# StreamFS

Jorge Ortiz
Computer Science Division
University of California, Berkeley
jortiz@cs.berkeley.edu

## 1 Introduction

StreamFS is a file system the represents the physical world as a collection of files.

Contributions:

- StreamFS treats metadata as a first-class citizen, allowing queries across snapshots of the structured, semi-structured, and timeseries data.
- The StreamFS view of the world revolves around signals or streams. Streams with history, both in value and in metadata. In StreamFS, a standard log file is decomposed into a collection of signals/streams files, whose associated pathnames and tags are maintained and querable in time.
- Job execution scheduling based to minimize calculation error, with backwards-updates for errors that occurred looking forward. Gives results with approximate errors.
- Non append-only streaming output is handled as a function that minimizes historical error calculations. [3]. You should use LCM to determine when the reverse scan and update will take place. The other trigger could be cumulative error – if it gets too high, run the process again on the historical data and update.

### 1.1 Sematics of a stream file

### 1.2 Inputting to a file with Regex and mapper

Use a regular expression and a map file to parse incoming data and put it in a file(s).

## 2 Introduction

Buildings consume 40% of the energy produced in the United States and nearly three quarters of the electricity produced [5]. It is conjectured that as much as 30-80% of their energy is wasted [2, 4]. With such astounding figures, buildings are a clear optimization target. In order to optimize building performance, we need fine grained visibility into the energy flow throughout the building. With over 70% of commercial buildings, 100,000 square feet or larger, having a building management system [**?**], the infrastructure to attain that level of visibility is available and widespread.

Building management systems (BMS) consist of thousands of sensors deployed throughout the entire building, measuring temperature, pressure, flow, and other physical phenomenon. They collect data from each of these sensors and present them to the building manager to show the current state of building and the system components within it. Despite this level of penetration and data availablity, energy-flow visibility is poor. This is mainly due to two reasons:

1. Building management system are designed for monitoring, not analytics.
2. Deep analysis requires careful meta/data management and processing to clean the sensor data.

In this paper we describe a system we have designed to address these short-comings. StreamFS is an analytical framework that uses an entity-relationship graph (ERG) to model the logical and physical objects and inter-relationships that inform the questions posed to the system about energy-flow. StreamFS provides traditional *slice and dice*, *drill-down*, and *roll-up* OLAP [**?**] operations; which come naturally from the graphical structure. It also informs our naming scheme, which we use to allow the user to see and manipulate the state of graph to capture the underlying physical and logical relationships and consequently update aggregate data calculations upstream.

Our naming scheme is hierarchically structured, like traditional filesystem naming, with support for symbolic links, allowing arbitrary links between sub-trees. We argue that this naming scheme is crucial, as it exposes the inter-relationships which inform aggregation semantics intended by the user. StreamFS distinguishes between nodes that represent streaming data sources in the real-world and those that do not. Those that do not, however, can be tagged as aggregation points. As part of the tagging processes, a user specifies the units of aggregation, with additional options for cleaning and processing.

We use StreamFS to organize and support applications using building data from three different buildings. The first one is a 110,000 square foot, seven-story building, the second one is an eleven-story 250,000 square-foot building, and the third is a 150,000 square-foot eleven-story building. Our contributions are:

- A naming scheme for physical objects and inter-relationship that is used to construct an entity-relationship graph.
- Use of the entity-relationship graph to provide OLAP *roll-up*, *drill-down*, and *slice and dice* operations.
- Show how sliding-window operations can be used on real-time data in combination with the entity-relationship graph to maintain accurate aggregates as the underlying objects and inter-relationships change.

We also discuss how we deal with the fundamental challenges that come with sensor data. Specifically, we address *re-sampling* and *processing models*. The incoming data does not have a common time source, so combining the signals meaningfully involves interpolation. There are various options that we provide for performing the interpolation, chosen by the user depending on the units of the data. For example, temperature data may involve fitting a heat model with the data to attain missing values in time. In addition, aggregation is done as a function of the underlying constituents: they can be combined arbitrarily, by adding subtracting, multiplying or dividing corresponding values. We provide an interface to the user that allows them to specify how to combine the aggregate signals as a function of the child nodes in the entity-graph. Futhermore, they can filter the data by unit. This kind of flexibility useful for visualizing energy consumption over time.

