide insight on the distribution of higher resolution vertices that aggregate into the queried vertex, and it also enables the data-exploration feature "step into/out of time" and "step into/out of space" (discussed in detail in Section III-E). The edges also make aggregation possible with queries that skipped one or many intermediate resolutions. For example, a query for air temperature in 2018 on the 1st day of the month at 11pm will apply to **all** months in 2018 because a particular month was not specified. This is done by recursively traversing all vertices from 2018 with a wild card month that matches all months, but is still constrained to the 1st day of each at 11pm.

The size and the density of the data structure is dependent on the maximum level of resolution and the spatial and temporal complexity of the input stream, rather than the volume of the input stream. The maximum depth of the DAG, which determines the worst case traversal time, is the sum of maximum spatial resolution and temporal resolution. As an example, the DAG in Figure 3 has a depth of 6, temporal resolution of 3 (year, month, day), and spatial resolution of 3 (three Geohash characters).

Pruning: Tracking statistics at multiple resolutions allows the higher resolution vertices — which are often the majority of the DAG — to be pruned over time to preserve memory. Once pruned, the aggregated statistics can still be accessed from low resolution vertices, but fine-grained statistics will not be available. The amount of time that each level of resolution persists in memory is configurable (e.g., keep the last 10 days of hourly data, a year of monthly data, a year of 4-character Geohashes, etc.). The pruning algorithm routinely traverses the DAG to remove vertices that exceed these limits. If a particular vertex is selected for removal, its children can also be safely eliminated. The pruning algorithm runs in $O(|V| + |E|)$.

D. Predictive Sketch

Similar to the SpaceTime Sketch, our Predictive Sketch operates on data that has been partitioned by the stream router. This produces clusters of records that share similarity based on numeric, 1-dimensional feature values or two-dimensional spatial locations backed by the Geohash algorithm. Each *predictor node* maintains online machine learning models that are trained on the node's particular subset of the data. Predictor nodes are organized in a hierarchy to allow specialization and create an *ensemble* of models that can be leveraged either individually or as a group to combine predictions with a Mixture of Experts (ME) approach [28]. This is often advantageous when data is partitioned geographically; for instance, a top-level model could train on data for the US state of California, while finer-grained models specialize on data from San Francisco and Los Angeles.

Indexing: One or more features can be used to index records for predictions. When indexing spatial data, each predictor node maintains one or more trie data structures of Geohash strings to describe the spatial locations under their purview. The Geohash precision of the root of each tree is configurable and set to four characters by default to roughly represent the size of an average administrative boundary around a small state/province or large city. In the case of general feature values, records are placed in *feature buckets* that represent ranges of values with subsequent levels in the trie hierarchy increasing the bucket precision by decreasing the ranges of values maintained by each vertex.

Training: The predictive sketch allows several online machine learning models to be trained at each vertex in the trie for forecasting feature values. To ensure broad applicability, AGAMI can train most models available in the Scikit-Learn [29] and XGBoost [30] libraries (including Random Forests, Gradient Boosting, Linear Regression, Neural Networks, etc.). Model parameters and the dependent and independent variables for each model are defined in a configuration file to reflect use case and dataset characteristics. In the case of XGBoost models, the system will selectively employ GPU acceleration based on observed training times; more complex models are better suited to GPU acceleration but simpler models are trained on the CPU to avoid overheads associated with transferring data between main memory and the GPU.

Instead of updating the models immediately as each new record arrives, AGAMI collects small batches of new data

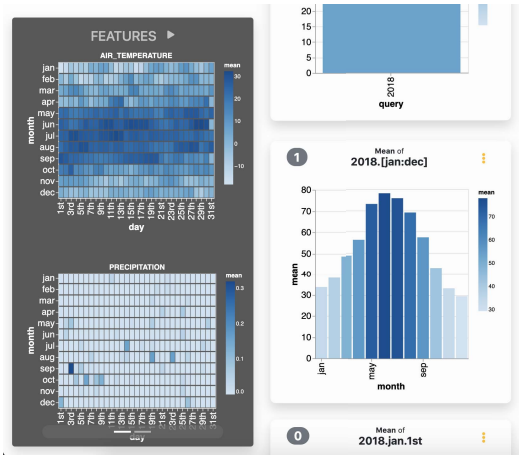

Fig. 4. The dashboard includes a live feature monitor on the left and pinned queries on the right.

points to amortize the CPU/GPU costs associated with training. In situations where the incoming stream velocity is low, prediction nodes use a *training time interval* to flush partial batches of data for training. This interval is set to 15 seconds by default, meaning that if a complete batch is not received over a span of 15 seconds the partial batch will be used to update relevant model(s). To ensure predictions remain relevant as the input data changes, old training data can be aged out based on a configurable threshold (e.g., removing training data that is over one year old).

