NilmDB: The Non-Intrusive Load Monitor Database

James Paris, John Donnal, Steven B. Leeb

***Abstract*—This paper presents NilmDB, a comprehensive framework designed to solve the "big data" problem of non-intrusive load monitoring and diagnostics. It provides the central component of a flexible, distributed architecture for the storage, transfer, manipulation, and analysis of time-series data. NilmDB is network-transparent and facilitates remote viewing and management of large data sets by utilizing efficient data reduction and indexing techniques.**

Energy monitoring and smart grid applications have rapidly developed into a multi-billion dollar market [1]. The continued growth and utility of monitoring technologies is predicated upon a necessity to economically extract actionable information from acquired data streams. User and operator needs define the nature of relevant information regarding power consumption and operation of the distribution system. The scale of this information can vary greatly in time, frequency, and amplitude or dynamic range. Basic energy-scorekeeping might be accomplished with time series data of real and reactive power consumption; essentially, information at or near line frequency. Power quality monitoring might require knowledge of line current and line voltage harmonics an order of magnitude higher in frequency. Diagnostic monitoring might require knowledge of non-integer-multiple frequencies of the line, e.g., tracking the principal slot harmonic of an important rotating machine. All of this data, and other streams as well, might be needed on time scales ranging from fractions of a second, for a transient study, to months or years, for energy scorekeeping and behavior tracking.

One of the largest roadblocks to effective analytics for power data arises from the disparities of scale inherent in data collection and processing, which often limits the speed and resolution at which data can be captured and effectively managed. This, in turn, affects the ability to extract actionable information from the data. The failure of several high-profile attempts at delivering such analytics to users, such as Google PowerMeter and Microsoft Hohm [2], demonstrates the difficulty of the problem. Similarly, "smart meters" may in many cases ultimately prove too limited in their feature set, communication requirements, and adaptability to justify their installation expense.

The non-intrusive load monitor (NILM) has been shown to be an effective and efficient energy monitoring system, and has been applied to a wide variety of systems [3]–[6]. With its ability to perform high-speed and high-resolution data acquisition, NILM also provides significant advantages for condition-based monitoring and diagnostics. This paper presents NilmDB, a comprehensive framework for solving the "big data" analytics problem of energy monitoring applications [7]. NilmDB is a network-enabled database that supports efficient storage, retrieval, and processing of vast, timestamped data sets. It allows a flexible and powerful separation between on-site, high-bandwidth processing operations and off-site, low-bandwidth control and visualization, through the use of unique indexing and data compression techniques. Specific analysis of NILM data can be performed as data is acquired, or retroactively as needed, using short filter scripts written in Python and transferred to the monitor. The NilmDB framework takes advantage of inexpensive contemporary computing to place adaptable processing power at locations in the utility best suited to minimize communications requirements, while preserving almost unlimited data analytics flexibility.

Large-scale power monitoring analytics poses a variety of unique challenges related to storage, transfer, and processing of data. Specific challenges include:

1) **Data Relationships**: Tracking the interrelationships between independent data streams, both raw and processed.
2) **Data Uniformity**: Managing the non-uniformity of data rates, and the potential unreliability of data capture across larger systems.
3) **Efficient Storage**: Supporting large amounts of stored data, which can easily exceed hundreds of billions of samples.
4) **Analytic Flexibility**: Supporting the extraction, and processing, and insertion of arbitrary time-ranges of data.
5) **Network Transparency**: Connecting distributed monitoring systems and users.
6) **Simultaneous Access**: Supporting simultaneous, error-free access from multiple programs and users.
7) **Data Visualization**: Developing compression methods and storage techniques to permit fast, efficient visualization of data with minimal network bandwidth usage.
8) **Disaggregation**: Extracting information about individual loads from aggregate data, and supporting diagnostics that target these loads.

The NilmDB framework runs on any conventional Linux platform, from a desktop to a Raspberry Pi [8]. It provides provides powerful solutions to all of these challenges, as detailed in the following sections.

I. Data Relationships: Streams

Complex relationships often exist between collected data sets. At the acquisition stage, a load monitoring system must track each data source and its relationships to other data. For example, three voltage waveforms may be related in that they are three phases of the utility supply. Three currents may represent the current draw from that utility, for a particular motor. At a higher level, a storage system should be able to group motors, perhaps by room or building. Computed metrics, such as total power computation and power harmonics, also require such details to be tracked.

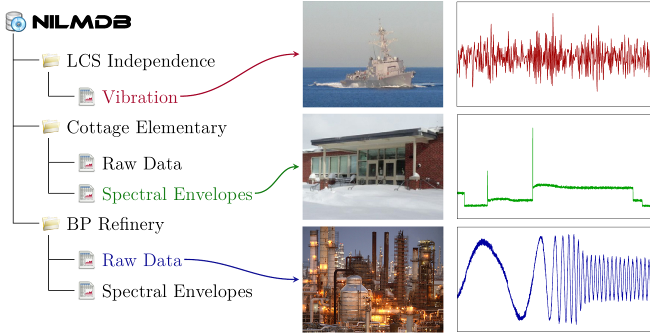

Fig. 1: NilmDB data streams contain homogenous data in a wide variety of formats. Streams are organized in a tree-like hierarchy.

