Visualizing Large Sensor Network Data Sets in Space and Time with Vizzly

Matthias Keller, Jan Beutel, Olga Saukh, and Lothar Thiele
ETH Zurich, Computer Engineering and Networks Laboratory, Zurich, Switzerland
Email: {kellmatt, beutel, saukh, thiele}@tik.ee.ethz.ch

***Abstract*—This paper presents Vizzly, a middleware for the interactive browsing of large sensor network data sets. Provided map and line plot widgets allow to visualize structured data from mobile and static sensors. A user is completely free in selecting sensor data based on time and location, suitable levels of temporal and spatial detail are automatically chosen by the Vizzly server. Vizzly automatically adapts to user interactions, new data is automatically loaded when query parameters change. Request response times are significantly reduced by the use of caching techniques, most requests are served from already pre-computed data that is stored in the memory of the Vizzly server. Vizzly has already been successfully integrated into the PermaSense and OpenSense projects, a single instance is currently handling more than 550 millions of data points.**

***Index Terms*—Visualization, Caching, Big Data, Wireless Sensor Networks, Mobile Sensing**

I. Introduction

Wireless sensor networks (WSNs) have proven their applicability in many different scenarios, *e.g.*, data center monitoring [21], environmental monitoring [16] and building monitoring [8]. By combining several sensing modalities with on-board communication facilities, current generation smartphones are powerful networked sensing systems [7]. The vision of the Internet of Things (IoT) foresees sensor nodes being deployed in any area of our everyday life.

With networked embedded sensing systems reaching maturity, a new class of problems is given by the large amounts of data being generated by today's and future sensor networks. For example, a single sensing device that measures temperature every two minutes will already generate 262,800 data points per year. After four years, this single sensor will generate more than 1 million data points. Thinking of large collections of sensing devices with multiple sensing modalities, it becomes easy to understand that hundreds of millions or even billions of data points must be processed, stored and eventually analyzed during a few years of operation. To this day, more than 600 million data points have been sampled using more than 60 sensor nodes concurrently deployed across three long-term deployments in the PermaSense [16] project.

The visual inspection of sensed data is an important use case, *e.g.*, for system supervision, failure analysis, or during the development of new data processing algorithms. Ideally, all data can be viewed at different spatiotemporal scales computed for a number of different data resolution levels. Reducing the resolution is needed for limiting the amount of data being transferred to a user, and for addressing the limitations of the displaying device used, *e.g.*, the number of displayable pixels. However, the computation time needed for loading, sorting and aggregating large amounts of raw data points on-the-fly often exceeds the tolerable waiting time.

This paper presents Vizzly, a middleware that enables the interactive browsing of large sensor network data sets that originate from static and mobile scenarios. The back-end infrastructure of Vizzly consists of two main components, namely a cache layer and a web service. The web service component of Vizzly provides a single point of entry for retrieving sensor data that may originate from multiple data repositories. Based on received request parameters, Vizzly automatically chooses the temporal and spatial levels of detail in which data is sent to a user. Significant reduction of the data access times is achieved by the cache layer. Its task is (i) to continuously monitor and read from multiple repositories of different kind, *e.g.*, SQL databases, feeds, or CSV files, (ii) to aggregate received data, and (iii) to store pre-computed results in data structures that are optimized for time-based access.

The Vizzly front-end library enables the integration of interactive map and line plot widgets into existing web pages. A user must only specify the position and size of a new widget, the setup of required displaying components and visual control elements is automatically handled by provided library functions. Additionally, the Vizzly client library handles all client-server communication, *e.g.*, decides when new data must be dynamically loaded.

The contribution of this paper is as follows:

- We present Vizzly, a middleware that enables the interactive browsing of large sensor network data sets. The presented client-server architecture focuses on the problem of adapting to each user by automatically choosing the temporal and spatial resolution of returned data. For example, highly aggregated data is loaded when a user wants to quickly navigate through multiple years of data. In contrast, raw data points are shown when the selection is narrowed down to a particular point in time.
- The effectiveness of our approach is evaluated in the context of two distinct static and mobile sensing scenarios.
- We evaluate the performance of Vizzly based on the instance used in a production environment. The analyzed instance currently handles more than 550 millions of data points originating from more than 2,500 different sensing channels.

The structure of this paper is as follows: Related work

and the positioning of Vizzly are discussed in Section II. Section III presents the challenges found in the visualization of large sensor network data sets. The system design of Vizzly is described in Section IV, implementation details of Vizzly can be found in Section V. Section VI presents two case studies to prove applicability and to highlight the conceptual advantages of our approach. The performance of Vizzly is evaluated in Section VII. The broader applicability of Vizzly is discussed in Section VIII. Section IX concludes this paper.

II. Related Work

A. Sensor Data Visualization

A web interface for displaying sensed data is part of many sensing projects. For example, GlacsWeb[1] [22] and LoCal[2] [10] provide an interface for visualizing sensed data on a timeline. Based on Microsoft SensorMap, the interface of Life Under Your Feet[3] [24] also allows to display the locations of static sensors on a 2D map. A similar solution is provided by Climaps[4], a data visualization interface that is part of the SensorScope [3] project. PowerTron [18] has been developed for the visualization of power meter data in the PowerNet [17] project. Here, the locations of available power meters are shown on the floor plans of a building. Pachube/Cosm is a very popular online platform for sensor data streaming. An exemplary project based on Pachube/Cosm is the Japan Geigermap[5]. Apart from displaying the raw measurements from static and mobile sensors, *da_sense*[6] [23] also includes a heat map representation of noise pollution measurements. Within a more general scope, Google Fusion Tables[7] is a web service for visualizing data as maps, timelines and charts.

Vizzly is a middleware for visualizing large sensor network data sets in space and time. Both static and mobile sensing scenarios are supported, measurements of arbitrary length can be visualized in any temporal and spatial level of detail. Independent of a particular front-end component used for eventually displaying loaded data, Vizzly focuses on the problem of making sensor data dynamically and efficiently loadable when request parameters change. For example, less aggregated data is automatically loaded and displayed when the length of the time period of interest is decreased.

