Respawn: A Distributed Multi-Resolution Time-Series Datastore

Maxim Buevich † Anne Wright ‡ Randy Sargent ‡ Anthony Rowe †
† Dept. of Electrical & Computer Engineering
‡ Robotics Institute
Carnegie Mellon University, U.S.A.
{mbuevich,arwright,rsargent,agr}@andrew.cmu.edu

***Abstract*—As sensor networks gain traction and begin to scale, we will be increasingly faced with challenges associated with managing large-scale time-series data. In this paper, we present a cloud-to-edge partitioned architecture called Respawn that is capable of serving large amounts of time-series data from a continuously updating datastore with access latencies low enough to support interactive real-time visualization. Respawn targets sensing systems where resource-constrained edge node devices may only have limited or intermittent network connections linking them to a cloud-backend. The cloud-backend provides aggregate storage and transparent dispatching of data queries to edge node devices. Data is downsampled as it enters the system creating a multi-resolution representation capable of low-latency range-base queries. Lower-resolution aggregate data is automatically migrated from edge nodes to the cloud-backend both for improved consistency and caching. In order to further mask latency from users, edge nodes automatically identify and migrate blocks of data that contain statistically interesting features. We show through simulation and micro-benchmarking that Respawn is able to run on ARM-based edge node devices connected to a cloud-backend with the ability to serve thousands of clients and terabytes of data with sub-second latencies.**

I. Introduction

Technological advances in low-power processors, communication and sensors are rapidly accelerating our ability to record information about the physical world. This data has the potential to revolutionize application domains including critical infrastructure monitoring, health care, transportation, defense systems, manufacturing, smart buildings and city-wide energy optimization. However, for this data to be actionable, we need scalable and efficient solutions designed to handle the increasingly enormous amount of data being generated.

Current large-scale sensing systems are typically composed of a tiered architecture where resource constrained sensor nodes transmit data to a server-backend. Limited networking capabilities (for example in wireless networks) often force designers to significantly filter data at the source. In many cases, this data filtering removes information that could be used to support new functionality beyond the original design scope. Given the current trend in disk and flash memory technologies, it is now possible to store large amounts of data locally on an embedded gateway device.

Unlike most of the approaches used in current cloud computing systems, embedded gateways are CPU limited, memory constrained and exist as field devices on networks with high latencies. Many distributed sensing systems also suffer network outages which without coordinated local storage lead to data loss. To support interactive visualization and querying of data, these systems need to intelligently organize, pre-process and pre-fetch data to provide timely access to data. Ideally, a sensor networking datastore should also have provisions to support streaming communication that might be required by control applications. The availability of fresh (recent) data often conflicts with the throughput gains achievable by batching data.

In this paper, we present Respawn, a distributed time-series datastore designed to manage hundreds of thousands of sensor feeds while providing range-based queries at sub-second latencies from resource-constrained devices. To achieve this, Respawn leverages two concepts: *cloud-to-edge partitioning* and *multi-resolution storage*. In Respawn's distributed architecture, communication is load balanced between a few server-class machines and many inexpensive, low-end embedded edge devices. This is achieved by partitioning the data between the cloud and the edge, storing the low-resolution aggregate data in *cloud nodes* and the high-resolution data at field-deployed *edge nodes*. A *dispatcher* front-end is responsible for directing queries between the cloud and the edge and, in effect, maintaining low request latencies. The dispatcher is designed to operate predominantly out of memory to support tens of thousands of concurrent connections. A bloom filter-based caching layer is used to determine the location of requested data and to avoid costly network accesses.

For multi-resolution storage, Respawn leverages the open-source Bodytrack Datastore (BTDS) [1], a light-weight multi-resolution datastore for time-series data. On edge node gateway devices, BTDS is responsible for down-sampling incoming data and storing hierarchically-organized copies of the data at differing resolutions. This pre-processing upon ingest of the data significantly improves range-based queries where a subset of data is requested from an arbitrary timespan. In the evaluation section, we show that BTDS can perform range-based queries running on an embedded platform at nearly the same speed as range-based requests for standard relational databases running on server-class hardware. We also extend BTDS to support lossless compression, where different levels of compression are applied to different levels in the hierarchy, allowing for more commonly accessed data to be served faster.

Since edge nodes typically have latencies of almost an order of magnitude greater than the cloud, selective data migration can be used to further improve latency. A Quality-of-Service (QoS) parameter at each gateway node is used to determine how much bandwidth is available for data and hence how aggressively tiles can be migrated. Top-level aggregate

tiles are migrated first since these are typically used as starting points for range-based queries. Low-level (higher resolution) tiles are migrated based on both client access patterns and based on data metrics like standard deviation. Standard deviation is one of many metrics that can be used to pinpoint tiles that would be of more interest to clients. We evaluate these different migration schemes through a small user study in which participants are required to search through data.

In order to evaluate the performance of Respawn, we benchmark ingest and query times on a single server machine comparing against MySQL, SQLite and OpenTSDB. This includes an evaluation of the impact of using LZ77 compression. We also benchmark BTDS on memory-contrained 450Mhz ARM9 embedded hardware which cannot support MySQL or OpenTSDB. Finally, we evaluate our latency masking techniques using user traces from a network consisting of an embedded gateway board, dispatcher, and cloud datastore. Using trace-driven simulation, we evaluate how the system scales across large networks with thousands of edge nodes.

In summary, this paper makes the following three main contributions. First, it provides the design of a highly-scalable time-series datastore that leverages local multi-resolution storage, along with compression, and is able to operate on embedded hardware. Second, we present a cloud-to-edge handoff mechanism with intelligent caching to reduce latencies for interactive queries and visualization. Finally, we evaluate the system in terms of single-server performance and horizontal scalability through the addition of edge nodes.

II. Related Work

It is becoming increasingly apparent that no single approach can solve all data storage problems [2]. For sensor networking applications, it is important to be able to store and process many streams of data simultaneously. There have been several efforts to build upon databases like MySQL to support stream processing functionality like pattern correlation queries [3]. Other similar work has tried to incorporate stream processing engines and historical storage [4]. These efforts illustrate many of the challenges involved with large-scale time series data, even in the context of highly-connected and resource abundant datacenters. In this section, we highlight a few of the key distinctions between traditional databases and those designed for sensing applications.

