DISTIL: Design and Implemention of a Scalable Synchrophasor Data Processing System

Michael P Andersen*, Sam Kumar†,Connor Brooks‡, Alexandra von Meier§ and David E. Culler¶
Department of Electrical Engineering and Computer Science
University of California at Berkeley
*m.andersen@berkeley.edu, †samkumar@berkeley.edu, ‡cbrooks@berkeley.edu, §vonmeier@berkeley.edu, ¶culler@berkeley.edu

***Abstract*—The introduction and deployment of cheap, high precision, high sample rate next-generation synchrophasors en masse in both the transmission and distribution tier – while invaluable for fault diagnosis, situational awareness and capacity planning – poses a problem for existing methods of phasor data analysis and storage. Addressing this, we present the design and implementation of a novel architecture for synchrophasor data analysis on distributed commodity hardware. At the core is a new feature-rich timeseries store, BTrDB. Capable of sustained writes and reads in excess of 16 million points per second per cluster node, advanced query functionality and highly efficient storage, this database enables novel analysis and visualization techniques. Leveraging this, a *distillate* framework has been developed that enables agile development of scalable analysis pipelines with strict guarantees on result integrity despite asynchronous changes in data or out of order arrival. Finally, the system is evaluated in a pilot deployment, archiving more than 216 billion raw datapoints and 515 billion derived datapoints from 13 devices in just 3.9TB. We show that the system is capable of scaling to handle complex analytics and storage for tens of thousands of next-generation synchrophasors on off-the-shelf servers.**

I. Introduction

Synchrophasors have been deployed in the hundreds throughout the transmission tier to improve power system reliability and visibility[1]. The data they provide is potentially valuable for a wide variety of applications, both in real-time operation, including wide-area situational awareness, monitoring frequency stability, power oscillation, and voltage, alarming, resource integration, and state estimation, and for off-line planning, including baselining, event analysis, model validation, load characterization, and response. Since phasor measurement units (PMU) produce high data rates from distributed locations across the grid, considerable attention has been devoted to the basic data transport, typically multiple PMUs are aggregated at a phasor data concentrator (PDC) which buffers data, provides basic integrity checking, correlates the streams by time tag, formats the data (e.g., IEEE C37.118, and delivers it over, typically, higher bandwidth links to additional concentrators or back-end data processing resources[2]. The backend represents a considerable challenge due to the immense footprint of the collective data streams. Various data historians are used to warehouse the phasor data and various relational and non-relational databases are use to hold small subsets and data processing results, typically as a part of a vendor proprietary solution. Very little published work attends to the storage and execution infrastructure required to implement the potential PMU applications.

The recent introduction of μPMUs [3], small inexpensive, synchronized phasor measurement units, holds the potential to vastly expand the number of streams (to tens of thousands deployed throughout the distribution tier) and the diversity of applications. In order to explore and understand the potential applications of this new technology and to develop robust analysis algorithms, researchers and practitioners require an extremely flexible framework for developing, deploying, and utilizing stream processing algorithms at scale with little effort. Such a framework must deal with the evolution of processing techniques, as well as expansion and modification of the physical infrastructure of devices, deployments, and so on.

This paper describes an agile, scalable stream processing infrastructure for large networks of μPMUs . Numerous μPMU timeseries are delivered into a novel multi-resolution, versioned time-series data store. Algorithmic transformations on streams are described by *distillers* that fire computation as chunks of data change on input streams and push computed results onto output streams. Synchrophasor algorithms tap into a network of distillers, which typically start with basic cleaners on raw data and feed through dataflow operators that may combine time-correlated streams within a PMU or across PMUs. The logic of each distiller is described by a simple kernel – a few lines of code that directly reflects the mathematics. The data processing framework deals with all of the performance optimizations and bookkeeping associated with multiple interleaved streams arriving at different rates, possibly out of order, chunking, buffering, scheduling and so on. Moreover, everything is versioned - the data, the distillers, and the intermediate streams. As a change occurs, the framework determines what needs to be recomputed to produce consistent results with precise provenance and schedules the processing required to propagate the change through associated streams. Our results show that easy-to-write kernels in Python can process many μPMU streams at full line rate and the architecture easily scales across cluster resources to achieve extremely large scale. As distillers move toward production they can be optimized to operate closer to the storage engine rate of tens of millions of inserts and removals per second per node.