## 2.1 Entity-relationship model

Tracking the operational energy consumption of a building requires the ability answer a series of questions about energy flow – energy data aggregated across multiple logical classes to determine how, where, and how much is being used. Sometimes, it even involves extrapolating forward in time to estimate future consumption patterns that could influence immedaite decisions. The ability to *slice and dice* the data allows the analyst to gain better insight into how the energy is being used, where it can be used more effectively, and how to change the operation of the building – through better equipment or activity scheduling – in order to optimize and reduce its energy consumption. Below is a typical list of questions:

1. How much energy is consumed in this room/floor/building? On average?
2. What is the current power draw by this pump? cooling tower? heating sub-system? Over the last month?
3. How much power is this device currently drawing? Over the last hour?
4. How much energy have I consumed today? Versus yesterday?
5. How much energy does the computing equipment in this building consume?

Notice, these question span spatial, temporal, and other arbitrary aggregates – some physical, some categorical. There is also an implicit hierarchical aspect to the grouping, in some cases. For example, there are many rooms on a floor and many floors in a building. Naturally, to answer the first question we can aggregate the data from the room up to the whole building. This hierarchical relationship is not as evident in the HVAC sub-components specified in the second question. However, local hierarchically relationships *do exist*. For example, the cooling system consists of the set of pumps, cooling towers, and condensers in the HVAC system that push condensor fluid and water to remove heat from spaces in the building.

We can model this as a set of objects and inter-relationships which inform how to *drill-down*, *roll-up*, and *slice and dice* the data – traditional OLAP operations. The main difference between this setup and traditional OLAP is the underlying dynamics of the inter-relationships: objects, particularly those meant to represent physical entities, are added and removed and their inter-relationships change over time. *The natural evolution of buildings and activities within them makes tracking energy-flow fundamentally challenging*.

In this paper, we show how the entity-relationship model [**?**] helps simplify this problem, both as an interface to the user and a data structure for the aggregation processes. We argue that the use of this model is a cleaner fit for this application scenario because it captures important semantic information about the real-world; facts critical for picking which questions to ask and how to answer them. In contrast, it has been shown that a relational model loses this information [**?**].

## 2.2 An example

Lets examine the requirements for answering the first question. A building is unware that there are rooms. Typically spaces in a building are called *zones* and, at construction time, walls are added to make rooms within zones. This makes rooms an abstract entity, used to group associated items with respect to it. It also means we typically do not have a single meter that is measuring the energy of a room; it must be calculated from the set of energy-consuming constituents.

What are the energy consuming constituents of a typical room? It is the set of energy-consumers that are active within or onto the room. Broadly, it consists of three things:

- Plug-loads
- Lights
- HVAC

For simplicity of demonstration, lets consider only plug-loads. In our construction of an entity-relationship graph lets assume there are nodes for each plug-load item and each room. For the room in question, the relationship between the plug-loads and the room is child to parent, respectively. The total energy consumed by the plug-loads can be aggregated at the parent node, the room, so the user can query the room for the total. Over time, plug-loads are removed and added to/from the room, but the relationship does not change. This simplifies the query; to obtain the total consumption over time, the query need only go to the room node. The parent-child relationship informs which constituents to aggregate over time to calculate the total.

### 2.3 General Approach

To realize this design we need to maintain the entity-relationship graph, present it to the user in a meaningful way; allowing them to update it directly to capture physical state and relationship changes. We also need to use this graphical structure to direct data flow throughout the underlying network. This allows us to accurately maintain the running aggregates as the deployment and activities churn.