NILMDB
School
Prep
Sinefit
Raw
Home
Pump
Raw
Temperature

Timestamp	φA	φB	φC
⋮	⋮	⋮	⋮
15:30:00.000000	34768	27673	35069
15:30:00.000125	34979	27777	34882
15:30:00.000250	35167	27732	34669
15:30:00.000375	35384	27887	34433
15:30:00.000500	35579	27965	34232
15:30:00.000625	35782	28083	33932
15:30:00.000750	36032	28032	33858
15:30:00.000875	36374	28072	33641
⋮	⋮	⋮	⋮

Fig. 2: Data contained within a NilmDB stream. A stream contains a fixed number of columns of a homogeneous type, and can be conceptually viewed as a table with an unbounded number of rows. Each row holds a single unique timestamp and the data for that time.

NilmDB achieves this by organizing all data in "streams". Streams contain time-series data from a particular source. This data can represent physical quantities, computed values, or any other timestamped information. Examples include voltage, current, temperature, vibration, spectral envelopes, system run-time and health metrics, error and event indicators, etc. Streams are organized and identified in the database using a tree-like path structure that mirrors an arrangement of files and folders, as shown in Fig. 1.

Streams contain recorded or generated waveforms. They can be viewed conceptually as large tables of data, as shown in Fig. 2. Each row contains a unique timestamp and data that matches the stream's layout, which is determined when the stream is created and indicates the number of columns and their data type.

Besides timestamped data, NilmDB also supports storing metadata with any number of arbitrary key/value pairs for each stream. These key/value pairs can be used for any ancillary information that should be stored alongside the stream. For example, a "scale_factor" key with a value of "1.337" might be used to indicate a conversion ratio. Other uses for metadata include adding appropriate labels for columns, or denoting the source from which a processing filter read its input.

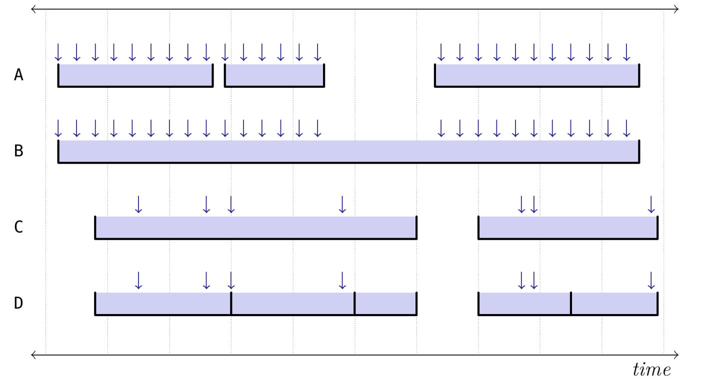

Fig. 3: Data intervals within a stream. Intervals mark ranges of time for which data is present; within these ranges, individual timestamped data values can be stored. Here, `A` and `B` contain the same data, but different intervals; `C` and `D` are equivalent.

II. Data Uniformity: Intervals and Timestamps

The data stored in streams may be non-uniform in time. For example, a remote acquisition source that nominally captures at 8 kHz may vary sample rate slightly with temperature or battery state. Streams containing data like spectral envelope harmonics typically store one sample per line period of the utility voltage waveform, which can vary greatly, particularly on isolated generator systems. Other types of load-monitoring data may not follow any regular pattern at all, such as a stream that identifies the turn-on events of a particular load.

Stream data may also be non-uniform due to unreliability of the capture or data transfer process. A system that collects data from remote sensors may periodically lose connection to those sensors, leading to missing periods of data. A robust data analytics system should handle these cases, allowing filters and other processing to easily identify regions of time for which data is and is not available.

In NilmDB, the time coverage of stream data is managed through the tracking of non-overlapping data intervals, and every sample of data within these intervals carries a unique timestamp. Fig. 3 demonstrates four streams, their intervals, and the data samples contained within those intervals. For every half-open interval $[\mathrm{S} \rightarrow \mathrm{E}\rangle$, the timestamp t of any sample stored within this interval satisfies the relationship

$$S \leq t < E. \tag{1}$$

Two streams can differ based on their intervals even if they contain the exact same data samples, as demonstrated in Fig. 3 in streams `A` and `B`. Similarly, two contiguous intervals $[T_1 \rightarrow T_2\rangle$ and $[T_2 \rightarrow T_3\rangle$ are functionally equivalent to the one long interval $[T_1 \rightarrow T_3\rangle$, as shown in streams `C` and `D`.

Intervals in a stream are immutable. Creating an interval and inserting data into that interval is a combined operation; once created, no data can be added to, or removed from, that particular interval. Instead, new non-overlapping intervals can be created in the same manner, or intervals or segments thereof can be removed together with their data. These constraints allow for efficient storage, lookup, and retrieval of data within NilmDB.

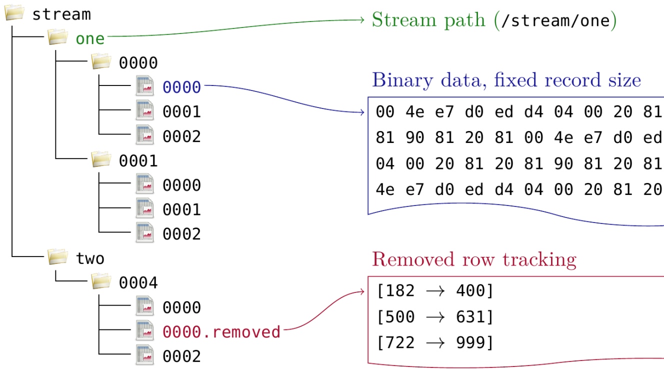

Fig. 4: The “Bulkdata” storage system. Stream data is stored in fixed-size files, in a multi-level structure that prevents any individual directory from growing too large. Removed data segments are tracked so that their space can be freed.