A. Time Series Datastores

There have been many examples of time series databases for infrastructure monitoring (for example in data centers) and financial analysis, such as DataGarage[5], RRDtool [6], tsdb [7], TSDS [8], OpenTSDB [9], Dremel [10], Vertica [11] and DataSeries [12]. Some of these databases already support multi-resolution queries [5], [6], [9], [11]. However, none of them natively support sub-second time stamping. RRDtool is one of the few options that can operate on an embedded target, but it is designed to run locally and does not provide any distributed management features. Many of these databases do not natively support access control [5], [6], [7], [9], [10], [12]. Even the ones that do have access control [11] lack per-stream permissions and do not operate in a distributed manner. Respawn in contrast embraces a cloud-to-edge distributed operation with per-sensor stream security through use of the a Publish-Subscribe access control model. The OceanStore [13] project has very similar goals and features to Respawn. OceanStore is a global persistent data store designed to be highly scalable. One of its primary goals is to provide consistent and highly-available service to an infrastructure running on top of untrusted devices. It also strives to reduce latency by capitalizing on data locality whenever possible. However, OceanStore is not primarily a time series database and hence does not provide multi-resolution access. Because the data could be significantly more general, it utilizes access-based caching schemes rather than proactive data migration. It is also not optimized to run on embedded targets.

B. Streaming and Aggregation

Another approach to managing continuous data is to perform on-the-fly aggregation using stream processing techniques [14] [15]. This approach works well for networks that have enough provisioning to transfer data in real-time, but suffer in that they potentially lose information that could be processed later. In Respawn, we leverage an XMPP communication layer to deliver real-time transducer information and then provide a componentized storage facility that operates in the background. Decoupling the design of the stream-processing functionality from the historical data allows the system to more flexibly batch entries to increase ingest throughput.

C. Sensor Network Databases

Many databases have been proposed that operate within sensor networks, such as Diffusion[16], Tag [17], TinyDB [18], Cougar [19] and DIMENSIONS [20]. Diffusion provides a routing and in-networking processing approach that propagates queries using flooded messages. Cougar uses a similar approach only it uses a static configuration of nodes that is selected using a proxy front-end. This proxy front-end is still responsible for dispatching requests out to the leaf-nodes of the sensor network for data collection. There is no notion of storage or caching at the proxy. TinyDB focuses on running the database within resource constrained sensor nodes on microcontroller hardware. Queries are processed directly on endpoint sensor devices. Respawn is more geared towards data management that runs on sensor gateway collector devices rather than on the leaf nodes themselves. In many cases, these gateway collectors could be connected to proprietary networks where it would be difficult to run custom software. For example, one could connect a Respawn edge node to a buildings BACnet data feed and it would act as a consolidator of potentially hundreds of sensor streams. Dimension further improves upon the idea of in-network storage by using wavelet transforms to deal with data aging and multi-resolution.

D. Middleware

Certain sensor networking middleware frameworks also provide storage capabilities. sMAP [21], for example, focuses on with data delivery, but also provides querying and retrieval of sensor data. sMAP is compatible with MySQL for storage, but can also use a custom binary time-series datastore called readingDB to help cope with the performance challenges. readingDBs performance is significantly improved over MySQL,

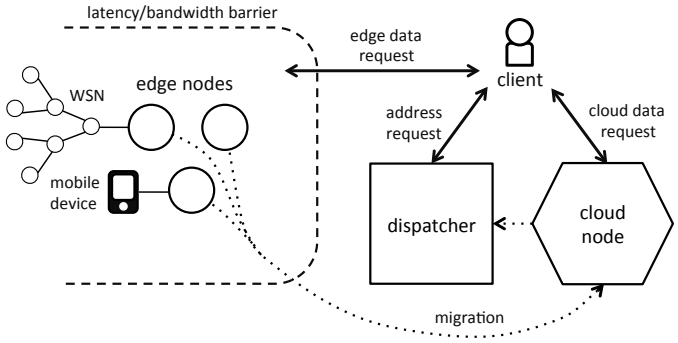


Fig. 1: Respawn High-Level Architecture

but it is still primarily a centralized datastore without built-in load balancing. readingDB does however provide delta encoding and zlib compression and can be manually distributed based on how sMAP is configured. In contrast, Respawns design is centered around the notion of edge-to-cloud partitioning and is highly distributed by default. Its primary goal is latency masking which it achieves through intelligent data migration.

E. Why Respawn?

Data centers have become increasingly effective at providing low-latency access to vast amounts of data. Services like Google Maps allows people to interactively scroll through databases that are hundreds of Terabytes in size with hundreds of Gigabytes of indexing information. Sensor data and the way in which it is collected differ significantly from the type of information that current data center practices are so efficient at handling. Typical operations performed on time series data include plotting, zooming, correlation, clustering, prediction, pattern matching queries and summarization. Time series processing systems often have both a historical and real-time component. In order to optimize ingest throughput, databases often batch large segments of data. Unfortunately, this introduces significant delay. Most traditional databases deal with read and write transactions that are for the most part independent. In time series databases, there is a constant stream of appends with intermittent queries. Specific to sensing systems, data sources are often distributed and must be collected on field devices. Unlike data centers with high-speed network interconnects, field devices are frequently connected with expensive links like cellular or satellite where customers are charged per kilobyte. Few open-source time series databases provide the timing granularity required for capturing physical events. The majority of these systems were designed for data center monitoring or financial applications that require integer second timing granularity. Sensing systems should be able to distinguish sub-second events. As sensor networks are finding their way into a wider range of large-scale complex applications, there is a growing need for a scalable storage platform customized for these particular needs.

III. Design and Implementation

The Respawn architecture enables interactive exploration and responsive querying of time-series sensor data, despite the data originating from embedded nodes behind a latency-bandwidth barrier. A high-level representation of the Respawn system architecture is displayed in Figure 1. Sensor readings are generated at the edge, in most cases from mobile devices and wireless sensor network deployments. The sensor data streams are transmitted through a series of adapters to Respawn edge nodes, low-cost embedded gateways outfitted with large amounts of flash storage. Edge nodes process and store the data locally and periodically notify a Respawn cloud node of new data received. Cloud nodes are server-class machines which handle much of the load of and serve as low-latency data caches for the Respawn system. The cloud nodes also facilitate a continuous migration of a subset of the total data being stored by the edge nodes. When data is migrated from the edge, the cloud node notifies the Respawn dispatcher.

The dispatcher serves as the system entry point for almost all data requests initiated by clients. It is designed to handle large volumes of small requests and to maintain thousands of open connections at once. The dispatcher's primary function is to redirect requests from clients to end-nodes. Redirections are decided based on client-perceived latency, as well as load on individual edge nodes and load on the entire network. In Section IV, we show that despite the centralized nature of the dispatcher, it scales to handle thousands of simulataneous users and does not represent a significant performance bottleneck in the architecture given our target network sizes.

A. Multi-Resolution Datastore

For the purpose of local data storage, Respawn leverages Bodytrack Datastore (BTDS), an open-source time-series datastore which targets multi-resolution queries and interactive visualization. BTDS stores raw data streams in a lossless format and additionally generates lossy aggregate versions of the streams for query acceleration. In the current implementation, aggregates are calculated by finding a mean average of datapoint values over a set window, however, it is possible to incorporate other aggregation functions. BTDS supports a command line interface for local client requests and an HTTP/JSON interface for web requests.