We present the graph to the user through a filesystem-like naming and linking mechanisms. The combination of a hierarhical naming scheme and support for symbolic links allows the user to access and manipualte underlying objects and relationships. Moreover, the underlying graph structure is overloaded with upstream communication mechanisms and buffering to allow data to flow from the data-producing leave nodes to the aggregation-performing parent nodes. Furthermore, the buffering lets us deal with the streaming nature of data flow from the physical world to StreamFS and lets us maintain a real-time view of energy flow in the system. Traversing the graph provides a natural way for the user to implicitly execute the OLAP operations necessary to give the user the kind of insight into energy usage in the building necessary to understand, optimize and reduce it.

## 3 Physical-data applications

Physical-data applications are applications built to provide insight about the physical world. They range from analytics and visualization to control and actuation. Analytics and visualization apps typically provide aggregates, statistical summaries, and future predictions about the behavior of the physical phenomenon being measured. The fundamental challenges for these application are dealing with flawed or missing sensor data, integrating smoothing and processing models to fill in gaps in the data, and maintaining consistency between the physical world and the representation of it within the processing system, particular as it evolves. *Context maintainence* is a fundamental challenge for correct interpretation of the physical data.

Control applications allow users to control their environment. Control applications are either built off an-

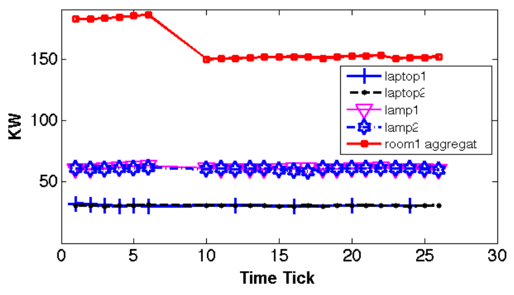

**Figure 1.**

alytics, providing intelligent, model-based control or they can provide simple, local, direct actuation by users through item or context-specific user interfaces.

Security is also a major concern in physical-data applications. Physical data reveals personal information about users through their affect on physical measurements. For example, light sensors and location information indicates occupation. Energy-consumption aggregates, when grouped by user or localtion can give information about the energy consumption habbits the occupants or users of those spaces.

### 3.1 Data organization, data services

### 3.2 Context consistency maintenance

### 3.3 Dashboards

### 3.4 Analytics

### 3.5 Control

## 4 Subscription semantics

A subscription expresses the intent to receive all associated streams whose topic is expressed by its hardlink name or any associated alias. Fundamentally, the subscriber expresses an interest in specific streams, referred to directly by name or indirectly by topic matching as it pertains to regular-expression name matching.

### 4.1 Continuous group-by for aggregation

Many of the application queries require aggregation of data over groups of streams. However, membership to those groups change over time, so aggregate recalculation must account for the changing constituents in order to maintain accuracy. Because of the way subscriptions are implemented in StreamFS, this is handled automatically. Recall, only streams whose tags match a subscriber's regular expression is forwarded to the subscription target. By removing a tag, you change group membership. Tag removal *can* change the membership of a stream to a group of streams defined by the regular expression. The data from streams belonging to the group are no longer accounted for in the aggregate calculation. The same is true when new members join the group. Aggregate calculation accuracy is maitained in either case.

Lets go through an example.

## 5 Stream processing

StreamFS allows users to perform stream processing on collections of streams.

Preserving timing semantics is challenging for two reasons:

1. NodeJs is single threaded; although it's an implementation issues, it's clear that the scheduler can only handle one job at a time, so threading is necessary
2. load is a metric of cpu activity, but the real metric we care about is expected versus actual start and completion time of a job.

The second point is the important factor that determines when a job gets moved to another server. We aim to maximize CPU utilization on a single server, without sacrificing timing constraints specified by the user. Job execution timing quality is measured by the absolute deviation of the completion time of a job from the scheduled completion time, that quality measure degrades as computing cycles become more scarce. In other words, the less cpu cycles there are to schedule the CPU-bound job, the larger the average error and variance of job completion times.

Now I have to show this experimentally and describe the methodology and job scheduling/migration algorithm.