From a survey of existing approaches based on publicly available information, *e.g.*, manual analysis of the client-server communication of public data interfaces and consulting of available documentation, we find that Vizzly distinguishes itself from other solutions by dynamically loading data of varying detail. While similar techniques can be found in map visualizations, *e.g.*, in the *da_sense* project, we could not find other time series visualizations that would allow to display measurements of arbitrary length in any temporal level of detail. We find existing solutions to either limit the presentable time range, or to only offer data on a fixed, reduced level of detail, *i.e.*, data is only loaded once after the initial request, but not refined when the user selects a smaller area of interest.

B. Processing and Storage of Time Series Data

The creation of materialized views [14] is a common technique for continuously pre-computing and storing aggregates in database systems. Data cubes [13] allow for the efficient computation of multi-dimensional aggregates. For example, the Life Under Your Feet Project uses data cubes for aggregating data over time intervals and other quantities. RasDaMan [4] is a database system that is optimized for the storage of multidimensional raster data.

Monitoring tools for IP networks, *e.g.*, Zabbix[8] and ntop, often also include the ability to visualize measured performance data. In this context, *tsdb* [11] implements a compressed database for time series. Using a special storage scheme, massive data volumes can be efficiently handled and made available to a user, *e.g.*, for plotting. Probably the closest to this work is the function of the Archiver Daemon (ARD) that is part of *sMAP*[9] [9]. The storage layer used by the ARD is highly optimized for the processing and storage of time series data.

sMAP, *tsdb* and Vizzly share the design decision of implementing data processing in an application layer and use the underlying database system as a key-value store only. Instead of heavily relying on the support of certain features, *e.g.*, data cube analysis, flexibility is gained when basically any database system can be used. While *sMAP* and *tsdb* can potentially also be extended to support this, Vizzly is in particular designed for data originating from both static and mobile sensors, *i.e.*, sensor readings that are annotated with both time and location information.

III. Visualizing Large Sensor Network Data Sets

Being able to visually inspect recorded data is advantageous for all stakeholders of a sensing system, *e.g.*, design engineers, system operators, scientists, and even the general public.

For small data sets that consist of some hundreds of data points, a simple approach is to just send raw data points to a client for plotting. However, this approach does not scale for large multi-year data sets that consist of millions of data points. The amount of data to be transferred to a user becomes too large, the client would need to perform extensive computation, *e.g.*, sorting and down-sampling, for making the data displayable. Instead, data must already be filtered, sorted and aggregated before it is transferred to a client that only displays the already prepared data.

Needed processing steps for making raw data displayable can be very expensive, *e.g.*, involve large database tables to be fully read and sorted. The response time of such a request can easily reach tens of seconds if the underlying system has not been specifically optimized. This is problematic as the

[1] http://env.ecs.soton.ac.uk/glacsweb/iceland/graph/
[2] http://new.openbms.org/
[3] http://dracula.cs.jhu.edu/luyf/en/tools/VZTool/Default.aspx
[4] http://climaps.com/
[5] http://japan.failedrobot.com/
[6] http://www.da-sense.de/
[7] http://www.google.com/fusiontables
[8] http://www.zabbix.com/
[9] http://code.google.com/p/smap-data/

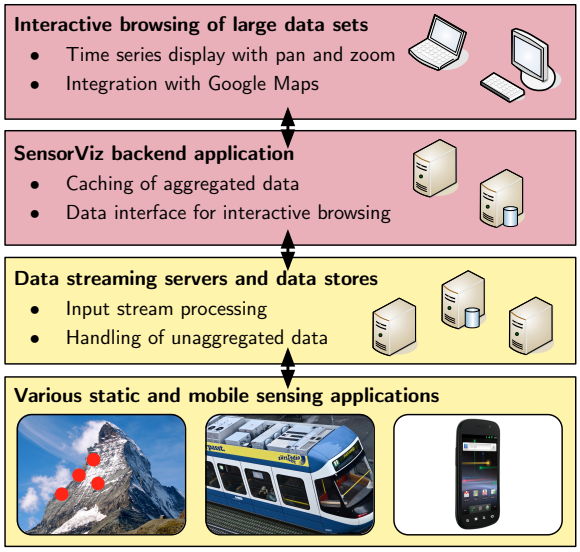


Figure 1. Exemplary usage scenario for Vizzly. Without the need of modifying existing infrastructure (yellow boxes), powerful plotting capabilities are added by Vizzly (red boxes).

adoption and success of a visualization tool are threatened when interested users perceive that the front-end responds too slow [6].

In the context of visualizing large sensor network data sets, this paper wants to address the following questions: Which data structures allow for the efficient retrieval of spatiotemporal, structured data at different scales? Which system design is suited for as many application scenarios as possible while also satisfying the needs of all user groups? What can we learn from standard PC memory architectures when designing a cache application that can choose to either store pre-aggregated data in different back-ends, *i.e.*, in memory (RAM) or in a database, or to further aggregate already pre-aggregated data on-the-fly?

In the following, we present the design and implementation of Vizzly, a middleware for the interactive browsing of large sensor network data sets. The applicability of Vizzly is evaluated in the context of two diverse research projects, the performance of Vizzly is evaluated based on the traces that originate from a production environment.

IV. Vizzly System Design

An exemplary usage scenario for Vizzly is shown in Figure 1. Starting from the bottom, structured data is recorded in various static and mobile sensing applications. Recorded data is then uploaded to a streaming server that annotates, *e.g.*, adds meta-information, and stores the data received. Vizzly continuously monitors and reads from multiple streaming servers, cached aggregates are immediately updated when new data arrives. The Vizzly back-end can handle multiple clients in parallel, requests are either served from pre-computed aggregates, from an on-the-fly aggregation of already aggregated data, or by forwarding the results of raw data requests. The latter are always served by the lower layer, *i.e.*, the corresponding streaming server. On the client side, the Vizzly front-

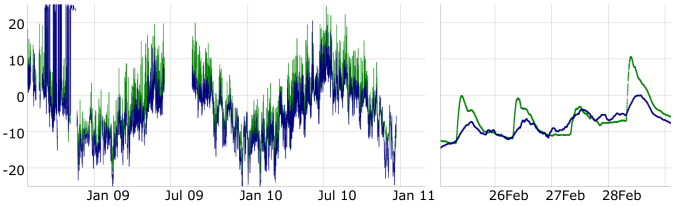
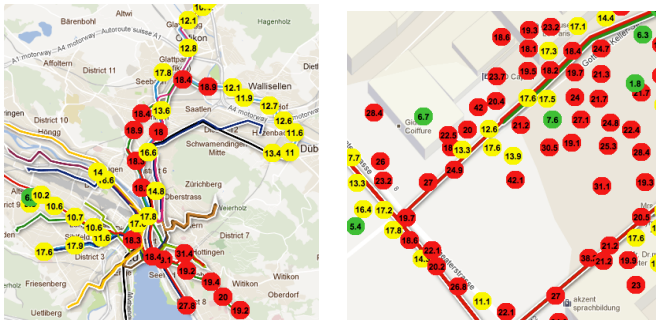


Figure 2. Temperature readings from two sensor nodes. The temporal resolution must be highly reduced for displaying multiple years in a single view. Less aggregated or even raw readings are shown when the time period of interest is lessened.