III. Efficient Storage: Bulkdata

Low bandwidth, higher-level data in NilmDB, such as the list of intervals for a particular stream, the stream names, and metadata variables, are stored in a standard SQL relational database. However, standard database engines are unable to store the vast quantities of data generated by a typical non-intrusive load monitoring system, which can easily exceed 10^{12} samples per monitored system, per year, equivalent to terabytes of data. Instead, the majority of NilmDB data is handled by the “bulkdata” storage system. Bulkdata is an addressable row store, meaning that each sample of NILM data is stored under, and can be retrieved by, a unique row number. It provides three fundamental operations:

- Extract data from a specified row number.
- Insert new data, and return the row numbers corresponding to the starting and ending rows of that new data.
- Remove data corresponding to a range of row numbers.

The rows of data are stored as raw binary on disk. The format of each row is derived from the stream layout, and each row of a stream takes up a fixed number of bytes, denoted $B_{\text{row_size}}$. The structure of the bulkdata storage is shown in Fig. 4. Stream paths, such as “/stream/one”, are used as a directories in the filesystem. The data itself is stored in numbered files inside these directories. The number of rows in each data file, $N_{\text{rows_per_file}}$, is determined automatically at stream creation time, based on stream layout, so that the data file size is approximately 128 MiB. Depending on the underlying operating system, there may be a limit to how many files can efficiently be stored in a single directory. To avoid this issue, the data files are further grouped inside numbered directories. The group size is typically $N_{\text{files_per_dir}} = 32768$.

A. Bulkdata Row Extraction

Extraction of data from a particular row or range of rows is straightforward. Because each row takes up a fixed amount of space in the binary storage, the location of row n in a given stream is fixed. The group directory number is given by:

$$\text{group_num} = \lfloor \lfloor n/N_{\text{rows_per_file}} \rfloor / N_{\text{files_per_dir}} \rfloor \quad (2)$$

Similarly, the file number is:

$$\text{file_num} = \text{mod}\left(\lfloor n/N_{\text{rows_per_file}} \rfloor, N_{\text{files_per_dir}}\right) \quad (3)$$

The specific offset of the row in the file is:

$$\text{file_offset} = \text{mod}(n, N_{\text{rows_per_file}}) \cdot B_{\text{row_size}} \quad (4)$$

The binary data for the requested is then read out from the $B_{\text{row_size}}$ bytes at offset $\langle\text{file_offset}\rangle$, in the file $\langle\text{file_num}\rangle$, in the directory $\langle\text{group_num}\rangle$.

B. Bulkdata Row Insertion

The bulkdata storage is append-only; that is, all newly inserted data is appended to the last existing file, and the corresponding row numbers for the new data will be greater than any other data in the stream. The system tracks the maximum row number N_{max} ever used for a particular stream. When inserting m rows of data, the new data is written to file offsets corresponding to row $(N_{\text{max}} + 1)$ through $(N_{\text{max}} + m)$, with file offsets calculated as they were for data extraction, and N_{max} is updated accordingly.

C. Bulkdata Row Removal

Every data file, such as “0000”, can have an associated row tracking file, “0000.removed”. This file contains a serialized representation of a list, created in the Python “pickle” format [9]. Each entry in this list is a pair of row numbers [*start*, *stop*], indicating a range of “removed” rows that are no longer referenced by any intervals and will no longer be accessed. As more rows are removed, more entries are added to this file. Finally, when every row in a particular data file has been marked as removed, both the data file and its tracking file are completely deleted from disk, freeing the space previously used.

IV. Analytic Flexibility: Interval Tree and Data Extraction

Data analysis and processing within the NilmDB framework is accomplished through the use of filters, which query NilmDB in order to extract and process data from arbitrary time intervals. The database provides analytic flexibility in this regard, by being able to efficiently supply any requested time range $[S_R \rightarrow E_R\rangle$ of data, even if this time range does not match up directly with one of the previously-inserted intervals of data in the stream.

The arbitrary data extraction is based on the stream interval tracking and the stream data timestamps. Rather than searching through each interval in a linear fashion to locate the requested data, or creating and maintaining a database index on the timestamps, extraction is performed quickly and efficiently by first utilizing an interval tree, as shown in Fig. 5. The interval tree is a red-black tree, a form of binary search tree with a per-node “coloring” [10]. The coloring is used to guarantee a balanced structure by maintaining the following invariants:

- The root node is black.
- If a node is red, its children are both black.