1) Data Model: A single instance of the datastore is a collection of data streams, with each stream separated into a single raw version and many aggregate version (Figure 2). An aggregate is a representation of the raw data down-sampled by a power of two, and therefore corresponds to a distinct level of resolution. A single aggregate is composed of many tiles, tiles in turn are composed of one-point buckets. Formally, a tile represents a contiguous time-range and a collection of *m* equally-sized buckets. A bucket is a "smaller" contiguous time-range containing either a single datapoint or no datapoint.

In a single instance of the datastore, all tiles contain exactly *m* buckets (regardless of resolution level) and thus contain a maximum of *m* points. Buckets at level *L* are half the size temporally of buckets at level *L+1*, thus tiles at level *L* represent a time-range half the size of tiles at *L+1* (please refer to Figure 2). This also means that, for any arbitrarily-chosen range in time, the level *L* view of the data contains twice the buckets and is up to twice the resolution of the level *L+1* view of the data. By convention, the bucket "width" (time duration represented by a bucket) at an arbitrary level is equal to 2^{level} seconds; thus at level 0 a bucket has a "width" of 1 second.

A tile is uniquely addressed by its resolution level and offset. A tile's offset at level *L* corresponds to its distance in time from the Unix epoch as measured in tiles of level *L*.

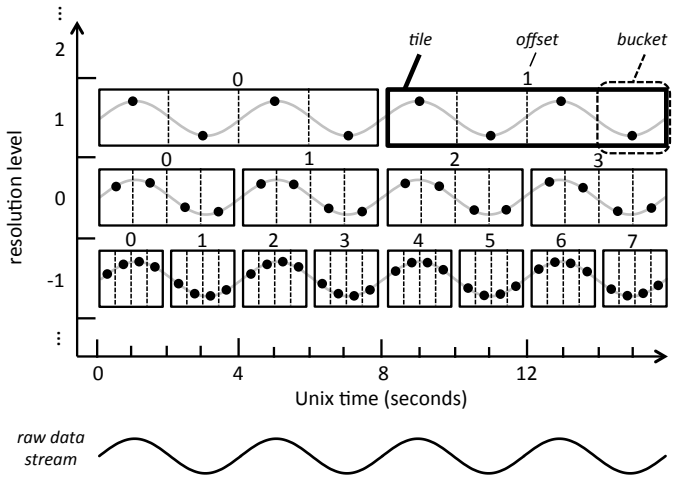


Fig. 2: BTDS Multi-Resolution Storage

Figure 2 shows an example of how aggregate data tiles are organized within BTDS. The raw data stream and three of the aggregate versions are displayed (other aggregates are not shown). Each aggregate is represented by tiles of *m=4* buckets (a small *m* value was chosen for presentation purposes). The stream in this example begins at the Unix epoch, thus the leftmost tiles have an offset of 0. At each level of resolution, tile offsets are incremented as time progresses.

BTDS stores data streams as uncompressed binary files on disk (In Section IV we explore the impact of adding compression to BTDS). Sets of contiguous tiles at a single resolution are organized into files in much the same way that buckets are organized into tiles. Files are named according to their corresponding range of offsets and are organized into two directory levels. Top-level directories separate individual data streams while bottom-level directories separate resolution levels within data streams. The structure of BTDS maps well to the organization of many common filesystems and enables performance benefits associated with filesystem-level caching and optimized lookup.

2) Making Requests: Respawn queries are implemented on top of independent BTDS tile requests. In Respawn, BTDS tile requests are made via HTTP and return JSON (Javascript Object Notation) representations of tiles. A tile request must specify a data stream, a tile resolution level, and a tile offset. Data stream names are split into two parts: a device name and a channel name. Channels which are grouped into devices can share timestamp values, resulting in accelerated ingest and saved disk space. The level and offset specified in a request maps directly to a tile position in the data stream. Because all tiles are precomputed at the time of ingest, requests are made in constant time, regardless of the resolution level. The format for an HTTP tile request is given below:

```
            (request format:)
GET /tiles/DEVICE.CHANNEL/LEVEL.OFFSET.json

                (example:)
GET /tiles/sensor.temperature/10.2609.json
```

Each tile request yields JSON data for a single tile. A response contains an array of datapoints, an array of fields describing each datapoint, and the tile level and offset. Typically, each time-series datapoint will contain a timestamp, a mean value, a standard deviation, and a count. Floating point values are stored with 64-bit precision. Count and standard deviation are meaningful for low-resolution aggregate tiles because they represent the number of raw datapoints used for calculating the mean value and the standard deviation of the datapoints. The format for a JSON tile response is given below:

```
            (response example:)
{
  "fields": [
    "time", "mean", "stddev", "count"
  ],
  "data": [
    [1367968303.677585, 486.696,  12.053, 1125],
    [1367969280.41341,  494.2760, 4.347,  1507],
    [1367970305.185479, 498.36,   3.0824, 1520],
    ...
    [1368391167.9347,   476.883,  5.690,  1531]
  ],
  "level": 10,
  "offset": 2609,
}
```

B. Distribution Layer Components

Time-series sensor data destined for storage enters through the edge nodes. It is first buffered in memory for a predefined amount of time so as not to incur the overhead of constant writes to the BTDS datastore. Periodically, the buffered datapoints are written to the datastore and committed to disk. The BTDS datastore parses the incoming points and, based on timestamp, collects them into tiles. Raw timestamp and value information is stored in the highest-resolution raw tiles; all other tiles are aggregate tiles generated from raw tiles. In addition, edge nodes run security key generators that publish keys to XMPP, which maintains network-wide access control.

A primary function of the cloud node is to serve data summaries collected from the edge nodes in the network. To improve latency, the cloud node maintains a collection of persistent and non-persistent data structures. The first of these is the Summary datastore, an instance of the BTDS datastore containing migrated edge tiles which are likely to be requested in the future. The migrated tiles represent an approximation of the total data in the distributed datastore. The storage-constrained nature of the edge nodes necessitates data aging, but this is not required of the cloud node. Additional modules are maintained to optimize accesses to the Summary datastore and to enforce access control policies; these are discussed further in the section on data migration.

The dispatcher acts as the first point of entry for almost all client-initiated data requests. It redirects requests for data tiles of particular channels to the appropriate end-nodes, either edge or cloud. The dispatcher does not serve time-series data; requests and response payloads are kept small (i.e. smaller than 100 bytes). This, coupled with the fact that most dispatcher data structures are kept in memory, enables the dispatcher to serve a steady-state stream of over ten thousand requests per second (as described in Section IV). The dispatcher's data structures are primarily copies of the structures generated by the cloud node. For most requests, the dispatcher decides where clients are redirected to and, as a result, how load in the system is distributed.