In many datasets, records that are numerically or geospatially similar often influence one another. For instance, the weather in a particular location tends to influence other nearby weather patterns. We provide a configurable *neighborhood threshold* to incorporate these circumstances into our machine learning models to improve accuracy; models associated with a particular Geohash or feature bucket will also include data from surrounding Geohashes at the same resolution.

Making Predictions and Measuring Uncertainty: Once the models are trained, using them to make predictions is straightforward. When a query arrives, the predictor node determines the set of vertices with relevant models by extracting features from the query and traversing the trie. The models are then used to predict each of the future data points and the results are sent back to the stream router. Results are accompanied by an *uncertainty list* that reflects the confidence in the predictions; depending on the model, these values can be provided directly or calculated from a *test set* of raw data that evaluates how well the model can predict known values that were not included during training.

E. User Interface

Our backend infrastructure allows for interactive data visualizations with predictive data points to be generated in the web browser. The main considerations with displaying these visualizations was to support interactivity as well as streaming data. This resulted in two main pages: a *dashboard* and a *builder* page. The dashboard displays live data for each feature as well as pinned queries from the builder. The builder has an emphasis on data exploration and allows for quick query building as well as chart interactions that support exploring the data by "stepping into/out of" time and space with respect to the selected data point.

Dashboard: The dashboard in Figure 4 includes live updating feature charts. The right side of the interface includes all pinned queries from the builder. These pinned queries serve as a way to make specific comparisons or as a quick way to save queries. Pinned queries can also include future predictions that are displayed with reduced opacity proportionate to the confidence in the prediction.

Targeted Exploration: In terms of interactivity, the builder page supports targeted data exploration using complex queries. Complex queries are a way to condense potentially long lists of queries into shorter, more readable queries. With ranges and multiple selections, specific questions can be built into queries that are then displayed as charts. These queries also have support for wildcards meaning that the queries do not need to include every feature; e.g., spatial location can be excluded to get data from all areas. As an example, the query 2016.[feb:may].[1st:10th].@9q.Temperature retrieves temperature values from a particular year (2016), a range of months (February through May), specific days (the 1st through the 10th only) located at Geohash `9q`.

Iterative Exploration: The charts that are generated in the builder page include a context menu that can be accessed with a double click. This allows for more options including downloading the source specification for the given chart. To promote data exploration, four options are included: stepping into time, stepping out of time, stepping into location, and stepping out of location. This allows for ways to explore data beyond the original query. If the original data point selected is 2018.jan.1st.@9ygq, stepping into time would display a chart with all times on 2018.jan.1st. This would effectively be the same as the 2018.jan.1st.[12am:11pm].9ygq complex query. Stepping out of time from the original data point would give a query such as 2018.jan.@9yq. Note that the day has been removed from this query. Stepping into a location selects all sub-locations within the current Geohash range and increases the prefix string by one character, while stepping out removes one character. This allows users to be able to easily explore (step around) the data spatially and temporally. The supported chart interactions can be seen in Figure 7.