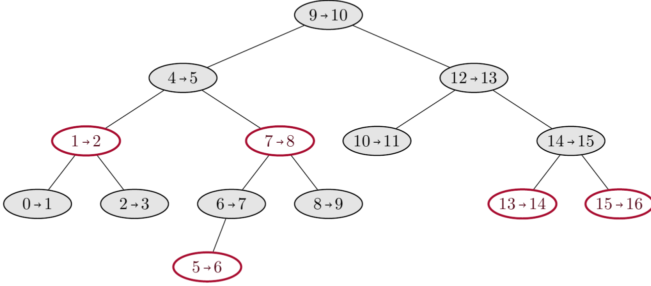

Fig. 5: Red-black interval tree, used to efficiently locate and manage the intervals in which data is stored. The structure is maintained such that the height of a tree with n nodes is at most $2 \cdot \log_2(n+1)$.

function ExtractData(S_R, E_R, I)
  result ← [ ] ▷ Initialize empty result list.
  for all intervals i **in** I **do** ▷ For each interval.
    $(S, E, \rho_S, \rho_E) \leftarrow i$ ▷ Get interval parameters.
    $\rho_{S_R} \leftarrow$ LocateTime(S_R, ρ_S, ρ_E) ▷ Locate start.
    $\rho_{S_E} \leftarrow$ LocateTime(S_E, ρ_S, ρ_E) ▷ Locate end.
    for $n \leftarrow (\rho_{S_R}$ **to** $(\rho_{S_E} - 1))$ **do**
      result ← result + bulkdata[n]
  return result ▷ Return all matched rows.

Algorithm 1: Data extraction from an arbitrary time range $[S_R \rightarrow E_R\rangle$, given the set I of all stream intervals that intersect this range.

- All paths from the bottom of the tree to a particular node contain the same number of black nodes.

When inserting or deleting nodes from the tree, these invariants can be maintained in $O(\log_2 n)$ time by recoloring and moving nodes as necessary [10]. The maximum height of a balanced tree with n nodes is at most $2 \cdot \log_2(n+1)$, and so search operations also take $O(\log_2 n)$ time. Thus, the time needed to insert, remove, and locate a specific interval in a stream grows with the logarithm of the number of intervals present. This is used to efficiently locate the set I of all intervals that intersect the requested time range $[S_R \rightarrow E_R\rangle$.

Given I, the NilmDB database layer proceeds to find the bulkdata rows, corresponding to these intervals, that specifically contain timestamps in the range $[S_R \rightarrow E_R\rangle$. Since any particular interval in the bulkdata storage stores only monotonically-increasing timestamps, the rows can be found efficiently with a binary search. The algorithms ExtractData and LocateTime, shown in Algorithms 1 and 2, are used to perform this search. They return the rows of data corresponding to the requested interval, completing the data extraction.

V. Network Transparency: Client/Server Model

Modern energy monitoring systems are increasingly distributed in nature. Sensors and data acquisition systems may involve a variety of network-connected or wireless components that are often capable of doing nontrivial computation themselves. As the complexity of data processing and analytics grows, the use of distributed or "cloud" computing models is crucial to maintain required performance levels.

function LocateTime(t, ρ_S, ρ_E)
  $\rho_{\text{low}} \leftarrow \rho_S$ ▷ Initial search range is all of the rows
  $\rho_{\text{high}} \leftarrow \rho_E$
  while $\rho_{\text{low}} < \rho_{\text{high}}$ **do** ▷ Repeat until row is found
    $\rho_{\text{mid}} \leftarrow \lfloor(\rho_{\text{low}} + \rho_{\text{high}})/2\rfloor$ ▷ Find midpoint
    if bulkdata[ρ_{mid}].timestamp $< t$ **then**
      $\rho_{\text{low}} \leftarrow \rho_{\text{mid}} + 1$ ▷ Narrow search to right half
    else
      $\rho_{\text{high}} \leftarrow \rho_{\text{mid}}$ ▷ Narrow search to left half
  return ρ_{low}

Algorithm 2: Binary search to locate the first bulkdata row in the range $[\rho_S, \rho_E\rangle$ with a timestamp greater than t.

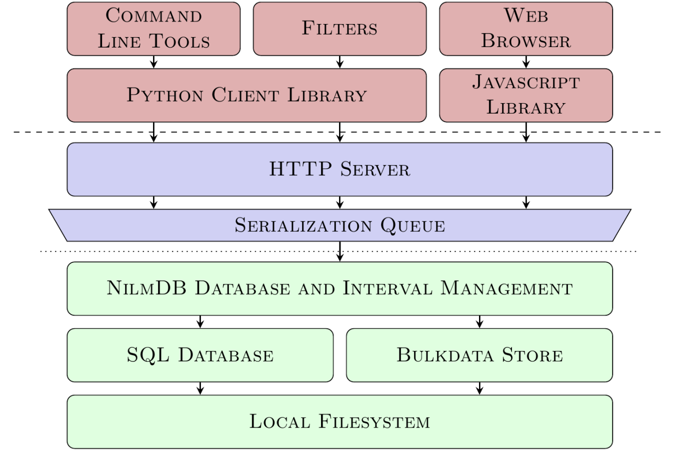

Fig. 6: NilmDB system architecture. Client programs and filters interact with NilmDB via Hypertext Transfer Protocol (HTTP), which supports local and remote connections equally well.

NilmDB supports these forms of networked computing by following a client/server model. The general architecture of the NilmDB server is shown in Fig. 6. Multiple clients, or end users, can access the server, and perform requests and actions. Users and systems can utilize a variety of interfaces to communicate with the server, such as command-line or web-based applications. Regardless of the source, all interaction with the NilmDB server eventually takes place through a standard HTTP/1.1 compliant interface [11]. The HTTP defines methods that perform actions on particular resources, which are identified by Uniform Resource Locator (URL). These actions correspond to the fundamental NilmDB operations, such as creating a stream, listing available intervals, and inserting or extracting data.