II. Use Case

The systems presented in this paper were developed to cope with the increased demands that next generation electric grid measurement data will place on storage and analysis infrastructure. We encountered these challenges in deploying an array of uPMUs that send GPS time-stamped measurements of voltage and current magnitudes and angles at 120 samples

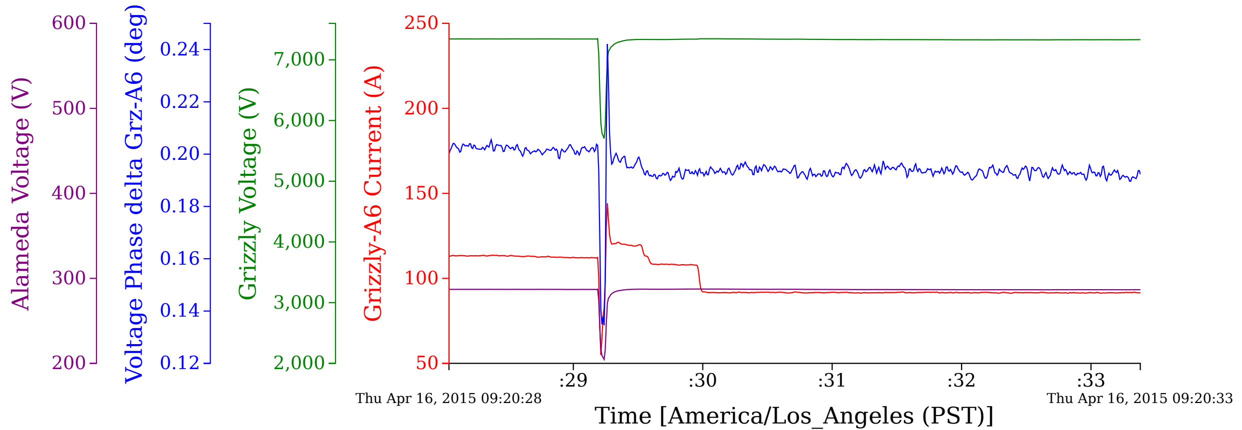

Fig. 1: Sample event as seen on the plotter (shown for one phase only): a voltage sag originating from the transmission system results in a current transient and the subsequent loss of some load. The voltage phase angle difference between locations on the same primary distribution feeder shows the disturbance and typical variations too small to observe with transmission-level PMUs.

per second [3] on each of the three AC phases (which in distribution systems, unlike transmission, can be highly unsymmetrical). Distillers calculate several refined streams per raw stream that represent quantities with relevant and recognizable interpretations in the grid context, such as phase angle differences, AC frequency, power factor, or symmetrical (positive/negative/zero-sequence) components.

While this data volume could be reduced at the source through "report by exception" rules, or within the analytic infrastructure by running distillers only on small subsets of data, there are good reasons to consider the streams in their entirety and to leverage data sets for multiple applications simultaneously. These applications include not only the analysis of specific events, but observation of the physical system at any given moment in time, as well as its evolution over a range of time scales, from AC cycles to seasons. A particular emerging challenge as distribution systems increasingly host diverse and active components is to combine steady-state, transient and dynamic analysis within the same tools to support operations and planning [4]. Rather than hamper the future development of applications by a design based on isolated use case assumptions, the infrastructure presented here requires no *a priori* censorship or compartmentalization and thus affords unique opportunities for learning. For example, diagnosis of a voltage sag as a transmission-level or locally caused event can be corroborated by the magnitude of a corresponding sag at different locations on the grid, since measurements for precisely that instant can be readily compared (Figure 1). Loss and recovery of various loads following a disturbance can be characterized in detail because data and distillates are available simultaneously at sub-cycle temporal resolution and over the course of minutes and hours around the event.