Chart Generation: The charts from these two pages are created using Altair [31], a declarative statistical visualization library. First, the API Gateway is passed a complex query and a chart type. This complex query serves as an abstraction of a list of base queries; the flow can be seen in Figure 6. A complex query may look something like 2018.[jan:dec].[1st:31st].@9ygq.Precipitation. This query is tokenized and parsed and then broken up into simple queries. In this example, the complex query would expand into 372 (12*31) simple queries with months spanning from January to December and days spanning from the 1st to the 31st. The API

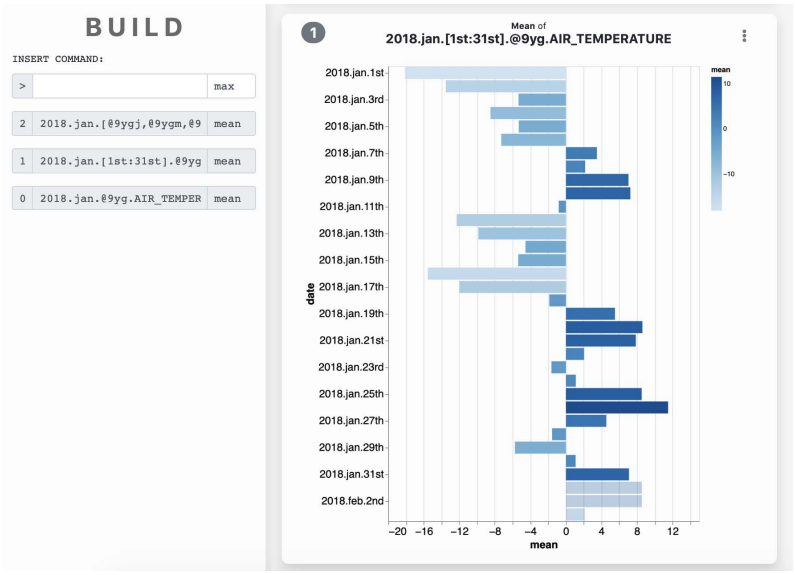

Fig. 5. The builder allows for targeted exploration using complex queries. The left column stores a history of queries. The right column shows the charts corresponding to the query. The charts in the right column also have support for "step into/out of" context options, the option to download the Vega-lite spec, and the ability to pin queries to the dashboard.

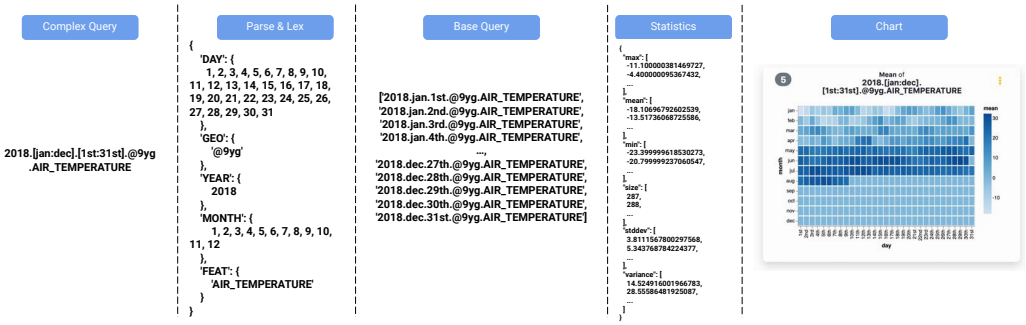

Fig. 6. This figure illustrates the flow from a complex query to the final chart. The user first inputs a complex query. This is parsed into different token types. Then a Cartesian product is used to generate base queries that are used to query the system. This results in statistics that are then used to generate the charts in the browser. Note that the query is dot separated.

Fig. 7. Builder charts support multiple forms of exploration: stepping in/out of Geohash and stepping in/out of time. The query 2018.jan.1st.@9ygq.Air_Temperature expands to 2018.jan.1st.[12am:11pm].@9ygq and 2018.jan.1st.[@9ygq6, @9ygq8, ...].Air_Temperature. The same query gives 2018.jan.1st.@9yg.Air_Temperature and 2018.jan.@9ygq.Air_Temperature when stepping out. These four charts give more context to the current query temporally and spatially.

Gateway then fetches all the results from the Stream Router. Note that these expanded simple queries include "impossible" dates like February 31st. The Stream Router will return empty results for such dates. With these results, the chart can now start to be rendered on the client side. Instead of generating the graphics at this point, Altair allows for a JSON specification to be built instead. This JSON specification is built using Vega-Lite [32], a high-level grammar for interactive data visualization.