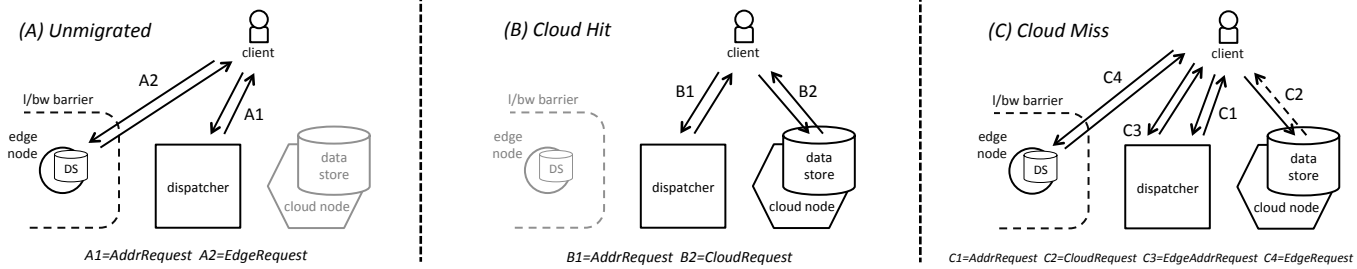


Fig. 3: Request Transaction Types

An XMPP back-end hosts a publish-subscribe network and is used by the three primary components for the sharing of event notifications and other lightweight metadata. XMPP also provides a delivery system for streaming data, as well as access control and security.

C. Request Redirection

In this section, we discuss the details of the Respawn dispatcher and the movement of a client's data fetch through the Respawn system as shown in Figure 3. All of the requests described in this section are standard BTDS-format HTTP requests extended with a security key.

A data retrieval is initiated by an *AddrRequest* arriving at the dispatcher. The dispatcher uses an access control list to map the requested channel to the security key currently in use by the channel's edge node. A key mismatch causes an error response to be sent. Upon a key match, a list of base resolutions and a summary bloom filter are checked. If either match, a redirection to the cloud node is returned. The Bloom filters are sized to keep the probability of false-positives low, implying a reasonable assurance that in the case of a match the requested tile exists at the cloud node. If the previous check resulted in false outputs, the requested tile has not been migrated and a redirection to the edge is returned, causing the client to make a subsequent *EdgeRequest*.

The probabilistic nature of the CTR Bloom filters allows for false-positive cloud redirections to occur. This results in three main types of request traces in the Respawn system: "Unmigrated", "Cloud Hits," and "Cloud Misses." The "Unmigrated" traces are the simplest of the three. When a request processed by the dispatcher fails to positively match, the tile has not been migrated and the client is redirected to edge. When a match on the bloom filters occurs, it is only likely that the tile has been migrated. Hence, a small percentage (i.e. 1%) of CTR Bloom filter matches result in "Cloud Miss" traces. In such cases, the client is redirected to the cloud node, causing it to send a *CloudRequest*. The request is checked against the full Cloud Tile Registry (CTR), which returns a negative result, signifying that the tile has not been migrated. The negative result is returned to the client, causing the client to issue an *EdgeAddrRequest*. An *EdgeAddrRequest* is functionally identical to an *AddrRequest*, with the exception that it is guaranteed to redirect the client to the tile's edge node. Once the client receives a response to the *EdgeAddrRequest*, it uses the response to generate an *EdgeRequest*, which it sends to the appropriate edge node. Upon receiving the request, the edge node verifies the client's access, retrieves the tile from the local datastore, and serves the client's request.

D. Data Migration

For distributed storage systems which leverage storage at the edge, the quality of network links to the edge devices can be a substantially limiting factor for performance, especially in terms of latency. In this section we will describe the mechanics of data migration used to reduce access latencies.

1) Predictive Caching: As an alternative to the classic caching scheme we have developed methods for *Predictive Caching* in order to minimize the effect of the latency-bandwidth barrier. Predictive caching leverages the structure of time-series sensor data by analyzing and isolating certain characteristics of the data. The data characteristics are used in the construction of a prediction model which decides which portions of data are likely to be requested and should be preemptively cached. We have defined and implemented two predictive caching schemes in Respawn, *Periodic Migration* and *Proactive migration*.

Periodic Migration: When browsing, clients will frequently request tiles with the lowest resolution first, in order to see a "big-picture" view of a dataset. From there they will "drill down" to explore particular sections of the data at a higher resolution. The effect of this phenomenon is that low-resolution tiles are on average requested more often than high-resolution tiles. Thus, it is often practical to bias tile migration toward lower-resolutions. The Respawn cloud node periodically migrates all tiles at or below a predefined resolution, which can be assigned on a per-edge-node or per-device basis. Periodic Migration is most effective for periodic data feeds from time-triggered sensing systems.

Proactive Migration: Whereas Periodic Migration leverages resolution information about data tiles for migration decisions, Proactive Migration involves extracting characteristics from the data itself for the same purpose. In Respawn, each tile generated at the edge is assigned a value corresponding to the standard deviation of the sensor values contained within the tile. For many datasets, standard deviation was found to be a good scalar approximation of the statistical importance of a time-series data-range. Furthermore, standard deviation over tiles is simple enough that it can be computed upon initial ingest of the data, even on a low-end embedded platform. We found Proactive Migration based on standard deviation to be especially effective at accelerating access times for sparse data feeds containing isolated events of interest. This suggests that Proactive Migration is especially suited for handling data feeds from event-triggered sensing systems.

2) Mechanics of Migration: Respawn maintains and migrates a collection of data structures to facilitate request redirection. The first of these is the Summary datastore, a subset

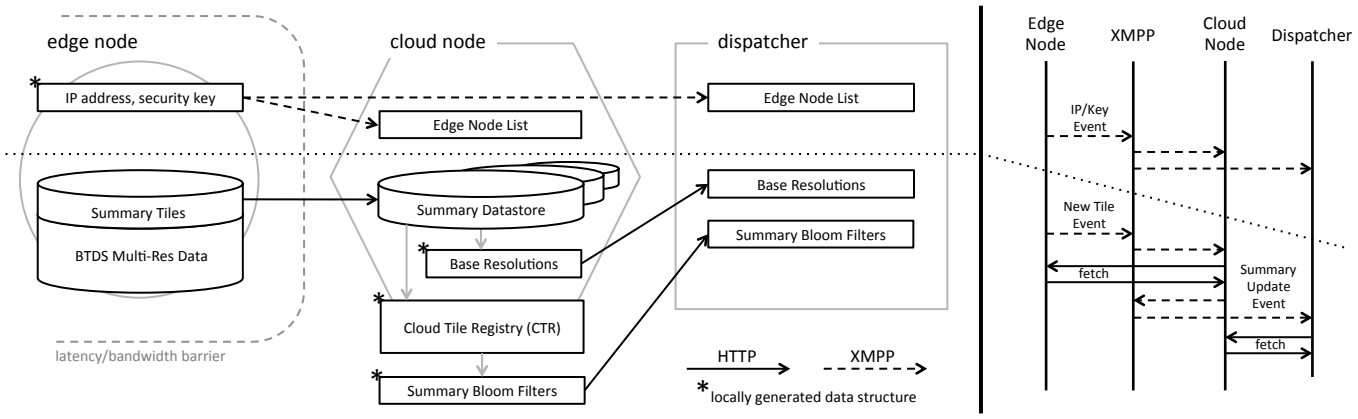


Fig. 4: Data Structure Migration within Respawn

of the tiles in the system chosen for migration (Figure 4). Two additional modules are maintained to optimize accesses to the Summary datastore: The Base Resolutions and the Cloud Tile Registry (CTR). The Base Resolutions are a mapping of time-series data channels to their base migration resolution, the resolution R for which the cloud node is guaranteed to migrate all tiles belonging to that channel whose resolution is R or lower. The CTR is a registry containing the addresses of all migrated tiles not represented by the Base Resolutions. Like the rest of the data structures maintained by the cloud Node, the CTR is segregated by channel and is thus implemented as a map from channel to list of unique tile addresses.