(a) 9.6 km x 9.6 km (b) 148 m x 148 m

Figure 3. Ozone data shown at two different spatial scales. Both excerpts correspond to the same measurement period, more detail is shown when the selected map area gets smaller. A user can change the data source of interest, *e.g.*, select CO measurements instead of ozone measurements, the time period of interest, and the map area of interest. After each interaction, missing data is automatically loaded from the Vizzly back-end. Shown data points are within the accuracy of the GPS receiver used.

end library implements map data and time series displays. Exemplary screenshots are shown in Figure 2 and Figure 3, respectively. The time series display can be used standalone, but is also part of the map data display. Only measurements taken within the corresponding map area are shown in a line plot when a user selects a map marker.

Decoupling data visualization from data storage certainly increases the overall system complexity, *e.g.*, requires data to be synchronized between two systems. However, adding missing functionality to an existing data store is often risky or even impossible. For instance, adding visualization capabilities may decrease the performance of another important use case. Organizational issues, *e.g.*, license restrictions, may not allow for adding modifications. Vizzly is an optimized and centralized solution that enables the visualization of sensor data. Vizzly is not restricted to a certain system, but can potentially be integrated into many existing platforms.

Detached from the concrete implementation of Vizzly (see Sec. V), we will now firstly present the overall system design of Vizzly. The interfacing between front-end and back-end is described in Section IV-A. Section IV-B presents the mechanism used for automatically determining the temporal and spatial resolutions in which data is sent to a user. The strategy used for aggregating spatiotemporal data in time and space is presented in Section IV-C. Section IV-D presents data structures used for caching and persisting aggregated data. Vizzly is not automatically informed of new sensor data, but

actively polls known repositories. The automated updating of cached contents is discussed in Section IV-E.

A. Client-server communication

Clients neither store sensor data nor any system state, *e.g.*, a list of available sensors. Instead, all information is dynamically loaded from the Vizzly back-end. Apart from necessary standard request parameters, *e.g.*, the time period of interest, choosing from large collections of sensors requires a precise but also flexible specification format. While mix-ups between similarly named sensors in different repositories need to be avoided, participatory sensing scenarios on the other hand require that sensor readings can be selected both dependent and independent of the recording device, *e.g.*, the particular smartphone used.

To address this problem, we introduce the notion of "virtual signals". In the context of Vizzly, a "virtual signal" can be described as a tuple (N, C, R) that consists of a sensor node N, a sensing channel C, and a data repository R. Selecting a particular sensor node can be omitted by setting N to a wildcard value. If the wildcard value is set, data is automatically combined on the equal sensing channel C and the equal data repository R[10]. Exemplary virtual signals are *(node 25, temperature, repository 1)* and *(node ANY, ADC channel 3, repository 2)*. Exactly one virtual signal must be specified in a *map data* request, arbitrary combinations of one or more virtual signals can be specified in a *time series* request.

B. Algorithmic Selection of the Returned Level of Detail

The size of the screen area for displaying map and line plots varies among different clients. In consequence, the optimal level of temporal and spatial detail in which data is to be sent to a client also varies. For example, the doubled number of data points can be displayed when a user views a map in full-screen mode instead of restricting the map to only one half of the screen. The lack of dynamically adapted levels of detail would lead to either unused space on large screens, or to overlapping data points on small screens. To overcome this, the Vizzly back-end first automatically decides if unaggregated data can be displayed or, if not, calculates suitable temporal and spatial target resolutions $\hat{r}_{\text{temp}}$ and $\hat{r}_{\text{spat}}$. Target resolutions $\hat{r}_{\text{temp}}$ and $\hat{r}_{\text{spat}}$, respectively, then define the window length of the aggregation operator used.

For this mechanism to work, clients are requested to specify the dimensions of the displaying widget used. Data included in the response of a *map data* request is always reduced to a dynamically computed $\hat{r}_{\text{spat}}$. Decided separately for each of multiple included time series, the response of a *time series* request can contain both unaggregated data and values that have been aggregated to a dynamically chosen $\hat{r}_{\text{temp}}$.

Based on a defined size of a grid cell in pixels, determining $\hat{r}_{\text{spat}}$ starts with calculating the number of displayable grid rows and columns. The geographical dimensions of a quadratic grid cell are then derived from the map area of interest that is specified in geographic coordinates. The optimal spatial target resolution $\hat{r}_{\text{spat}}$ corresponds to the geographic length of the edges of a grid cell, *e.g.*, $\hat{r}_{\text{spat}} := 1\,\text{km}$.

Determining the temporal level of detail starts with calculating the maximum number of data points per time series that the client can display. This number is obtained by dividing the reported width of the plotting canvas by a defined maximum average number of data points per pixel. With the help of continuously updated estimates of the sampling rate of each virtual signal (see Sec. IV-E), Vizzly first decides for each virtual signal if the client can be supplied with unaggregated data. To decide this, the estimated number of unaggregated data points is compared with the maximum displayable number of data points. If the level of temporal detail must be reduced, the optimal temporal target resolution $\hat{r}_{\text{temp}}$, *e.g.*, $\hat{r}_{\text{temp}} := 1\,\text{hour}$, is determined by dividing the time period of interest by the maximum number of data points per time series.

For being able to further aggregate already aggregated data, target resolutions $\hat{r}_{\text{spat}}$ and $\hat{r}_{\text{temp}}$ can not be arbitrarily chosen, but need to be multiples of defined highest spatial and temporal target resolutions. Intermediate resolution levels $\hat{r}_{\text{spat}}$ and $\hat{r}_{\text{temp}}$ are "rounded" to the next available smaller target resolution.