## 6 Processes Job Scheduling

Lets assume we have a job $J$ that takes $T_c$ to complete is scheduled to run every $T_p$ seconds.

## 7 Process and sample scheduling

A job specifies both the report period and timeout time. The job also specifies, implicitly or explicitly, which sensor streams will be consumed. These parameters are used by the process manager for scheduling those streams such that 1) we maximize the number of jobs that make use of a report and 2) we minimize the error introduced by waiting for a report; interpolating later in time, introducing more error. The former keeps the CPU more highly utilized doing active work, the latter allows jobs to run on the freshest data (or its closest approximation).

The driver has two reporting modes. The first is for sensors whose report period can be set and the other is for those whose report period cannot be set. The driver decouples the report capabilities of the underlying hardware from the report mechanism to the process manager. This allows for scheduling flexibility in the process manager. We argue that this is a critical design choice, for processing efficiency, when dealing with the integration of heterogenous physical data streams. In addition, we embedded a simple linear interpolation process in the driver code itself. This gives the scheduler flexibility in choosing the data value from the sensor for timestamp alignment with the other sensors. We observe that readings, in close temporal proximity, are linearly associated with prior readings; especially as the temporal distance decreases.

Jobs in the system specify the report period of the job and the $k$ streams it consumes. Data streams fall into two categories: those that can be scheduled and those that cannot. For those that can be scheduled the questions are how do we ...

1. schedule their report period to maximize overlap with the job report period?
2. schedule their report period to minimize the wait time between the actual reading and the report time?

By default, drivers linearly interpolate pairs of readings. We assume, for simplicity, that 1) a reading contains no error and 2) the longer we wait, the larger the error. The error is proportional to the amount of time since the last reading, times a constant. The constant is stream specific. We capture this, using a stream-specific weight constant. For streams that cannot be scheduled, we ask, how do we ...

1. set their report periods to overlap as much as possible with jobs that use them.

### 7.1 Schedule Overlap Maximization

There are $n$ jobs, each with a period $P_i$ for some job $j_i$. Each stream, $j_i$, consumes a set of streams $K_i$. In this scenario, each stream $s \in K_i$ can be scheduled with any period $p$ and start time, $t_{init}$.

#### 7.1.1 *Variable Sensing Schedule*

#### 7.1.2 *Fixed Sensing Schedule*

Let $t_i$ be the next time the job will run.

#### 7.1.3 *Error Minimization*

## 8 Mixed Strategy

### 8.1 Experimental results

## 9 Filesystem for streaming data

A file system provides an abstraction for managing a collection of bytes on disk. The standard file itself provides logically sequential access to bytes distributed through the disk. Directories provides a simple container mechanism for grouping together files that are in some way related to each other. We think deployment metadata and data can similarly benefit from a simple hiearchical naming structure. Grouping items that share overlapping name-elements separated by forward-slashes in their tags. It provides a simple way to specify groups of streams as the unit of access or processing. File systems have also developed a convenient set of tools for quickly locating information, sharing reading, and pipelining processing; simple tools that fit the access, sharing, and processing needs for local-area sensor deployments. Filesystem also provide a security model baked into their construction. We examine the filesystem interface to StreamFS and examine how well the interface fits the needs for the class of applications StreamFS supports.

### 9.1 Naming and access

For local-area deployments, where spatial or categorical referral patterns are common in the tagging structure, hierarchical grouping provides a good, simple fit for naming objects. However, we separate logical grouping from physical, and by doing so, also separate naming and access. This prevents overly-stringent access patterns that may degrades performance over time. How the data is accessed is independent of how the object named. It also support multiple names without affecting without affecting access.

Tags are abitrary, but forward-slash separated naming conventions map cleanly into a filesystem naming. Each string between the forward slashes is treated like a directory in the filesystem. If a user attempts to create a file with an element in the name that is already set for a non-container object in StreamFS, then creation of the file is not allowed.