The cloud node makes migration decisions on a per-data-channel basis and thus a set of migrated tiles for one channel is independent from the set of another channel. Access to CTR addresses are further optimized by the addition of Summary Bloom filters. This enables faster, no-false-negative lookups on the existence of addresses. The last high-level structure maintained by the cloud node is the Edge Node List, which maps channels to their associated IPs, security keys, and other access control metadata. This listing is checked on every request in order to verify that a client requesting cached data originating at an edge node is authorized to access the edge node.

3) HTTP vs. XMPP: Respawn separates two common modes of communication, event notifications and bulk transfers, and implements them with two distinct layer-7 protocols. This choice has the benefit of increasing responsiveness to client requests, horizontal scaling potential, and reliability in overload cases. Event notifications, the first mode, are always sent as the direct or indirect result of an asynchronous sensor event entering the system through an edge node. These notifications are lightweight data transfers whose destination is a single node or a subset of all nodes in the system; global broadcasts are very rare. In many cases, event notifications cannot be predicted and thus it is difficult for a receiver to control data flow without severely hindering communication. Respawn uses the XMPP publish-subscribe protocol for these types of transactions. XMPP provides an event node abstraction for which each event node has one or more publishers and one or more subscribers, in addition to tunable quality-of-service parameters to place limits on data flow. Each event node is also associated with an access control list. Respawn uses these access control features as a basic building block in its security model. XMPP was explicitly designed for lightweight, asynchronous massaging and is thus a good fit for Respawn event messages.

Data fetches represent bulk transfers of structured data facilitated by client-server interactions. They can occur both periodically and as a result of an asynchronous event being received. Respawn uses HTTP for transferring time-series tiles and large metadata structures. HTTP serves these purposes well as a result of its simplicity, its bulk transfer efficiency, and its request-driven model. One common limitation of HTTP-based systems is the need to periodically ”poll” for data, which can result in wasted bandwidth in cases where data is not yet available or does not exist. This effect is mitigated in Respawn because most fetches are triggered by the receipt of a lightweight XMPP event. The use of HTTP’s request-driven model is also key in preventing overload for the most important components of Respawn. In terms of bandwidth, most back-end data transferred by the dispatcher and cloud node are initiated by the dispatcher and cloud node. Thus, it would be difficult to overload the dispatcher and cloud node through the addition of edge nodes.

4) Bloom Filter: A bloom filter is a probabilistic data structure used for quickly and space-efficiently testing an element’s membership in a set. Bloom filters are most commonly implemented as bit arrays in which each element is mapped to multiple bit positions via multiple unique hash functions. A set is queried for an element by hashing the element with the same set of hash functions and checking whether all of the bit positions are set to 1. Querying a bloom filter does not return an exact result; false positives are possible while false negatives are not.

Respawn maintains a bloom filter per data stream in the Summary Bloom Filters data structure. As tiles are added to a Bloom filter, the filter becomes less accurate and produces a larger percentage of false positives. A Bloom filter which produces a false positive ratio of 1% contains 9.57 bits of space per element in the set. When the number of elements in a Bloom filter becomes too great to maintain a false positive ratio of 1%, the cloud node will ”resize” the bloom filter by generating an empty one of double size, rehashing the values, and reinserting them.

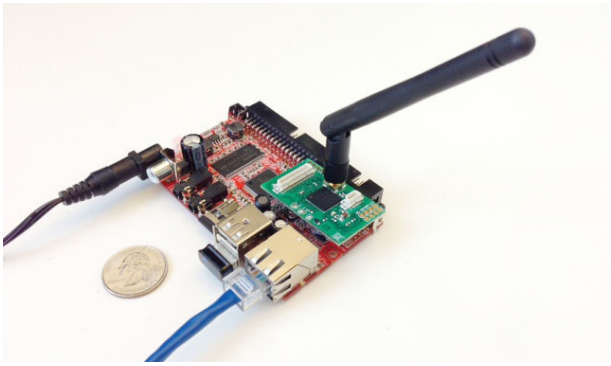


Fig. 5: ARM9 Embedded Gateway

IV. Evaluation

In this section we evaluate the primary components of the Respawn system. By micro-benchmarking BTDS against MySQL, SQLite, and OpenTSDB, we profile the ingest and query performance of a multi-resolution architecture against standard relational and time-series architectures. We also show initial compression ratio and compute the overhead for performing gzip compression on a large environmental sensor dataset. We then perform a design-based simulation of various dispatcher architectures and then benchmark the actual throughput of our implementation. Finally, we use traces collected from users browsing a typical time-series dataset to evaluate various Respawn migration schemes.

A. Multi-Resolution Storage

Each database was profiled on an 8-core E5320 Intel Xeon CPU clocked at 1.86GHz with 4GB of RAM. Since BTDS fits within a small enough footprint to operate on edge-node routers, we also benchmark it on a 450MHz ARM9 processor with 32MB of RAM operating from a 64GB flash micro-SD card (shown in Figure 5). Unless otherwise specified, each point on the following graphs represents the average of at least one thousand runs.

Table I shows the average request time required to fetch a 1024-point (1K) block of data from each database at a random location within a one million point dataset. We see that SQlite performs best and BTDS runs slightly faster than MySQL and significantly faster than OpenTSDB. Enabling compression adds a factor of six slowdown in BTDS performance. The embedded platform runs approximately 13 times slower on the embedded target. Given that each embedded edge node will receive a small fraction of the requests as compared to the main server, the overall response time of 310 ms is promising.