C. Aggregation of Spatiotemporal data

The level of spatial and temporal detail may need to be reduced before data is sent to a client. In this case, the data of higher or equal resolution is loaded from the cache and, if needed, further aggregated on-the-fly. While users can choose arbitrary combinations of temporal and spatial levels of detail, the number of possible combinations is clearly too large for each combination being stored in a cache.

To address this problem, we propose a location-preserving aggregation scheme: Temporal aggregation is carried out separately for each location, the results of the location-preserving aggregation are then cached. While this strategy certainly decreases the efficiency of data reduction, *i.e.*, leads to a smaller reduction in terms of the number of returned samples, one stored aggregate per virtual signal is sufficient for being able to choose any level of detail for any later spatial data aggregation. Additionally, this approach preserves advantages of aggregating and organizing data in the temporal domain. Compared to executing the spatial aggregation step first, this order of aggregating spatiotemporal data in time and space benefits from the structure of the data. Preserving temporal locality is much less complex than organizing data in the two-dimensional spatial domain.

Data series without location information can be described as a set of tuples (t, v). Each tuple describes a data point that consists of a timestamp t and a sensor reading v. For aggregating data in the temporal domain, the first step is to reduce the resolution of the timestamp t to the target resolution $\hat{r}_{\text{temp}}$, *e.g.*, $\hat{r}_{\text{temp}} := 10$ minutes.

$$R(t, \hat{r}_{\text{temp}}) := \left\lfloor \frac{t}{\hat{r}_{\text{temp}}} \right\rfloor \cdot \hat{r}_{\text{temp}}$$

[10]Please note that Vizzly expects underlying time series to be normalized with respect to possibly different sensor types or sensor calibrations used.

In case the sensor data is not signed with a location information, data points are grouped by their truncated timestamp t'. Data points with an equal truncated timestamp are put into the same set $\mathcal{G}$, the aggregated sensor value v' of the new tuple (t', v') is obtained by applying the aggregation function $A(\mathcal{G})$. Exemplary aggregation functions are the calculation of the mean, the sum, the number of elements, or the smallest value.

$$v' := A(\mathcal{G}) \text{ for } \mathcal{G} := \{v \mid R(t, \hat{r}_{\text{temp}}) \equiv t'\}$$

Already aggregated data, *e.g.*, a set of (t', v') tuples, can be further aggregated, *e.g.*, to obtain (t'', v''). Given that data was reduced to a target resolution $\hat{r}_{\text{temp}}$ in the previous step, the new target resolution $\hat{r}'_{\text{temp}}$ has to be a multiple of $\hat{r}_{\text{temp}}$, *i.e.*, $\hat{r}'_{\text{temp}} := \hat{r}_{\text{temp}} \cdot c$ with $c \in \mathbb{N}, c > 0$.

For measurements from location-aware sensors, the extended tuple $(t, v, l_{\text{lat}}, l_{\text{lng}})$ also includes the geographical latitude l_{lat} and longitude l_{lng} of the location of measurement. [11]

To preserve a later aggregation by the location of measurement, location information must remain untouched during temporal aggregation. Data points from location-aware sensors must therefore be grouped by their truncated timestamp t' and by their location of measurement that is specified by l_{lat} and l_{lng}. Data points with an equal truncated timestamp and equal location information are put into the same group, the aggregated sensor value v' of the new tuple $(t', v', \bar{l}_{\text{lat}}, \bar{l}_{\text{lng}})$ is similarly obtained by applying the aggregation function $A(\mathcal{G})$ to the set of data points $\mathcal{G}$.

$$v' := A(\mathcal{G}) \text{ for } \mathcal{G} := \{v \mid R(t, \hat{r}_{\text{temp}}) \equiv t' \wedge l_{\text{lat}} \equiv \bar{l}_{\text{lat}} \wedge l_{\text{lng}} \equiv \bar{l}_{\text{lng}}\} \quad (1)$$

The location of measurement for which v' is computed is defined by $\bar{l}_{\text{lat}}$ and $\bar{l}_{\text{lng}}$. In practice, $\bar{l}_{\text{lat}}$ and $\bar{l}_{\text{lng}}$ are automatically set while iterating over the list of locations.

Aggregating data in the spatial domain starts with reducing the resolution of the location of measurement. Both l_{lat} and l_{lng} are reduced to the same target resolution $\hat{r}_{\text{spat}}$.

$$R(l_{\text{lat}}, \hat{r}_{\text{spat}}) := \left\lfloor \frac{l_{\text{lat}}}{\hat{r}_{\text{spat}}} \right\rfloor \cdot \hat{r}_{\text{spat}} \qquad R(l_{\text{lng}}, \hat{r}_{\text{spat}}) := \left\lfloor \frac{l_{\text{lng}}}{\hat{r}_{\text{spat}}} \right\rfloor \cdot \hat{r}_{\text{spat}}$$

Spatiotemporal data that has been recorded within the time period of interest $[t_s, t_e]$ is grouped by its truncated location information. Based on the length of the time period of interest, Vizzly automatically chooses the level of temporal detail that is used as the input for the spatial aggregation step. Without further specifying the concrete level of detail used, the data is taken from the set of tuples $(t^{(n)}, v^{(n)})$.

$$v' := A(\mathcal{G}) \text{ for } \mathcal{G} := \{v^{(n)} \mid t_s \leq t^{(n)} \leq t_e \wedge R(l_{\text{lat}}, \hat{r}_{\text{spat}}) \equiv l'_{\text{lat}} \wedge R(l_{\text{lng}}, \hat{r}_{\text{spat}}) \equiv l'_{\text{lng}}\}$$

[11] As the altitude of the location of measurement is not required for 2D map plots, this information is currently omitted.

D. Efficient Storage of Pre-Computed Data

When processing a user request, Vizzly first loads all data that lies within the time period of interest. If location information is relevant, found records are then filtered to only include data from the requested map area of interest. To support this, the caching layer of Vizzly must fulfill the following three requirements. First, data structures used for storing aggregated data must be optimized for time-based access. Second, although data is aggregated before it gets cached, storing aggregated data can still require considerable space for large data sets. Third, the operation of Vizzly requires certain meta-data to be stored, *e.g.*, how often cache contents were loaded, or when the last update took place.