In order to fully support a distributed computation model, the NilmDB framework includes a secondary server, NilmRun, which gives clients the ability to control the execution of software on remote NilmDB hosts. For example, consider a system where a remote NilmDB server is collecting and storing data, and a local client wishes to manipulate this data in some way, to extract a single metric. Using NilmRun, the client

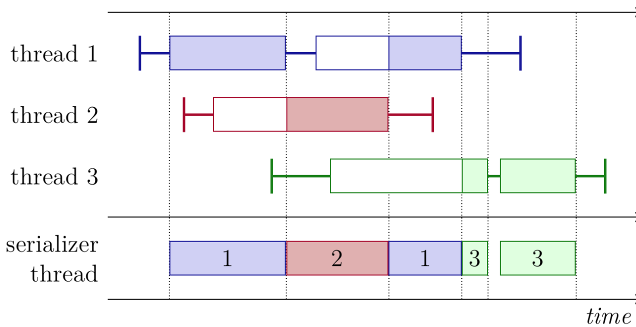

Fig. 7: Serialization of database operations. Incoming HTTP requests are handled by a multi-threaded server, which performs lower-level database operations through a serializer thread. All operations are then run in a single thread, in the order they were enqueued.

can transmit short processing filters to the remote machine, execute them there, and retrieve the status and results. This conserves bandwidth, and increases throughput, compared to the client pulling down the data and computing the metric itself.

The networking capabilities of NilmDB are heavily used by NILM Manager, which is fully described in [12]. NILM Manager is a unified and centralized management infrastructure for the distributed NilmDB system. It connects to remote NilmDB systems through a secure virtual private network (VPN), providing a simple, user-friendly and web-based interface to any authenticated user with an Internet connection. This interface includes a wide variety of tools, including configuration of servers and streams, real-time data visualizations, and the interactive testing and development of new data processing filters.

VI. Simultaneous Access: Serialization

Support for simultaneous access from multiple clients is a requirement for real-time load monitoring applications, as analytics must continuously be performed while data is being captured. In order to support this use case, the HTTP server interface of NilmDB is multi-threaded, and supports client connections and requests that can come in at any time. Receiving these requests, or sending responses, might proceed slowly, depending on network conditions, and so the server supports and can transfer data on any number of simultaneous connections. However, operations that modify the database state generally need to be performed one at a time, in order to maintain internal consistency of the stored data and state. While complex fine-grained locking and ordering may allow some database operations to run concurrently, NilmDB uses a more straightforward approach where direct database access must be performed from one thread only.

Single-threaded access is accomplished through the use of the "serializer" module, which allows any running thread to enqueue a function call "request" and wait for the result. When the database is not busy, the serializer will retrieve the earliest request from the queue, perform its function call in a single,

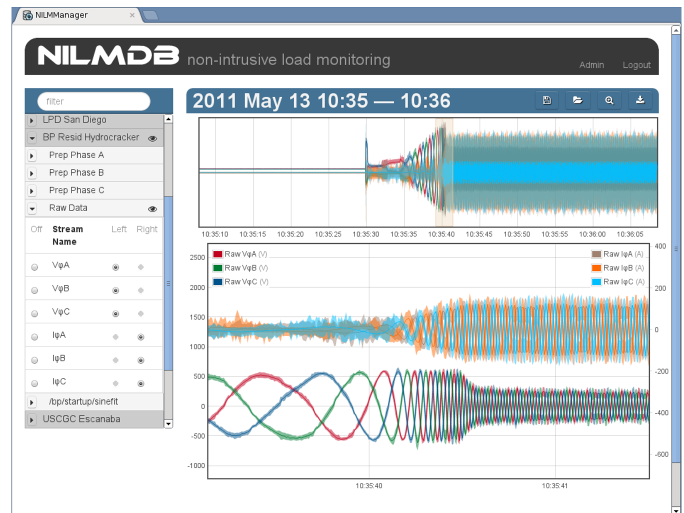

Fig. 8: Visualization of NilmDB streams through the NILM Manager. The NILM Manager provides a user-friendly interface that allows interactive nagivation and zooming of arbitrary NilmDB data streams.

global thread, and return the result to the original thread. Thus, the serializer ensures that any operations on the database are serviced one at a time, in first-in, first-out (FIFO) order. The process of running queued requests in the serializer thread is shown in Fig. 7.

Some HTTP requests may take an unbounded amount of time to complete, such as extracting data from a large interval. To ensure fairness between clients, the NilmDB database layer may choose to only perform a portion of the requested operation before returning, which gives other threads that are waiting for the serializer a chance to run. The HTTP server handles such occurrences automatically, resubmitting the remainder of the operation to the serializer until it is complete. The client sees it as single HTTP request and response.

VII. Data Visualization: NILM Manager and Decimation

Visualization of data provides significant benefits to load monitoring. In the installation and testing phases of a non-intrusive load monitor, the ability to visually explore stored data enables real-time refinement of sensor networks. During the research and development of specific models, performance metrics, and reports, a visual display of initial, intermediate, and final processing results provides significant insight into the analytics under development. For end-users, the ability to plot and explore data trends, energy usage, reports, and diagnostic indicators leads directly to actionable results.