Database	1K-Point Random Query Time (ms)
BTDS	23.86
BTDS Compressed	159.51
BTDS Embedded	310.98
SQlite	11.98
MySQL	25.49
OpenTSDB	1206.38

TABLE I: Full resolution query performance

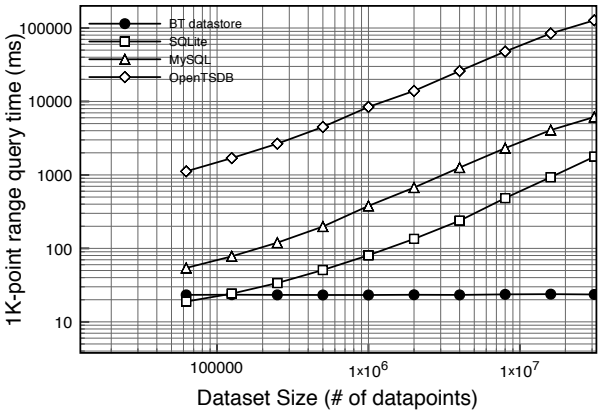


Fig. 6: Range-based query performance (log-log scale)

Next we evaluate the performance of range-based queries that are defined by a starting timestamp, an end timestamp and a resolution. On standard relational databases like MySQL, this requires fetching raw values and then computing aggregates. BTDS performs its processing on ingest. Figure 6 shows the difference in performance of each database with respect to 1K-point range-based query. The x-axis shows the performance as the size of the database increases. For standard databases, as the size of the dataset increases this incurs a larger penalty when computing aggregates. Note the log-scale on both the x and y axis. On small datasets, SQLite outperforms BTDS. However as the dataset grows, BTDS's ability to efficiently traverse pre-computed aggregates allows it to provide a constant query response time independent of the dataset size. Both SQLite, MySQL and OpenTSDB show drastically increasing delay as the data scales.

Next, we evaluate the ingest penalty required for fast range-based queries. Table II shows the total time required to ingest a 1M-point dataset. BTDS is approximately 1.7 times slower than SQLite and over 9 times slower than MySQL. The embedded BTDS target is more than 70 times slower than its server counter-part due to poor disk I/O performance. Since the embedded nodes are closest to the streaming data, it would be rare for the system to need to ingest a large (1 year) batch of data. The compression overhead on write ends up being approximately a 400% decrease in performance. As the database increases in size, BTDS must aggregate data across increasingly larger datasets. Figure 7 shows that the BTDS ingest performance scales as the dataset increases but rapidly levels off. MySQL, SQLite and OpenTSDB remain nearly constant since they are typically just seeking to the end of a structure and appending data. With a sensor input rate of 1Hz, the average ingest performance stabilizes at about half a year worth of data and still remains competitive with MySQL.

Database	1M-Point Ingest Time (seconds)
BTDS	373.03
BTDS Compressed	1503.32
BTDS Embedded	26404.00
SQlite	217.18
MySQL	41.12
OpenTSDB	6072.56

TABLE II: Ingest performance with 1K batches

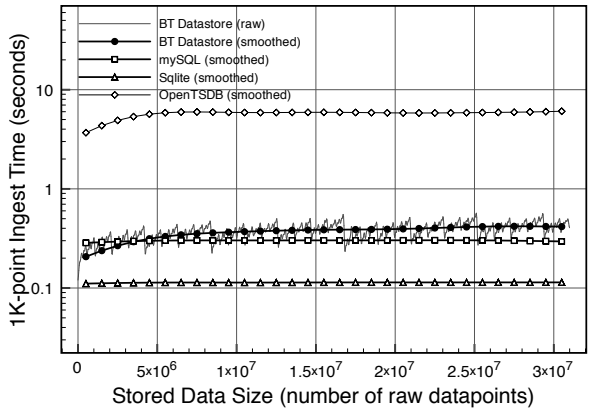


Fig. 7: Ingest Performance vs Datastore Size (log scale)

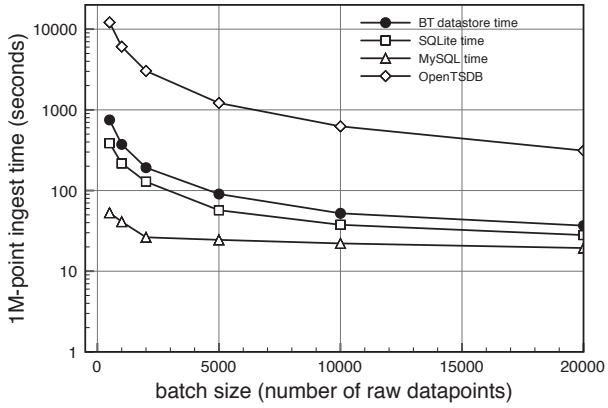


Fig. 8: Ingest Performance vs Data Batch Size (log scale)

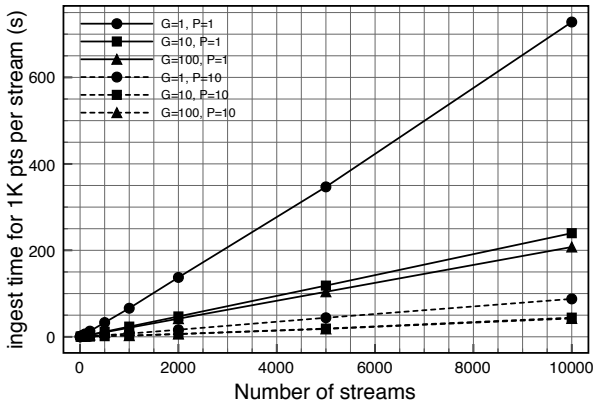


Fig. 9: Ingest Performance vs Number of Streams

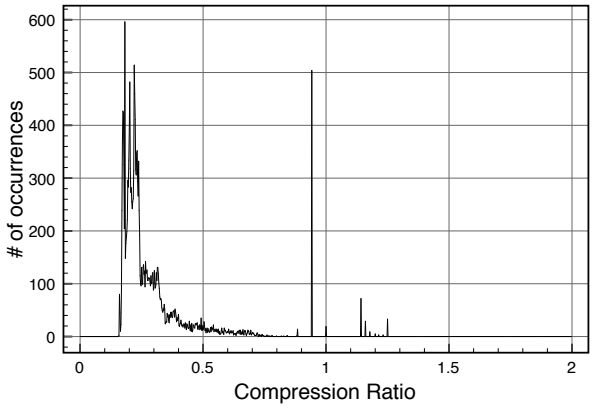


Fig. 10: Compression Ratio for Example Dataset

The distributed nature of Respawn also horizontally scales to alleviate central ingest bottlenecks.

One mechanism used to optimize reads and writes in streaming storage is batching of data. Figure 8 shows how each database scales in terms of ingest time as the batches of points increase in size. They all follow a similar trend and level off with approximately 1K data points. 1K data points on a 1Hz sensor feed corresponds to 15 minutes of data. There is a trade-off with how much a sensor should be willing to batch in memory and how soon data becomes available to incoming queries. Approaches like [22] have proposed making queries on both disk and in-memory to reduce those overheads.