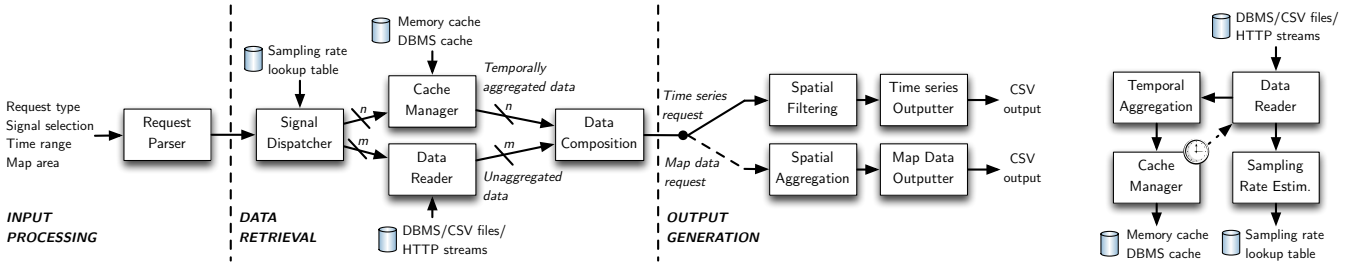
We propose two diverse back-ends for storing aggregated data, *i.e.*, storing aggregated data in memory (RAM) and in a SQL database. While aggregated data that is stored in memory can be accessed significantly faster, the SQL database adds cheaper storage capacity and persistence. Infrequently used data can be offloaded to the slower tier, populating cold memory from the SQL database is faster than again fetching unaggregated data from its source. Since only aggregated data is cached in Vizzly, any time information used in the following description must be seen in the context of a certain temporal target resolution $\hat{r}_{\text{temp}}$.

The memory back-end uses multi-dimensional arrays for storing sensor readings and location information. Timestamps are not explicitly stored, but are implicitly given by the index of an array element. The data structure used for storing time series data without location is depicted in Figure 4(a). Each instance of this data structure can be described by a tuple $((N, C, R), \hat{r}_{\text{temp}}, T_{\text{start}})$ that specifies the corresponding virtual signal, the temporal resolution $\hat{r}_{\text{temp}}$ of the stored data, and the timestamp T_{start} that corresponds to the first array element.

The index of the first element is always 0, the timestamp of any array element is obtained by multiplying its index by $\hat{r}_{\text{temp}}$ and adding the result to T_{start}, *i.e.*, $T := T_{\text{start}} + i \cdot \hat{r}_{\text{temp}}$. Not only reducing the amount of stored data, this linear relationship between array indexes and timestamps allows to access data very efficiently. Each array element covers a time period of length $\hat{r}_{\text{temp}}$. Adding new data only requires to either update the last array element or to append a new element to the end of the array.

An additional lookup table, see Figure 4(b), is needed for storing time series data with location information. As location information is preserved during temporal data aggregation, there can be multiple aggregates referring to the same timestamp. The relationship between array indexes and timestamps is again linear in the lookup table. Each lookup table record specifies the start and end of contiguous segment in a second one-dimensional array that stores sensor readings with location information. Both arrays must be updated when new data is added. The scope is again limited to replacing at most the last array element or the last contiguous segment, respectively.

The database back-end organizes aggregated data in several database tables. The contents of each database table can be



(a) Processing of a data request

(b) Background operation

Figure 5. The returned temporal level of detail is decided separately for each requested virtual signal. After all data has been collected and put together, generating the final response may require further aggregation or filtering in the spatial domain. Sampling rate estimations and cached contents used for generating CSV outputs are continuously updated by a number of concurrently running background threads.

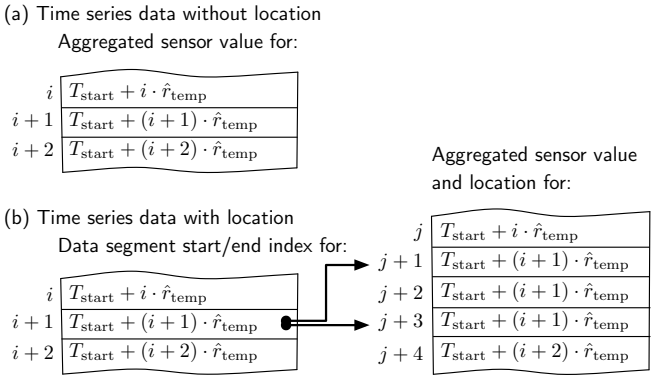


Figure 4. Data structures used for storing aggregated data in memory. T_{start} is the timestamp of the first array element. Location-preserving temporal aggregation can yield multiple aggregates for the same time but distinct locations.

described by a reduced tuple $((N, C, R), \hat{r}_{\text{temp}})$ that specifies the source and the temporal resolution $\hat{r}_{\text{temp}}$ of the table contents. Aggregated data without location information is stored in database tables consisting of two columns, *i.e.*, timestamp and aggregated sensor value. Four columns are necessary for storing aggregated data with location. An index on the time column is used for faster accessing data based on time information.

E. Continuous Maintenance of Cached Contents

After Vizzly learned about the existence of a certain virtual signal, *e.g.*, from a user requesting certain data, it starts fetching the complete history of this virtual signal from the respective repository. Received data is then aggregated, the results of this aggregation step are stored within Vizzly.

Furthermore, unaggregated data is also used for estimating the sampling rate of sensors used. Vizzly needs this information for deciding if a user request can be served with unaggregated data, or if the response would contain too many entries (see Sec. IV-B). A simple solution for estimating the sampling rate is to divide the number of raw samples by the measurement duration. For achieving more robustness against configuration changes that might occur over time, we currently use a window-based approach that maintains monthly sampling rate estimates.

Both aggregated data and sampling rate estimates get outdated when new data is being added to the source repository. Vizzly continuously polls respective repositories and, if necessary, updates cached contents.