The primary purpose of DISTIL is to serve as an analytic tool to enable humans to visualize never before observable quantities, and develop applications in the research setting. It bears emphasis that measurements, such as those from our uPMU array, represent a vast expansion of visibility into power distribution systems, which are highly diverse, idiosyncratic, time-varying and thus data-rich, yet still largely opaque to operators.

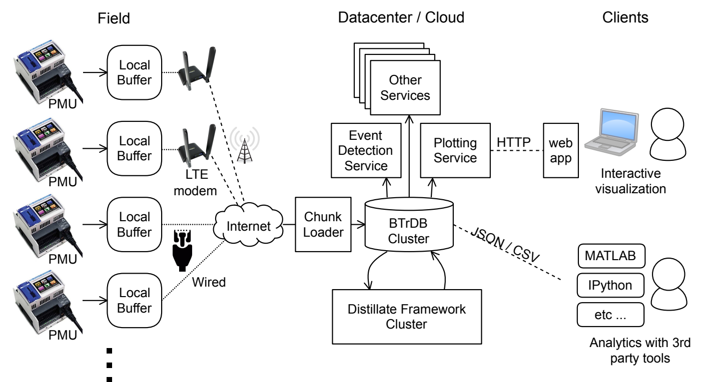

Fig. 2: Whole system architecture overview

III. Architecture

The system (Figure 2) consists of multiple hardware and software components distributed across cellular, wireless and wired links. In the field μPMUs are deployed at critical points in the power distribution infrastructure. The μPMU is a well connected embedded device, with an Ethernet connection and an ARM processor running Linux. In many locations, wired Internet is not available, so an off-the-shelf LTE modem is used to obtain Internet access. To cope with intermittent connectivity, an agent on the μPMU collects the readings in 2 minute (172,800 point) chunks and reliably transmits them directly across the Internet to the processing infrastructure, either located in the cloud or in a utility datacenter. It is worth noting that no dedicated communication infrastructure or data concentrators are required, i.e the PDC is essentially collapsed into the μPMU .

In the datacenter, there are three subsystems. The storage subsystem, provided by BTrDB, is responsible for archiving the raw data, and the products of the distillate framework. It acts as a crosspoint that connects every other service, analysis algorithm or external third party toolchain. The distillate

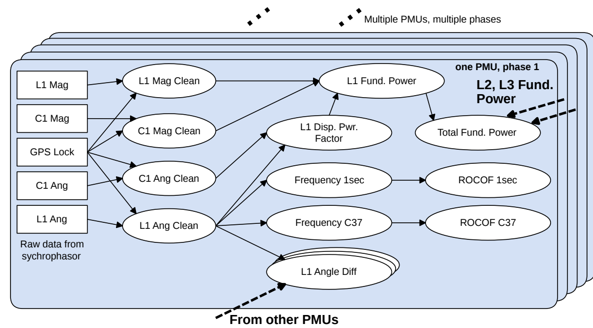

Fig. 3: A subsection of the data flow graph

framework subsystem, discussed in Section IV, provides high throughput distributed computation with eventual consistency in the face of data changes or software/hardware failures. Finally, the service subsystem provides utilities, such as interactive visualization and event detection. Additional protocol translation to enable client modalities not supported by BTrDB natively are implemented as services.

Clients, such as researchers, system operators, or automated tools, can consume data either via an interactive visualization tool, or in machine-readable formats that are compatible with third party analysis and simulation tools.

A. *Data flow*

μPMUs stream raw data directly into the database via the chunk loader. To facilitate rapid human-centric analysis, this data is then automatically *distilled* into globally time-aligned, clean streams by utilising the GPS lock stream and constantly-evolving heuristics for good data. These streams are in turn the inputs for an ensemble of further algorithms or *distillers* that form a directed data flow graph, a portion of which can be seen in Figure 3, for a single phase of a single μPMU. These are duplicated for the additional phases, and again for each addition μPMU. Distillers such as fundamental/reactive power combine distillates across phases, and there are several algorithms that require data from multiple different μPMUs. Of these, only the phase angle difference is shown.