Another technique for improving ingest performance is to group sensors that have identical timestamps. In BTDS, grouping can be used to save on the bookkeeping overhead normally required to setup a unique timestamp entry. Figure 9 shows the time required to ingest 1K-points per stream as the number of total streams is increased. Each line represents a different grouping size G and a different number of processor cores P used by the server. We see that grouping sensor improves efficiency up to a point, but saturates at about 10 inputs. We also see that BTDS becomes CPU limited since increasing the number of cores is able to double performance.

Figure 10 shows a histogram of the compression ratios achieved when running gzip compression on a 21GB repository containing 6 months of mixed time series sensor data. The data consisted of environmental sensors like light, temperature, motion, pressure, humidity and audio as well as a wide variety of HVAC sensors that monitor blower speeds and water intake temperatures. We see that in general, the data compresses to 25% of its original size, however the runtime performance overheads are non-trivial. As future work, we intend to investigate utilizing the tile structure of our database to compress only infrequently access tiles or ones with low-standard-deviation numbers that would achieve higher compression ratios.

B. Dispatcher

Next, we compare against the architectures shown in Figure 11. The first is a centralized architecture where all nodes stream data to the cloud for storage and there is no database running on the edge nodes. The second architecture, proxy, is one where storage exists on the edge nodes and requests are routed through the cloud. The final redirection architecture is the most distributed option, where a a lightweight dispatcher redirects requests either to the cloud or, in this case, only to edge nodes that are running a local copy of the database.

Figure 12 shows the limit on number of requests per second in each architecture. These lines were generated using a simulation based on hardware profiles. Networking overhead was determined by running requests from a separate machine against the database or dispatcher until its performance saturated. Real requests were generated and rerouted to closely mimic the real system. Since we do not have access to thousands of edge nodes, we assume that these requests can

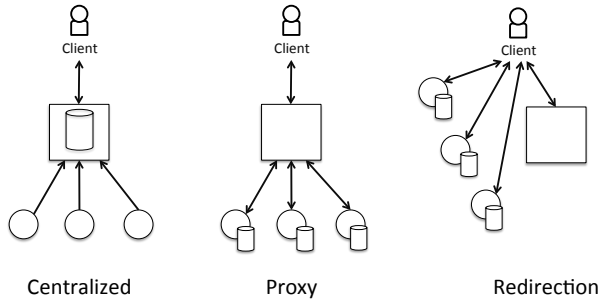


Fig. 11: Dispatcher Architectures

be serviced outside of the local network independently (i.e. requests are directly handed off to edge nodes do not interfere with each other). We also assume that the requests for each edge node arrive in an equally independent fashion.

The centralized storage architecture is limited by the number of simultaneous requests that can be made to BTDS (in this case 80 per second). As the number of nodes in the system scales, the performance gradually worsens as additional edge nodes consume the server's bandwidth. This approach, which is most common in simple sensor collection systems, scales very poorly. The proxy architecture shows improved scalability due to its ability to offload the majority of the database workload to the edge. The system becomes network bandwidth constrained at about 2000 requests per second.

For many applications, this approach is simple and able to handle medium workloads. The Respawn components are flexible enough that they can be easily configured to support this type of access pattern if so desired. The redirection and hand-off approach which best represents the standard Respawn configuration scales quite well supporting up to 16,000 requests per second. Eventually, the limiting bottleneck is the performance of the dispatcher. These operating points can be used as general rules of thumb to determine if a particular scheme meets the expected application demands.

C. Migration

In this section, we evaluate three different tile migration schemes. The first is a simple periodic migration technique where top-level aggregate tiles are periodically migrated based on available bandwidth of the edge node. Since each Respawn edge node is configured with a QoS limit by the user or is limited by the network connection itself, tiles are transmitted at a fixed rate so as to keep average throughput below the configured setting. The second scheme is a statistical pre-fetch mechanism that prioritizes which tiles get migrated based on standard deviation of the data. The intuition being that clients are more likely to investigate areas where the data is exhibiting significant changes. Again, these tiles are migrated within the specified QoS limits. The final scheme is a combination of the first two approaches. In this case, tiles are selected in a round-robin fashion from the periodic and the stddev-based transmit queues. The cloud caches data periodically at a lower granularity and also maintains a few regions of highly variable data ready to serve.

In order to evaluate these migration approaches, we performed a small user study where participants were asked to browse around a web-based plot of a week worth of time-series

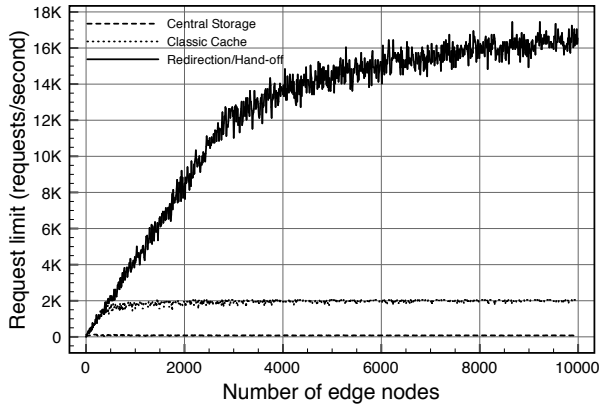


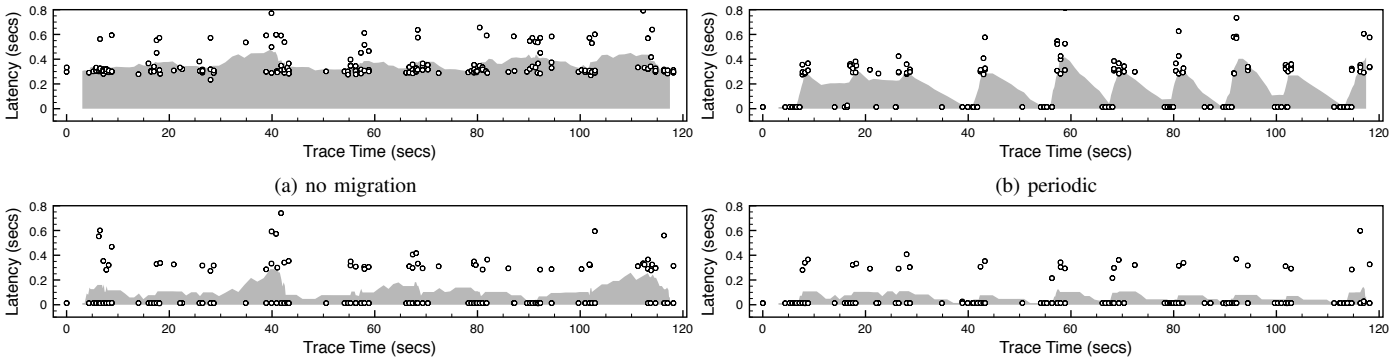
Fig. 12: Architecture Request Limit vs Number of Edge Nodes

data from environmental sensors placed in our lab. Users were asked to determine approximately how many hours in the last week a lab was occupied based on sensor data. This forced users to navigate and zoom into each region to determine the length of each section and if it was continuous. Each click and zoom selected on the plotting tool logged the corresponding range-based query to the database. These traces could then be played back against a version of the system executing each of the migration schemes.