V. Vizzly Implementation

The Vizzly software package consists of a Java web application and a JavaScript library. After the back-end has been setup once, adding interactive map and line plots to existing web pages is very easy. Setting up a new plot only requires to specify the virtual signals of interest and an empty placeholder object on the web page itself. Size and position of the placeholder object can be freely chosen in the HTML markup of the web page, a fully functional visualization widget is then automatically created by the Vizzly front-end library. The Vizzly front-end library itself integrates *dygraphs*, a line plot library, the *Google Maps JavaScript API*, and the *jQuery UI* user interface library. Event-based communication with the back-end is established using *XMLHttpRequest*, loading new data is triggered by several user interactions, *e.g.*, a change of the time period of interest. Request details are specified in the JSON format, requested data is returned in the CSV (comma-separated values) format.

All back-end functionality is provided by a Java web application that runs within *Jetty*, a light-weight HTTP server. Unaggregated data is fetched from several instances of Global Sensor Networks [1] (GSN). GSN is a middleware for sensor networks that allows to access its data streams over HTTP. Multiple background threads are concurrently updating cached contents, aggregated data is stored in memory (RAM) and in a MySQL database. *DBCP* is used for pooling database connections, *i.e.*, subsequent database accesses are accelerated by connection re-usage. By applying a *gzip* compression filter before sending a response to a client, the amount of transferred data is significantly reduced up to a factor of five and more.

Apart from offering a data access interface, the Vizzly back-end also provides a web-based management console and a web-based performance probe. The management console allows to see which objects are currently stored in the cache, a user can request single cache contents to be removed. The web-based performance probe exposes certain performance indicators, *e.g.*, the current cache size. To support system

supervision, this information is periodically sampled by a network monitoring system.

Integral components of the Vizzly back-end and their interplay are shown in Figure 5(a) and Figure 5(b). The organization of cached contents is encapsulated by the *Cache Manager* component that only exposes an interface for retrieving aggregated sensor data. The *Cache Manager* maintains an internal list of known virtual signals, new virtual signals are learnt when a user requests a yet unknown virtual signal. The list of virtual signals is continuously traversed, the *Cache Manager* can decide to update related contents, to remove contents from the cache, or to move contents between different available cache back-ends.

VI. Two Diverse Use Cases

The requirements for the design of Vizzly originate from the PermaSense [16]/X-Sense [5] and OpenSense [2] research projects. For being able to analyze both long-term and short-term dynamics of monitored processes, the PermaSense project requires data to be visualizable on arbitrary time scales. Focusing on mobile sensors, *e.g.*, sensing systems mounted on vehicles, the OpenSense project requires data to also be visualizable at varying spatial scales. While more than 600 million data points have already been collected in the PermaSense project, we also expect the OpenSense project to collect 100 million data points in the next few years.

A. PermaSense: Dynamics of Varying Temporal Horizons

The PermaSense project strives for the observation of geophysical phenomena in high-altitude regions. Since the initial deployment in 2008, we currently operate four long-term sensor network deployments [19]. Approximately 60 deployed low-power sensor nodes fulfill different monitoring tasks, *e.g.*, the monitoring of ground temperatures and the monitoring of ice clefts, data is typically sampled every two minutes. New generation sensing devices also monitor deformation processes within rock walls, the movement of rock glaciers is monitored using around 30 online and offline GPS devices [25].

Apart from scientific data, operating complex, remotely located systems also requires large amounts of additional system information to be recorded. Health data is continuously sampled at all layers of the system architecture that ranges from low-power sensor nodes up to powerful back-end servers. Used sampling intervals range from 30 to 120 seconds.

Within the first year of test operation, Vizzly has been established as an important tool for system supervision. Being able to visualize data at arbitrary time scales down to single events allows on the one hand to assess the current system state, but also to detect long-term trends, *e.g.*, slowly degrading components. Apart from this specific application, Vizzly has also proven its usefulness for many other domain experts. Exemplary applications are the visual inspection of data quality, *e.g.*, outliers or data gaps, the visual inspection of signal characteristics, *e.g.*, its value range, and the visual selection data segments that are the most suited for a particular analysis.

B. OpenSense: Mobile Sensors of Different Kind

The OpenSense project [2] investigates the challenge of monitoring urban air quality using (i) mobile sensing stations installed on top of public transport vehicles and (ii) personal sensors such as enhanced smartphones and pocket sensors [12] in the city of Zurich, Switzerland. The long-term goal is to raise community interest in air pollution data and to foster its involvement in monitoring air quality in urban areas.

To this date, we equipped five trams with sensing stations measuring ozone, carbon monoxide (CO) and fine particles (PM). Ozone and CO concentrations are measured every minute, the PM sensor generates one sample every 5 sec. Additionally, we also use the GasMobile system for measuring ozone concentrations with an Android smartphone [15].

Over the past six months, one sensing station collected around 2.2 millions of measurements[12]. We plan to install ten mobile stations in Zurich by the end of the year and foresee being able to collect up to 100 millions of data points during two years of system operation. See [20] for a more detailed description of the OpenSense dataset.

Interactive browsing though time- and location-sensitive historical data at any desired level of detail is crucial (i) for fine-grained analysis of raw data by domain experts and (ii) for building various data processing and data access services on top of raw data. In particular, non-expert users are often interested in summaries on the development of air quality in a particular region and whether any limit on the concentration of pollutants was exceeded.

While the success of community-driven sensing highly depends on the interest of individual persons and the possibility to make data easy accessible and understandable for those parties, Vizzly helps OpenSense to solve challenges that arise with the large volumes of gathered data.

VII. Performance Evaluation

The sensor data of both projects, PermaSense and OpenSense, is currently handled by a single instance of Vizzly[13]. Data of more than 2,500 different virtual signals is read from six data repositories, the input consists of more than 535 million unaggregated data points. Most data originates from the PermaSense project and thus does not include location information. More than 5.5 million data points that originate from the OpenSense project also include location information.

In the current configuration, unaggregated data points are first down-sampled to the temporal resolution of $\hat{r}_{\text{temp}} := 4\,\text{min}$. The aggregation function used is the calculation of the mean value. While the temporal resolution could also be separately chosen for each virtual signal, this static setting is currently derived from the two minute sampling period used in PermaSense. Lower levels of detail are retrieved by further down-sampling already aggregated data.

Aggregated data with the temporal resolution $\hat{r}_{\text{temp}}$ of $4\,\text{min}$ is stored in a MySQL database. Currently 306 million stored

[12]Note, that the stations are usually turned off over night.