The NILM Manager, shown in Fig. 8, provides a powerful data visualization and nagivation interface for NilmDB systems [12]. This interface forms a central component of the manager, and is used both as a standalone tool for exploring NilmDB streams, and as an embedded component for controlling and visualizing the output of other tools. The features of the plot engine include:

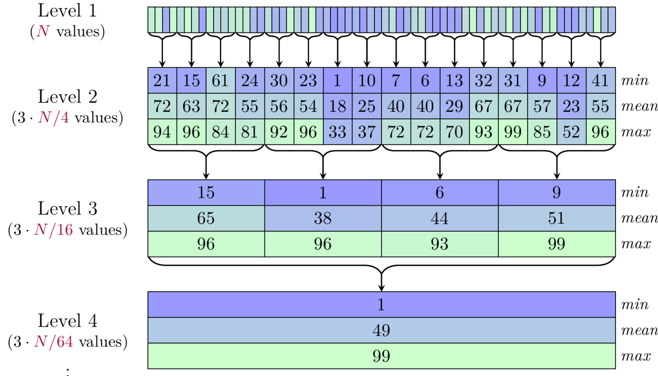

Fig. 9: Decimation of stream data. Each decimation level tracks the minimum, mean, and maximum of a block of values from the previous level. The total storage requirement for N original samples is only $2N$, regardless of the number of levels.

- Live, draggable, zoomable plots with "Google Maps-like" navigation.
- Dual y axes with independently adjustable scaling and positioning.
- Simultaneous plotting of data with compatible units.
- Overview window to facilitate navigation of large data streams.
- Antialiasing of high frequency data through shading of signal envelopes.
- Support for discontinuous data and gaps in streams.

Crucially, the plotting engine achieves these features while transferring only a minimal amount of data from the remote NilmDB server. Typically, the number of data points retrieved to display a particular window of data is on the order of 1,000 per plotted stream, regardless of zoom level.

The plotting engine achieves these low data transfer rates in two ways. First, it makes heavy use of the stream and interval support of NilmDB, particularly when extracting data. For example, when extracting data corresponding to the currently displayed x-axis, the server manages all details of finding and returning only that data which is both present and needed.

The second feature that enables efficient plotting is decimation. Here, decimation is a process by which ancillary streams of filtered, downsampled data are pre-computed and stored on the server, similar to the computer graphics technique of "MIP mapping" [13]. An example of the decimation process is shown in Fig. 9.

Decimated streams store the mean value for each column of an input stream, including the timestamp, calculated over small successive blocks of γ rows. Typically, $\gamma = 4$. Thus, for an input stream with N rows, the first decimation contains $N/4$ rows. The process can be repeated in multiple "levels", with each level having correspondingly fewer rows, until just one row remains, containing the average of all the data in the stream. In addition to the mean, decimated streams also store the minimum and maximum values of each successive block. For repeated decimations, these are calculated over the previously computed extrema.

When a requesting data for a plot, NILM Manager queries NilmDB for the total number of data rows in the desired interval. Based on the response, it automatically determines and requests data from the decimation level that contains the optimal number of points for display. The means are plotted as a line, and the minima and maxima are used to plot signal envelopes in a lighter shade. This helps maintain a visual indication of the data range of the original stream, similar to the display of a digital oscilloscope. The averaging operation also provides a simple low-pass filter, removing aliasing effects from the plot.

The additional storage requirements for the decimated streams are modest. Consider a stream with N rows and one column per row that is decimated by a factor of 4, as shown in Fig. 9. Decimated streams store three times as many columns (minimum, mean, and maximum) as the original stream, but each decimation level contains one-fourth as many rows. The total number of stored values for the original data plus L decimation levels is given by the geometric series:

$$N_{\text{total}} = N + 3N \cdot \sum_{k=1}^{L} \left(\frac{1}{4}\right)^k \tag{5}$$

$$= 3N \cdot \sum_{k=0}^{L} \left(\frac{1}{4}\right)^k - 2N \tag{6}$$

Taking the limit of this as $L \to \infty$ gives:

$$\lim_{L\to\infty} N_{\text{total}} = 3N \cdot \left(\frac{1}{1-(1/4)}\right) - 2N \tag{7}$$

$$= 2N \tag{8}$$

Therefore, storing *every* decimation level of a stream in NilmDB will at most only double the storage requirements of the original data, when decimating by a factor of 4. This overhead is low enough that it is almost always outweighed by the resulting bandwidth reduction and visual quality of the plots.

VIII. Disaggregation: Trainola and Diagnostics

One of the defining aspects of non-intrusive load monitoring is sensor reduction through the acquisition of aggregate power measurements from a collection of loads. Individual load identification and diagnostics requires subsequent disaggregation of these loads. Generally, this relies on the existence of some unique metric or feature of individual systems that distinguishes loads of interest. Typical metrics include steady-state power level and transient event shape, amplitude, and sequencing. In particular, event identification based on exemplar matching has been demonstrated as a particularly useful technique for identification and diagnostic monitoring [4], [14]–[16]. To support this, NilmDB and NILM Manager provide the "Trainola" transient event identification tool.