Due to the nature of the query language, there is no guarantee that a specific part of the query can be used for the x- and y-axes. One option is to use the queried feature as the x-axis and its corresponding value(s) for the y-axis. However, this is not always easy to read so is used as a fallback. With the complex query abstraction, we can have one, multiple, or none of a given token: year, month, day, hour, feature, or Geohash. If we have a single range such as 2018.[jan:dec].@9ygq.Precipitation, it is likely that the user wants to inspect data from January to December. In this case, months would be much easier to read as an axis label. Other cases follow with a very similar logic: two ranges of values leads easily to a heat map based on each of the ranges. In this heat map, the color of the cells are encoded by the value. If no ranges are given, but there is an element related to a date, it will be used for the x-axis. In other cases such as three ranges, a bar chart is used with the query as the axis label. In all of these charts a sequential color scale is used.

When the Vega-lite chart specification has been created, the JSON is sent to the client browser using web sockets. The client then renders the chart with Vega-Embed [32] and displays the information with the Vue.js, a lightweight MVVM framework.

Query Caching: To optimize query latency and performances, simple query results from both the SpaceTime and Predictive sketches are cached on the API Gateway in a Time Aware Least Recent Used (TLRU) cache [33] with a 60-second timeout. The API Gateway only fetches results and predictions from the Stream Router if the input Complex Query translates to simple queries or prediction requests that result in cache misses. Caching simple query results and prediction results can drastically improve the query latency of subsequent queries by avoiding requests to the entire cluster.

IV. Experimental Evaluation

We evaluated AGAMI with the U.S. Surface Climate Observing Reference Networks (USCRN) dataset [34] from the National Climatic Data Center (NCDC). The dataset consists of climate measurements taken every 5 minutes from several locations within the United States. To aid in demonstrating the scalability and processing throughput of our system, the stream emitters in the following benchmarks were configured to produce data as fast as the network could support, i.e., faster than real time. Features in the dataset include UTC date, UTC time, precipitation, solar radiation, longitude, latitude, humidity, air temperature, and surface temperature. Our test

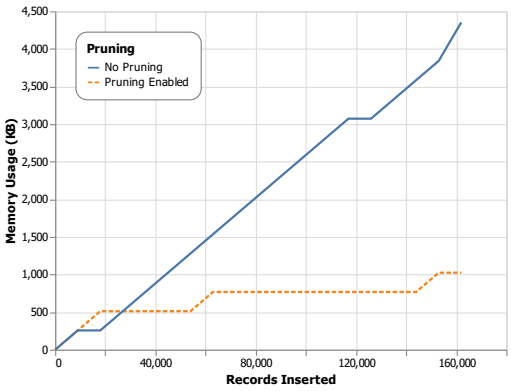

Fig. 8. SpaceTime sketch memory consumption. While the memory consumed by sketch remains low even with unbounded growth, enabling pruning of the DAG provides a predictable memory consumption profile.

environment consisted of 12 machines with 4-core Xeon E3-1230 v6 CPUs running at 3.5GHz, with 32GB of memory and 1 Gbps network. Benchmarks were carried out using CPython 3 on CentOS 7, Linux kernel 3.10.

A. Data Structure Microbenchmarks

Our methodology consists of several distributed components, so our first set of evaluations tested each data structure in isolation to minimize interference.

SpaceTime Sketch Memory Usage and Pruning: A key aspect of our approach is the compactness of the statistics maintained by the SpaceTime sketch. We evaluated memory consumption of one of the sketches in the cluster as records were streamed into to the system, along with a second iteration of the test with DAG pruning enabled. The results of this experiment are shown in Figure 8. Without pruning, memory usage is low (approximately 2.7 MB for 100,000 records on a single node) but grows as more vertices and edges are added to the DAG. Enabling pruning functionality allows for more predictable memory consumption over time, but also means that older records will be collapsed into higher-level vertices that are less accurate. This makes it possible to tune the SpaceTime sketch based on hardware capabilities as well as the accuracy requirements of the particular deployment.

Sketch Insertion Speed: While stream emitters and routers play a large role in overall system performance during ingestion, aggregator clusters must be able to process incoming records quickly to prevent placing backpressure on upstream components. We tested the SpaceTime and Predictive nodes in isolation to measure their ingestion speed. In the case of the SpaceTime sketch, this involves creating DAG nodes and edges along with running statistics instances, while Predictive nodes must collect batches of records and incrementally train machine learning models. The SpaceTime sketch was able to insert **988.8** records/s, and the Predictive sketch could ingest **490.9** records/s. Training machine learning models tends to be more CPU-intensive, but AGAMI can be configured to train less often to ease computational load. The scalability of our

design also means that in this case deploying two predictive nodes for each aggregator node would result in near-uniform insertion speeds.