[13]The public data interfaces of both projects can be accessed at http://data.permasense.ch and http://data.opensense.ethz.ch

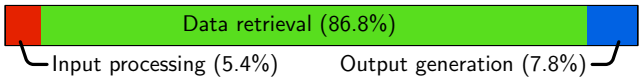


Figure 6. The request execution time is dominated by the data retrieval phase. From 14,900 measured requests, fulfilling the request without transferring the result to the client took 0.4 seconds or less in 90% of the cases. More than 97% of the requests could be fulfilled within at most 2.5 seconds.

aggregated data points account for around 11 GB of data on the MySQL server. Further down-sampled data is additionally stored in memory, around 230 million aggregated data points with a temporal resolution of 960 sec occupy circa 880 MB of RAM. Only two levels of temporal detail are stored, all other temporal resolutions are computed on-the-fly by down-sampling already aggregated data. While currently all virtual signals are treated equally, precious memory will be better utilized when contents are moved based on some metrics, *e.g.*, the number of requests for a particular virtual signal.

For understanding the performance of the Vizzly back-end, we are measuring the execution time of each request. Three separate measurements are made for each of the three phases that are required for serving a request, see Figure 5(a). Serving a single request can require the data of several virtual signals to be read in multiple cache accesses, the timing of each data access is measured separately. Apart from the execution times itself, other interesting metrics, *e.g.*, the number of returned data points, are also logged. While directly writing performance data to a database would significantly increase the response time of Vizzly, a background thread is in charge of asynchronously moving collected performance data from the memory to a MySQL database.

Generating the output usually takes less than a second. Additional time in the order of a few tenth of a second is added for compressing the result and eventually sending the result to the client. The distribution of the request execution time across the three phases needed for processing a request is shown in Figure 6. The request execution time is clearly dominated by the data retrieval phase. While initially processing the input data is very simple, generating the output may require the execution of an additional spatial aggregation or filtering.

Figure 7 shows the distribution of data fetch times. Aggregated data that is stored in memory (RAM) can be loaded very fast, the performance of loading aggregated data from the MySQL database is also acceptable. In contrast, loading unaggregated data from a GSN server can take several tens of seconds. Visible variations show no correlation with the size of the fetched result, but are caused by variations in the load of the system and by the changing state of other components, *e.g.*, the state of the database query cache.

As already aggregated data can be further processed on-the-fly without a noticeable delay, it is sufficient to cache only one aggregate of each virtual signal. While the current level of temporal detail can only be further reduced by down-sampling, the highest temporal level of detail in which aggregated data should be available must be cached. If the resulting data is too large for being completely stored in memory, it is reasonable to only keep frequently requested data in memory and to load

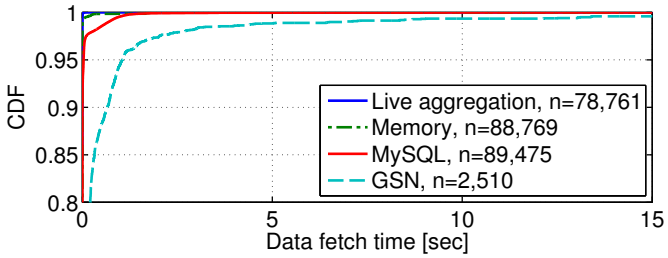


Figure 7. Analysis of more than 250,000 data accesses. Retrieving already aggregated and indexed data from memory takes less than 5 milliseconds in 99% of all cases. Further down-sampling data in the temporal domain takes less than 4 milliseconds in 99% of the cases. The value of 99th percentile for loading already aggregated and indexed data from the MySQL database is 690 milliseconds. Loading unaggregated data from GSN takes 6.9 seconds or less in 99% of the cases.

all other data from the less constraint MySQL database.

VIII. Integrating and Extending Vizzly

Ideally, Vizzly can be deployed out-of-the-box by just starting the server application and integrating provided front-end libraries into existing web pages.

However, individual project requirements, *e.g.*, the scheme used for storing data, might require extending Vizzly. For allowing us and other parties to easily extend Vizzly, the implementation of Vizzly follows modular design principles.

Regarding the Vizzly server, new data sources and cache back-ends can be implemented by either extending already existing components, *e.g.*, refining an existing MySQL data source, or by adding new components that follow defined abstract interfaces. On the front-end side, we find delivering CSV data as a common denominator that can be used together with many existing visualization libraries. While any client implementation must follow the defined request format of Vizzly, there are no further restrictions concerning platforms or libraries used for building new clients, *e.g.*, a native smartphone application.

Vizzly is open source software and free to use. The Vizzly project repository that is hosted on Google Code at `https://code.google.com/p/vizzly` includes all resources needed for installing and extending Vizzly.

IX. Conclusions

We presented Vizzly, a middleware for the interactive browsing of large sensor network data sets. Vizzly can be easily integrated into existing systems, only a web browser is required for accessing map and line plot widgets. Vizzly has been successfully applied to both mobile and static sensing scenarios, its ability to handle very large data sets that consist of hundreds of millions of data points has clearly been proven feasible. As next steps, we expect the implementation of suited cache replacement strategies to further improve response times while at the same time decreasing the memory consumption of Vizzly. Another topic is the implementation of other map representations, *e.g.*, heat maps.

Acknowledgements

We want to thank the anonymous reviewers for their valuable feedback that helped us to improve this paper. Roman Lim, Tonio Gsell and David Hasenfratz helped us with integrating Vizzly into the PermaSense and OpenSense data interfaces, their help is truly appreciated. The work presented was supported by NCCR-MICS, a center supported by the Swiss National Science Foundation, under grant number 5005-67322. The X-Sense and OpenSense projects are scientifically evaluated by the SNSF, financed by the Swiss Confederation, and funded by Nano-Tera.ch.

References

[1] K. Aberer, M. Hauswirth, and A. Salehi, "A middleware for fast and flexible sensor network deployment," in *Proc. 32nd Int'l Conf. Very Large Data Bases (VLDB '06)*, 2006, pp. 1199–1202.

[2] K. Aberer, S. Sathe, D. Chakraborty, A. Martinoli, G. Barrenetxea, B. Faltings, and L. Thiele, "OpenSense: open community driven sensing of environment," in *Proc. of the 1st Int'l Workshop on GeoStreaming (IWGS '10)*, 2010, pp. 39–42.