A. User Interface

Trainola is exposed by NILM Manager as an interactive workspace, shown in Fig. 10. The lower half of the window mirrors the data visualization interface, where the user can

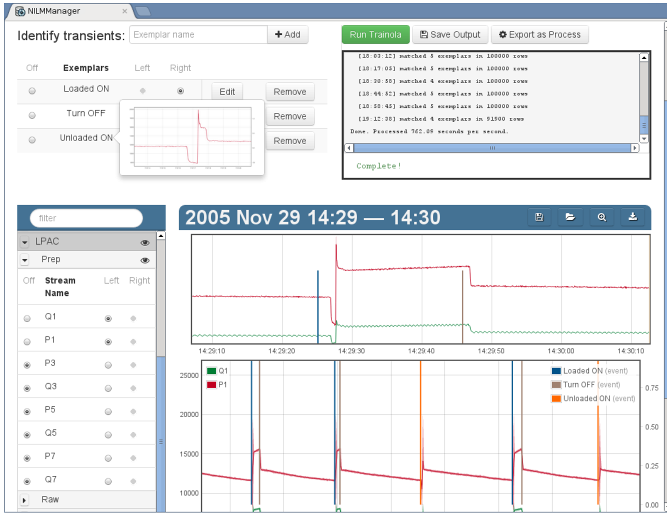

Fig. 10: "Trainola" exemplar-based event identification. The user graphically identifies examples of transient events in an input stream, which can then be automatically located and marked in an output stream. Matching events are plotted as vertical lines, overlaid on the input data.

freely select, plot, and navigate NilmDB streams. At the top, the user can create and name exemplars based on the data currently in view. Typically, these exemplars correspond to particular "turn-on" and "turn-off" events, and extend to include a few seconds of steady-state behavior before and after each transient event. For streams with multiple columns, such as spectral envelope preprocessor harmonics, the exemplars consist of whichever columns are visible when the exemplar is saved.

To run the automatic identification process, the user visually navigates to the target data stream and zooms out to the time-frame over which events should be identified. The exemplars and target data do not have to come from the same stream, but the same named columns must be present in both. Then, the "Run Trainola" button starts an identification process on the remote machine, which will classify events. During and after the identification, matched events can be viewed as vertical lines overlaid on the plot, by selectively enabling each exemplar. The events are also stored in a dedicated NilmDB output stream, and can be accessed by other filters and tools.

B. Matching Algorithm

The Trainola tool matches the shape of exemplars to the input data using the following algorithm. Consider two sampled waveforms of equal size N, for example, an observation $f[n]$ and an exemplar $g[n]$. After removing dc offset, a measure of similarity between two waveforms is the Euclidean distance, defined as:

$$D = \sum_{n \in N} \left(f[n] - g[n]\right)^2 \tag{9}$$

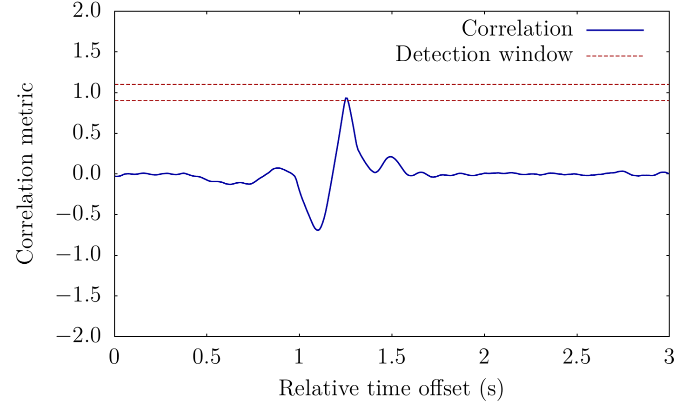

Fig. 11: Correlation metric for a matching exemplar. A peak that falls within a detection window around 1.0 indicates that the exemplar matches at that time.

This expression can be expanded to:

$$\begin{aligned} D &= \sum \left((f[n])^2 - 2f[n]g[n] + (g[n])^2\right) && (10) \\ &= \sum f[n]^2 + \sum g[n]^2 - 2\sum f[n]g[n] && (11) \end{aligned}$$

which is more conveniently expressed in terms of the dot product:

$$\begin{aligned} D &= (f \cdot f) + (g \cdot g) - 2(f \cdot g) && (12) \\ &= |f|^2 + |g|^2 - 2(f \cdot g) && (13) \end{aligned}$$

If the waveforms match, the Euclidean distance between them would be $D = 0$, and so the equation reduces to:

$$\begin{aligned} 0 &= |f|^2 + |g|^2 - 2(f \cdot g) && (14) \\ f \cdot g &= \frac{|f|^2 + |g|^2}{2} && (15) \end{aligned}$$

Furthermore, if the amplitudes match, $|f| = |g|$, giving:

$$\begin{aligned} f \cdot g &= \frac{2|g|^2}{2} && (16) \\ \frac{f \cdot g}{|g|^2} &= 1 && (17) \end{aligned}$$

Thus, (17) holds when the two waveforms match in amplitude and shape. Non-matching waveforms may also satisfy this condition in degenerate cases, but in general, $M = (f \cdot g)/|g|^2$ has been found to be a useful figure of merit to use when judging power signature similarity [16]. As M approaches 1.0, it indicates confidence that f and g match, both in shape and amplitude.