Our architecture focuses on efficient and reliable calculation and storage of these *distillates* in advance of queries, rather than just-in-time materialization. The advantage is that many months or years of data can be queried in milliseconds, whereas with this density of data, just-in-time materialisation would take hours or days. In addition, emphasis is placed on guaranteeing correctness of the resulting analysis. This is because most interesting data lies in transients which are hard to distinguish from faults in analysis. This guarantee is established by treating the graph as a true dataflow graph that specifies a relationship between streams, rather than simply as a pipeline that processes incoming data. The difference is subtle and best illustrated by example. In a traditional streaming pipeline, if a researcher were to identify a range of data that needs to be considered invalid after it had already been processed, they would need to manually invalidate the corresponding sections of results that were produced using that data. If there were derivative streams that the researcher were unaware of, they might slip through the cracks and later lead to invalid results and reasoning. In DISTIL, the researcher need only change the single stream, and the results will propagate through the graph of dependent streams automatically.

B. *Database*

At the center of the system is a new timeseries store – Berkeley Tree Database (BTrDB) – that has been developed for high density scalar timeseries data, typical of synchrophasors. Initial attempts to use off-the-shelf technology (e.g. OpenTSDB, SQL databases, Cassandra) met with insurmountable challenges: lack of timestamp precision (32 bits / millisecond precision being the norm), slow queries for fast-path operations such as "find what has changed since this stream was last checked", poor storage efficiency and inability to preserve reproducibility despite deletes. Over and above these, the single biggest problem was extremely poor throughput. While existing solutions work well for devices sampling at 1 Hz or slower, they cannot cope with the (120 x pipeline depth) Hz rate we required. Similar studies by other researchers in the field concur with our findings [5].

These shortcomings necessitated the development of a new timeseries database for storing synchrophasor readings, capable of advanced query functionality to support the data processing methodology presented in Section IV. This methodology relies on the database as a critical component to decouple stages in the processing pipeline allowing for failure tolerant distributed computation and reuse of intermediate products; sustained throughput is essential. The resultant database posseses a novel architecture that offers extremely high performance, multi-resolution fixed-response-time queries, persistent per-commit multi-versioning with difference computation, and scalability to petabyte scale datasets.

1) *Performance*

BTrDB offers very high throughput. A *single* computation node deployed on the cloud can handle sustained writes at more than 16.7 million points per second – more than ten thousand three-phase microsynchrophasor units. Read throughput is similar at more than 19.8 million points per second, enough to support complex on-line computation infrastructure (covered in Section IV) or a heavy query load. These figures come from a deployment on EC2[1] with distributed storage, showing that deployment and scalability on commodity cloud platforms is possible.

To achieve these results, – a per-node throughput that is two to three orders of magnitude faster than OpenTSDB [5], Cassandra, Couchbase, HBase or MongoDB[2]– the IO pattern has been very carefully engineered. In addition, it is suitable for the spinning metal drives used in data warehousing, especially on the write path that is the primary concern in phasor data concentrator and archival deployments.

In addition, the database scales horizontally to large clusters on two tiers. Figure 4 shows the tiered architecture of a BTrDB cluster. At the top, the IO tier is made up computationally powerful machines as, due to the compression techniques used, both reads and writes are CPU intensive. The design of the database allows for concurrent reads to be distributed amongst multiple machines, whereas writes need

[1]The compute node is a c3.8xlarge, the storage nodes are i2.4xlarges

[2]a very recent third party comprehensive benchmark of these offerings can be found in [6]