Message Serialization Overhead: AGAMI can employ both JSON and Protocol Buffers [24] for serializing the messages transmitted between components. Our initial implementation used JSON, but migrating to Protocol Buffers provided a substantial decrease in message sizes while maintaining similar serialization/deserialization speed. To evaluate the performance differences between these two serialization formats, we monitored message sizes and serialization/deserialization times while streaming 300,000 records into the system. Table I summarizes the results of this experiment.

TABLE I
JSON AND PROTOCOL BUFFER SERIALIZATION OVERHEAD.

Performance	**JSON**		**Protocol Buffers**	
Metric	Mean	SD	Mean	SD
Message Size (bytes)	671.39	11.67	83.86	7.94
Serialization Time (μs)	395.88	6.30	389.76	23.18
Deserialization Time (μs)	7.82	0.53	25.40	3.91

In general, JSON-based communication tends to be easier to debug, so we allow both serialization formats to be selected in the AGAMI configuration. Note that while Protocol Buffers are used internally for communication between components in the system, visualization specifications are always represented in JSON format.

Prediction Accuracy: To evaluate our predictive sketch, we selected an XGBoost regressor trained to predict air temperature in Manhattan, Kansas. Features used were latitude, longitude, year, month, day hour, and sine/cosine of the day of the year (cyclic encoding to reflect that days near the end of the year are temporally close to days near the beginning of the next year). Parameters used for the model included a maximum tree depth of 8, minimum child weight of 0.001, shrinkage factor (η) of 0.5, and the default objective function (regression with squared loss). After parameter tuning, the MAE resulted in 17.59. Figure 9 shows 10,000 randomly-selected predictions made by the model; while other models produced by our system may be more or less accurate than this example, our visualizations always include uncertainty information (varied opacity) so users can judge how reliable the predictions are.

B. End-to-End System Benchmarks

Our second set of evaluations involved the two main entry points and data processing paths of AGAMI: (1) ingesting, processing, and aggregating data streams, and (2) parsing, resolving, and producing results for user queries.

Stream Ingestion Scalability: To ensure AGAMI scales as more resources are added to the system, we used 12 stream emitters to ingest our dataset and increased the number of aggregator nodes to scale out, for a maximum of 40 aggregator nodes on our test cluster (4 aggregators per machine distributed across 10 machines, with 3 machines hosting emitters). As

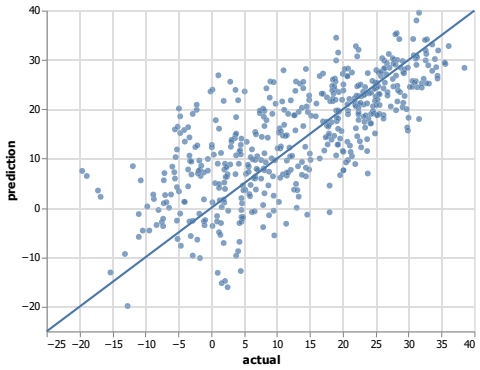

Fig. 9. Accuracy of an XGBoost regressor model predicting air temperature in Manhattan, Kansas. Features used were latitude, longitude, year, month, day hour, and day of the year.

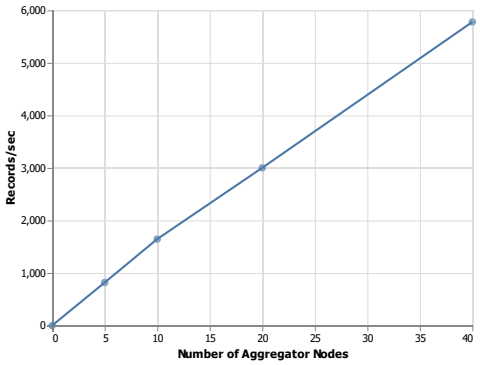

Fig. 10. Aggregation throughput (records ingested per second) as AGAMI scales out with more aggregator nodes.

shown in Figure 10, AGAMI scales out in an approximately linear manner, with minor overhead from distributed communications. Note that unlike our insertion microbenchmark, this experiment includes the full serialization, communication, and network overheads seen in a real-world deployment.