### 9.2 File types: directories, regular, special, pipes

### 9.3 Filesystem tools: grep, find, ls

## 10 Publish/subscribe model

Eugster et al. [1].

StreamFS uses a flexible flavor of the publish/subscribe model in order to support a wide range of applications. Publish/subscribe is necessary is physical data application development in order to scale in the number of supported applications. The publish/subscribe model used in StreamFS provides mechanisms that enable a flexible combination of space and time decoupling that enable StreamFS to support of a wide arrange of application requirements.

Our pub/sub engine is also tightly coupled with the namespaces expose to users, and this design choice allows an application to control the space coupling between the publisher and the subscriber (similar to TIBCO [**?**]).

### 10.1 Space decoupling

By its very design, space decoupling is achieved. Publisher do not hold a reference to the subscriber and subscribers do not hold references to publisher. However, because of the coupling of a full pathname and an object, subscriptions to topics expressed as a full pathname refer to the single publisher.

### 10.2 Time decoupling

Time decoupling is achievable through the timeseries data store. Publisher push data to StreamFS whether or not subscribers are online. Moreover, data may be received at the subscriber even if the publisher becomes disconnected. Currently, subscribers do not receive all information that was missed. In order to achieve fill time-decoupling, we allow the subcription target to enable or disable the option to buffer all missed readings for an associated subscription target, while the subscription target it offline.

### 10.3 Synchronization decoupling

Sychronization decoupling is achieved by the publisher and subscribers through StreamFS. Events are received out of sequence from their arrival to StreamFS. This is true even when the subscription target is a processing element. The thread that buffers incoming data for each processing element is seperate from the thread where the process is executed.

## 11 Security

### 11.1 Confidentiality

With SSL over Https for rest interface. With SSL flat over standard sockets.

### 11.2 Authentication

Public-private key encryption used for authentication:
$A \rightarrow Msg(Username, Public\_key\_A) \rightarrow B$
$A \leftarrow Msg(Enc\_Pub\_A(APP\_ID)) \leftarrow B$

1. Logging in should be an done over a secure socket and include the "username" and "password" of the user. If successful, the server will return an "sid" that has a TTL of 24 hour idle time.
2. Over secure socket "query_params" in json object (or query parameters in the https request) should include the "sid" (session identifier), returned from the server after sucessfully logging in a creating a new session.

    - confidentiality provided by secure socket layer
    - the "sid" provides a ticket for a previous successful login.

The key is that we want to determine who is making the request so we can determine whether to satisfy that request.

### 11.3 Access control and privacy

ACLs. (Username, APP_ID)

Users can create various applications, giving each application a unique identifier. This allows users to share data explicitly between applications. There are 5 levels of access:

1. Group: groups of (users,application) tuples
2. User: identified by (name, public key)
3. application
4. resource
5. operation (read/get, write/post,put, execute/put,post)

Application boundaries are useful for separating data-only (read/write only data) application from control applications. For example, we have one application that wants to display the total energy consumed by the lights in the building and it even writes the aggregate light energy consumption data to a resource and makes that resource available. The light-control application also read data coming from the lights through resources owned by the light data application, but it produces information about how it's controlling the light accordingly that it does not want other application to view. In this case the light data app shares its resources with the light control app, but not the other way around, even though both apps were written by the same user. In addition, the user is relieved from managing which collections of resources should not be read by the light-data app, resting assured that anything owned by the control app is not accessible by the light-data app.

## 12 References

[1] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

[2] N. Gershenfeld, S. Samouhos, and B. Nordman. Intelligent infrastructure for energy efficiency. *Science*, 327(5969):3, 2010.

[3] T. M. Ghanem, A. K. Elmagarmid, P.-A. Larson, and W. G. Aref. Supporting views in data stream management systems. *ACM Trans. Database Syst.*, 35(1):1:1–1:47, Feb. 2008.

[4] Next10. Untapped Potential of Commericial Buildings: Energy Use and Emissions, 2010.

[5] U.S. Environmental Protection Agency. Buildings Energy Data Book, 2010.