When the full waveform f contains more points than the exemplar g, the calculation can be performed over sliding windows of the input data, determining $M(t)$ at each window offset. As g "slides" over a feature in the f that matches in shape and amplitude, $M(t)$ will approach a local maximum of 1.0. Fig. 11 demonstrates this metric for a matching exemplar. In Trainola, a peak-finding algorithm is applied to locate local maxima, and values that fall within a small detection window around 1.0 are marked as matched events.

C. Diagnostics

Once individual transient events have been identified by Trainola, processing algorithms can be used to extract actionable system status information, such as health metrics, failure indicators, and motor speed estimations. Many such diagnostics have been developed within the context of the NILM [6], [15], [17]–[19], and can be applied within the NilmDB framework.

IX. Conclusions

The NilmDB data storage and management framework represents a shift in the design and implementation of load monitoring systems. It provides a fully structured, consistent, network-aware architecture that enables the development of actionable diagnostics across a wide variety of systems. It provides these services while minimizing demand for network communication reesources. NilmDB organizes and standardizes the collection and processing steps, enabling modular and reusable filter components to streamline and simplify the deployment of monitoring systems. With NILM Manager, NilmDB provides the solution to the "big data" analytics problem of large-scale power system monitoring. It enables modern advanced NILM techniques, which can disaggregate and report the operating schedule of individual loads strictly from measurements of aggregate current consumption, while maintaining low network bandwidth requirements and flexible computing options.

Acknowledgments

This research was funded by the BP-MIT Research Alliance, the ONR Structural Acoustics Program, the MIT Energy Initiative, and The Grainger Foundation.

References

[1] D. J. Leeds, "The Soft Grid 2013-2020: Big Data & Utility Analytics For Smart Grid," http://www.greentechmedia.com/research/report/the-soft-grid-2013, GTM Research, Tech. Rep., Dec 2012.

[2] Google, Inc., "An update on Google Health and Google PowerMeter," Available http://googleblog.blogspot.com/2011/06/update-on-google-health-and-google.html, accessed 2013-07-26.

[3] S. B. Leeb, S. R. Shaw, and J. J. L. Kirtley, "Transient event detection in spectral envelope estimates for nonintrusive load monitoring," *IEEE Transactions on Power Delivery*, vol. 10, no. 3, pp. 1200–1210, July 1995.

[4] L. K. Norford and S. B. Leeb, "Non-intrusive electrical load monitoring in commercial buildings based on steady state and transient load-detection algorithms," *Energy and Buildings*, vol. 24, pp. 51–64, 1996.

[5] J. Paris, "A framework for non-intrusive load monitoring and diagnostics," Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, February 2006.

[6] R. W. Cox, "Minimally intrusive strategies for fault detection and energy monitoring," PhD, MIT, Department of Electrical Engineering and Computer Science, September 2006.

[7] J. Paris, "A comprehensive system for non-intrusive load monitoring and diagnostics," Ph.D. dissertation, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, September 2013.

[8] Wikipedia Foundation, "Raspberry pi," Available http://en.wikipedia.org/wiki/Raspberry_Pi, accessed 2013-08-30.

[9] Python Software Foundation, "Pickle - python object serialization," Available http://docs.python.org/2/library/pickle.html, accessed 2013-08-30.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. The MIT Press, 2001.

[11] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1," IETF, RFC2616, Jun 1999.

[12] J. Donnal, "Home NILM: A Comprehensive Non-Intrusive Load Monitoring Toolkit," Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, June 2013.

[13] L. Williams, "Pyramidal parametrics," in *Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '83. New York, NY, USA: ACM, 1983, pp. 1–11. [Online]. Available: http://doi.acm.org/10.1145/800059.801126

[14] S. R. Shaw, S. B. Leeb, L. K. Norford, and R. W. Cox, "Nonintrusive load monitoring and diagnostics in power systems," *IEEE Transactions on Instrumentation and Measurement*, vol. 57, no. 7, pp. 1445–1454, July 2008.

[15] E. Proper, R. W. Cox, S. B. Leeb, K. Douglas, J. Paris, W. Wichakool, L. Foulks, R. Jones, P. Branch, A. Fuller, J. Leghorn, and G. Elkins, "Field demonstration of a real-time non-intrusive monitoring system for condition-based maintenance," in *Electric Ship Design Symposium*, National Harbor, Maryland, February 2009.

[16] J. Paris, Z. Remscrim, K. Douglas, S. B. Leeb, R. W. Cox, S. T. Gavin, S. G. Coe, J. R. Haag, and A. Goshorn, "Scalability of non-intrusive load monitoring for shipboard applications," in *American Society of Naval Engineers Day 2009*, National Harbor, Maryland, April 2009.

[17] P. R. Armstrong, "Model identification with application to building control and fault detection," PhD, MIT, Department of Architecture, September 2004.

[18] W. Greene, J. S. Ramsey, S. B. Leeb, T. DeNucci, J. Paris, M. Obar, R. Cox, C. Laughman, and T. J. McCoy, "Non-intrusive monitoring for condition-based maintenance," in *American Society of Naval Engineers Reconfigurability and Survivability Symposium*, Atlantic Beach, Florida, February 2005.

[19] U. Orji, Z. Remscrim, C. Laughman, S. B. Leeb, W. Wichakool, C. Shantz, R. Cox, J. Paris, J. Kirtley, and L. Norford, "Fault detection and diagnostics for non-intrusive monitoring using motor harmonics," in *Applied Power Electronics Conference*, Palm Springs, CA, February 2010.