Query Latency: To test query performance, we compared our initial design (complex queries broken into many single queries) with batch querying. Queries were randomized and submitted to the API Gateway using 200 concurrent client threads. Combining queries and their results into batches reduces the amount of messages sent between components in the system, thereby reducing socket connection and serialization overhead, which led to a substantial improvement in response times. Figure 11 demonstrates these response times; the complete query operation took roughly 500 ms for all batch operations, but the latency of single queries increases linearly. We continued to test up to a 500-query batch — representing an approximate upper bound for query complexity generated by our UI — which completed in **625 ms** on average.

V. CONCLUSIONS AND FUTURE WORK

AGAMI addresses a particular niche in the world of big data: streaming, in-memory aggregation for live visualizations

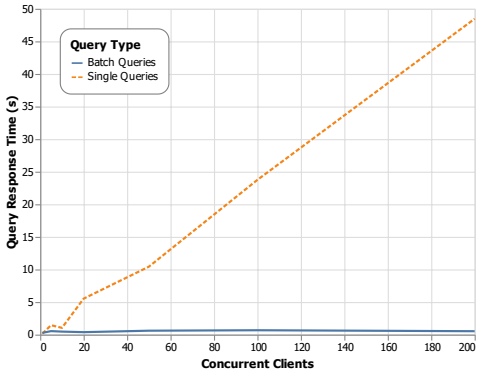

Fig. 11. Query response times tested with 200 concurrent clients for (1) complex queries broken up into single queries and submitted individually, and (2) batch queries. Combining queries and results into batches reduces communication, leading to faster response times and better throughput.

with integrated feature forecasting. Our SpaceTime Sketch is compact and allows us to store historical data in memory specifically for visualization (**RQ1**). As data is streamed into the system, we also train multiple machine learning models at a variety of resolutions to provide forecasting functionality through our Predictive Sketch. Queries can span across the past and future to produce a single, integrated visualization (**RQ3**). Our web-based client UI reacts to changes in the underlying dataset as new records arrive (**RQ4**) and our distributed data sketches enable us to produce visualization specifications for clients without sending them raw data or requiring excessive post-processing operations to occur (**RQ2**).

In our future work, we plan to incorporate more visualizations and add support for additional query types, such as "top-k" queries that find the k most frequently-occurring feature values. We plan on investigating additional ways to convey uncertainty associated with predictions or classifications, and add support for users to interact directly with partial visualization results to improve responsiveness.

References

[1] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. IEEE, 2010, pp. 1–10.

[2] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[3] M. Zaharia *et al.*, "Apache spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[4] P. Carbone *et al.*, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[5] B. Yost and C. North, "The perceptual scalability of visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 837–844, 2006.

[6] Z. Liu and J. Heer, "The effects of interactive latency on exploratory visual analysis," *IEEE transactions on visualization and computer graphics*, vol. 20, no. 12, pp. 2122–2131, 2014.

[7] Z. Liu *et al.*, "immens: Real-time visual querying of big data," in *Computer Graphics Forum*, vol. 32, no. 3pt4. Wiley Online Library, 2013, pp. 421–430.

[8] J. Koontz, M. Malensek, and S. L. Pallickara, "Geolens: Enabling interactive visual analytics over large-scale, multidimensional geospatial datasets," in *Proceedings of the 2014 IEEE/ACM International Symposium on Big Data Computing (BDC)*, Dec 2014, pp. 35–44.

[9] C. A. Pahins, S. A. Stephens, C. Scheidegger, and J. L. Comba, "Hashedcubes: Simple, low memory, real-time visual exploration of big data," *IEEE transactions on visualization and computer graphics*, vol. 23, no. 1, pp. 671–680, 2016.

[10] N. Kamat and A. Nandi, "A session-based approach to fast-but-approximate interactive data cube exploration," *ACM Trans. on Knowledge Discovery from Data (TKDD)*, vol. 12, no. 1, pp. 1–26, 2018.