[3] G. Barrenetxea, F. Ingelrest, G. Schaefer, M. Vetterli, O. Couach, and M. Parlange, "SensorScope: Out-of-the-box environmental monitoring," in *Proc. 7th Int'l Conf. Information Processing Sensor Networks (IPSN '08)*, 2008, pp. 332–343.

[4] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann, "Spatio-temporal retrieval with RasDaMan," in *Proc. 25th Int'l Conf. on Very Large Data Bases (VLDB '99)*, 1999, pp. 746–749.

[5] J. Beutel, B. Buchli, F. Ferrari, M. Keller, L. Thiele, and M. Zimmerling, "X-Sense: Sensing in extreme environments," in *Proc. of Design, Automation and Test in Europe (DATE '11)*, 2011, pp. 1–6.

[6] N. Bhatti, A. Bouch, and A. Kuchinsky, "Integrating user-perceived quality into web server design," *Computer Networks*, vol. 33, no. 1, pp. 1–16, 2000.

[7] A. T. Campbell, S. B. Eisenman, N. D. Lane, E. Miluzzo, R. A. Peterson, H. Lu, X. Zheng, M. Musolesi, K. Fodor, and G.-S. Ahn, "The rise of people-centric sensing," *IEEE Internet Computing*, vol. 12, no. 4, pp. 12–21, 2008.

[8] M. Ceriotti, L. Mottola, G. P. Picco, A. L. Murphy, S. Guna, M. Corra, M. Pozzi, D. Zonta, and P. Zanon, "Monitoring heritage buildings with wireless sensor networks: The Torre Aquila deployment," in *Proc. 8th Int'l Conf. Information Processing Sensor Networks (IPSN '09)*, 2009, pp. 277–288.

[9] S. Dawson-Haggerty, X. Jiang, G. Tolle, J. Ortiz, and D. Culler, "sMAP: a simple measurement and actuation profile for physical information," in *Proc. 8th ACM Conf. Embedded Networked Sensor Systems (SenSys '10)*, 2010, pp. 197–210.

[10] S. Dawson-Haggerty, S. Lanzisera, J. Taneja, R. Brown, and D. Culler, "@scale: Insights from a large, long-lived appliance energy WSN," in *Proc. 11th Int'l Conf. Information Processing Sensor Networks (IPSN '12)*, 2012, pp. 37–47.

[11] L. Deri, S. Mainardi, and F. Fusco, "tsdb: A compressed database for time series," in *Proc. 4th Traffic Monitoring and Analysis Workshop (TMA '12)*, 2012, pp. 143–156.

[12] P. Dutta, P. M. Aoki, N. Kumar, A. Mainwaring, C. Myers, W. Willett, and A. Woodruff, "Common sense: participatory urban sensing using a network of handheld air quality monitors," in *Proc. 8th ACM Conf. Embedded Networked Sensor Systems (SenSys '10)*, 2009, pp. 301–318.

[13] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh, "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals," *Data Mining and Knowledge Discovery*, vol. 1, no. 1, pp. 29–53, 1997.

[14] A. Gupta and I. Mumick, "Maintenance of materialized views: Problems, techniques, and applications," *Data Engineering Bulletin*, vol. 18, no. 2, pp. 3–18, 1995.

[15] D. Hasenfratz, O. Saukh, S. Sturzenegger, and L. Thiele, "Participatory air pollution monitoring using smartphones," in *Proc. 1st Int'l Workshop on Mobile Sensing: From Smartphones and Wearables to Big Data*, 2012.

[16] A. Hasler, I. Talzi, J. Beutel, C. Tschudin, and S. Gruber, "Wireless sensor networks in permafrost research - concept, requirements, implementation and challenges," in *Proc. 9th Int'l Conf. on Permafrost (NICOP '08)*, 2008.

[17] M. Kazandjieva, B. Heller, P. Levis, and C. Kozyrakis, "Energy dumpster diving," in *Proc. 2nd Workshop on Power Aware Computing (HotPower'09)*, 2009, pp. 1–5.

[18] M. Kazandjieva, O. Gnawali, and P. Levis, "Visualizing sensor network data with Powertron," in *Proc. 8th ACM Conference on Embedded Networked Sensor Systems (SenSys '10)*, 2010, pp. 395–396.

[19] M. Keller, M. Woehrle, R. Lim, J. Beutel, and L. Thiele, "Comparative performance analysis of the PermaDozer protocol in diverse deployments," in *Proc. of the 6th Int'l Workshop on Practical Issues in Building Sensor Network Applications (SenseApp '11)*, 2011, pp. 969–977.

[20] J. J. Li, B. Faltings, O. Saukh, D. Hasenfratz, and J. Beutel, "Sensing the air we breathe – the OpenSense Zurich dataset," in *Proc. 26th Int'l Conf. on the Advancement of Artificial Intelligence (AAAI '12)*, 2012, pp. 323–325.

[21] C.-J. M. Liang, J. Liu, L. Luo, A. Terzis, and F. Zhao, "RACNet: a high-fidelity data center sensing network," in *Proc. 7th ACM Conf. Embedded Networked Sensor Systems (SenSys '09)*, 2009.

[22] K. Martinez, R. Ong, and J. Hart, "Glacsweb: a sensor network for hostile environments," in *Proc. 1st IEEE Communications Society Conf. Sensor, Mesh and Ad Hoc Communications and Networks (IEEE SECON '04)*, 2005, pp. 81–87.

[23] I. Schweizer, R. Bärtl, A. Schulz, F. Probst, and M. Mühläuser, "NoiseMap - real-time participatory noise maps," in *Proc. 2nd Int'l Workshop on Sensing Applications on Mobile Phones (PhoneSense '11)*, 2011, pp. 1–5.

[24] A. Terzis, R. Musaloiu-E, J. Cogan, K. Szlavecz, A. Szalay, J. Gray, S. Ozer, C. Liang, J. Gupchup, and R. Burns, "Wireless sensor networks for soil science," *Int'l Journal of Sensor Networks*, vol. 7, no. 1, pp. 53–70, 2010.

[25] V. Wirz, P. Limpach, J. Beutel, B. Buchli, and S. Gruber, "Temporal characteristics of different cryosphere-related slope movements in high mountains," in *Proc. 2nd World Landslide Forum*, 2011.