Figure 13 shows the latency per time of a characteristic trace performed on the data. In (a), there was no migration. Each request shown by a small circle was redirected to an embedded cloud node that was running on the campus wireless network. These requests took on average 340 ms. The grey background shading shows a sliding window average of the latency to help visualize the impact of clusters of high and low latency requests. In trace (b), the system utilized periodic tile migration. Accesses that went to the cloud instead of the edge are shown with much lower latencies that correspond with connecting to a wired server machine on campus. In trace (c), the variance-based prefetching approach is used to migrate data with higher levels of variation, a scheme which has a slight performance increase over periodic migration. Finally, trace (d) shows both periodic and variance-based prefetching, which achieves the best performance at an average latency of 85 ms. In all of these cases, the migration bandwidth was kept constant, and 5% of edge tiles were migrated. We see that combining both approaches is most effective in our tests. This can be attributed to users making a reasonable number of low-resolution tile accesses to get an overview and subsequently "drilling down" into regions containing interesting features.

V. Conclusions and Future Work

In conclusion, in this paper we present a highly-scalable distributed time series datastore called Respawn. Respawn is able to run a small multi-resolution datastore on embedded leaf nodes that pre-processes incoming data in order to accelerate range-based queries. A high-speed dispatcher server is responsible for routing client requests to either a cloud storage backend or directly to the edge node. We propose and evaluate various data migration techniques that define how edge nodes should push tiles to the cloud backend to help mask latency for client request. This architecture is ideal for supporting interactive visualization tasks, allowing scientists to



(a) no migration

(b) periodic

(c) prefetch

(d) periodic and prefetch

Fig. 13: Trace latency given different tile migration schemes

easily browse large repositories of sensor data. Respawn allows large-scale sensing systems to horizontally scale in a manner that still provides responsive access to raw or aggregate data. Through micro-benchmarking, we show that Respawn is able to support range-based queries on large datasets with nearly constant access time while other commonly used databases scale exponentially. We also show that the data ingest penalty can be kept at levels similar to that of MySQL databases by batching and grouping data. We show that in terms of network scalability, that Respawn can support thousands of edge nodes handling tens of thousands of requests. Finally, we demonstrate how intelligent data migration can be used in cloud-to-edge systems to automatically mask latency. As future work, we intend to investigate different aggregation functions as well as more sophisticated data compression and aging techniques.

VI. Acknowledgments

This research was funded in part by the Intel Science and Technology Center on Embedded Computing, the Bosch Research and Technology Center in Pittsburgh and TerraSwarm, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. We would like to thank Ilari Shafer for the many interesting discussions about time-series databases, and David O'Hallaron for his insightful advice and feedback.

References

[1] Bodytrack: http://www.bodytrack.org/ (viewed 5/01/2013).

[2] Michael Stonebraker and Ugur Cetintemel. "one size fits all": An idea whose time has come and gone. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 2–11, Washington, DC, USA, 2005. IEEE Computer Society.

[3] Nihal Dindar, Peter M. Fischer, Merve Soner, and Nesime Tatbul. Efficiently correlating complex events over live and archived data streams. In *Proceedings of the 5th ACM international conference on Distributed event-based system*, DEBS '11, pages 243–254, New York, NY, USA, 2011. ACM.

[4] Magdalena Balazinska, Yongchul Kwon, Nathan Kuchta, and Dennis Lee. Moirae: History-enhanced monitoring. In *In Proc. of the Third CIDR Conf*, 2007.

[5] Slawek Smyl Charles Loboz and Suman Nath. Datagarage: Warehousing massive performance data on commodity servers. In *In Proc. VLDB*, 2010.

[6] Oetiker, T. RRDtool: http://www.mrtg.org/rrdtool (viewed 4/1/2013).

[7] Luca Deri, Simone Mainardi, and Francesco Fusco. tsdb: a compressed database for time series. In *Proceedings of the 4th international conference on Traffic Monitoring and Analysis*, TMA'12, pages 143–156, Berlin, Heidelberg, 2012. Springer-Verlag.

[8] A. Wilson J. Faden R.S. Weigel, D. M. Lindholm. Tsds: high-performance merge, subset, and filter software for time series-like data. In *Earth Science Informatics*, 2010.

[9] Sigoure, B. OpenTSDB. http://opentsdb.net/ (viewed 4/1/2013).

[10] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proc. VLDB Endow.*, 3(1-2):330–339, September 2010.

[11] Ramakrishna Varadarajan Nga Tran Ben Vandier Lyric Doshi Chuck Bear Andrew Lamb, Matt Fuller. The vertica analytic database: C-store 7 years later. In *VLDB*, 2012.

[12] Martin; Morrey III Charles B.; Veitch Alistair Anderson, Eric; Arlitt. Dataseries: An efficient, flexible data format for structured serial data. In *Tech. Rep. HPL-2009-323, HP Labs*, 2009.

[13] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: an architecture for global-scale persistent storage. *SIGPLAN Not.*, 35(11):190–201, November 2000.

[14] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, August 2003.

[15] Jianjun Chen, David J. Dewitt, Feng Tian, and Yuan Wang. Niagaracq: A scalable continuous query system for internet databases. In *In SIGMOD*, pages 379–390, 2000.

[16] John Heidemann, Fabio Silva, Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin, and Deepak Ganesan. Building efficient wireless sensor networks with low-level naming. *SIGOPS Oper. Syst. Rev.*, 35(5):146–159, October 2001.

[17] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):131–146, December 2002.

[18] S. Madden, M. J. Franklin, J. M. Hellerstein and W. Hong. TAG: A Tiny AGgregation Service for Ad-Hoc Sensor Networks. *Operating Systems Design and Implementation (OSDI)*, 2002.

[19] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Towards sensor database systems. In *Proceedings of the Second International Conference on Mobile Data Management*, MDM '01, pages 3–14, London, UK, UK, 2001. Springer-Verlag.

[20] Deepak Ganesan, Deborah Estrin, and John Heidemann. Dimensions: why do we need a new data handling architecture for sensor networks? *SIGCOMM Comput. Commun. Rev.*, 33(1):143–148, January 2003.

[21] Stephen Dawson-Haggerty, Xiaofan Jiang, Gilman Tolle, Jorge Ortiz, and David Culler. smap: a simple measurement and actuation profile for physical information. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, SenSys '10, pages 197–210, New York, NY, USA, 2010. ACM.

[22] James Cipar, Greg Ganger, Kimberly Keeton, Charles B. Morrey, III, Craig A.N. Soules, and Alistair Veitch. Lazybase: trading freshness for performance in a scalable database. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 169–182, New York, NY, USA, 2012. ACM.