[11] N. Pezzotti *et al.*, "Deepeyes: Progressive visual analytics for designing deep neural networks," *IEEE transactions on visualization and computer graphics*, vol. 24, no. 1, pp. 98–108, 2017.

[12] C. Turkay, E. Kaya, S. Balcisoy, and H. Hauser, "Designing progressive and interactive analytics processes for high-dimensional data analysis," *IEEE transactions on visualization and computer graphics*, vol. 23, no. 1, pp. 131–140, 2016.

[13] E. Zgraggen, A. Galakatos, A. Crotty, J.-D. Fekete, and T. Kraska, "How progressive visualizations affect exploratory analysis," *IEEE transactions on visualization and computer graphics*, vol. 23, no. 8, pp. 1977–1987, 2016.

[14] D. Fisher *et al.*, "Trust me, i'm partially right: incremental visualization lets analysts explore large datasets faster," in *Proceedings of SIGCHI*, 2012, pp. 1673–1682.

[15] S. Rahman *et al.*, "I've seen" enough" incrementally improving visualizations to support rapid decision making," *Proceedings of the VLDB Endowment*, vol. 10, no. 11, pp. 1262–1273, 2017.

[16] A. Satyanarayan *et al.*, "Reactive vega: A streaming dataflow architecture for declarative interactive visualization," *IEEE transactions on visualization and computer graphics*, vol. 22, no. 1, pp. 659–668, 2015.

[17] L. Battle, R. Chang, and M. Stonebraker, "Dynamic prefetching of data tiles for interactive visualization," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 1363–1375.

[18] N. Bikakis, G. Papastefanatos, M. Skourla, and T. Sellis, "A hierarchical aggregation framework for efficient multilevel visual exploration and analysis," *Semantic Web*, vol. 8, no. 1, pp. 139–179, 2017.

[19] T. Buddhika, M. Malensek, S. L. Pallickara, and S. Pallickara, "Synopsis: A distributed sketch over voluminous spatiotemporal observational streams," *IEEE TKDE*, vol. 29, no. 11, pp. 2552–2566, Nov 2017.

[20] A. Wasay, X. Wei, N. Dayan, and S. Idreos, "Data canopy: Accelerating exploratory statistical analysis," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 557–572.

[21] R. Developers, "Redis," *https://redis.io*, 2020.

[22] mongoDB Developers, "Mongodb," *http://www.mongodb.org/*, 2020.

[23] The Apache Software Foundation. (2020) Apache Hadoop.

[24] Google, Inc., "Protocol buffers: Google's data interchange format," 2020.

[25] J. Kreps, N. Narkhede, J. Rao *et al.*, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, vol. 11, 2011, pp. 1–7.

[26] B. Welford, "Note on a method for calculating corrected sums of squares and products," *Technometrics*, vol. 4, no. 3, pp. 419–420, 1962.

[27] G. Niemeyer. (2008) Geohash. [Online]. Available: http://en.wikipedia.org/wiki/Geohash

[28] S. E. Yuksel, J. N. Wilson, and P. D. Gader, "Twenty years of mixture of experts," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 23, no. 8, pp. 1177–1193, 2012.

[29] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[30] T. Chen, T. He, M. Benesty, V. Khotilovich, and Y. Tang, "Xgboost: extreme gradient boosting," *Standalone Library 1.3.0*, pp. 1–4, 2015.

[31] J. VanderPlas *et al.*, "Altair: Interactive statistical visualizations for python," *Journal of open source software*, vol. 3, no. 32, p. 1057, 2018.

[32] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer, "Vega-lite: A grammar of interactive graphics," *IEEE transactions on visualization and computer graphics*, vol. 23, no. 1, pp. 341–350, 2016.

[33] G. Neglia *et al.*, "Access-time-aware cache algorithms," *ACM Trans. on Modeling and Perf. Eval. of Comp. Systems (TOMPECS)*, vol. 2, no. 4, pp. 1–29, 2017.

[34] H. J. Diamond *et al.*, "U.s. climate reference network after one decade of operations: Status and assessment," *Bulletin of the American Meteorological Society*, vol. 94, no. 4, pp. 485–498, Apr. 2013. [Online]. Available: https://doi.org/10.1175/bams-d-12-00170.1