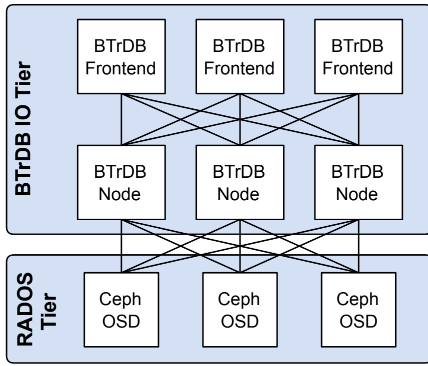

Fig. 4: BTrDB tiered architecture

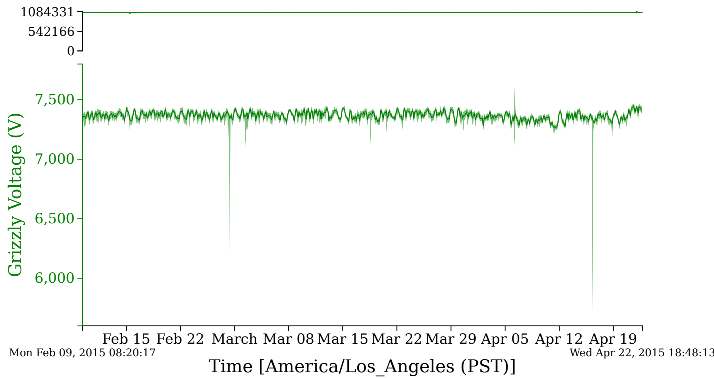

Fig. 5: Two months of voltage data plotted with 2.4 hours (1084331 points) per pixel column. Several 100ms voltage sags (including one shown in Figure 1) are clearly visible

to be redirected by a frontend daemon to the server currently holding the write lock. This daemon generally runs on the same machine as the main BTrDB node. To scale throughput, additional machines with high CPU and memory are added to this cluster.

Storage is provided by the second tier: Ceph RADOS [7] a scalable, reliable storage service for petabyte-scale storage clusters. This proven technology allows for seamless replication and migration of data across clusters of thousands of machines. In addition, Ceph is engineered to allow for *hot storage* on SSDs that accelerate IO operations on the working set. It handles the migration to and from *cold storage*, composed of cheaper and larger spinning metal drives that store historic data. To scale storage, additional RADOS Object Store servers with large numbers of attached hard drives are added to the cluster.

2) Multi-resolution store

The database stores the data in a time partitioning copy-on-write tree. The internal nodes are used to store statistical representations of the data in their children. This data structure guarantees that the statistical representations are up-to-date at the end of every commit, as the internal nodes are persisted before the root superblock is written to disk. The statistics stored by BTrDB internal nodes are the minimum, mean, maximum and count of the values in the subtree. This allows these queries, or any queries containing these as a subexpression, to be accelerated by several orders of magnitude.

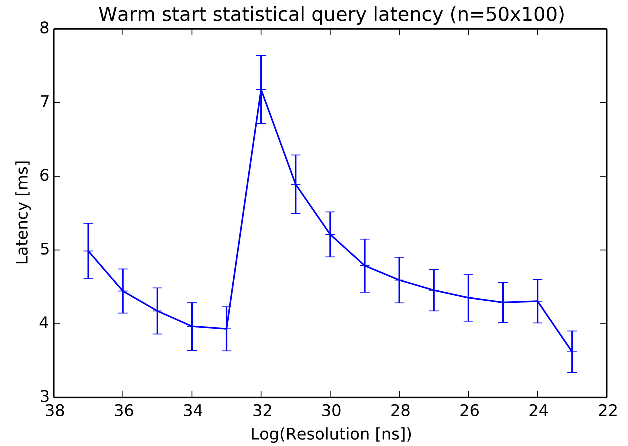

Fig. 6: Response times for statistical queries backing a 1024 pixel wide interactive window as it is zoomed in from 2^{37}ns per pixel to 2^{23}ns per pixel (2.3 minutes to 8.4ms per pixel)

The statistical queries can be used for efficiently locating transients such as threshold excursions without needing to query the full resolution data. This is utilised by the event detection service to offer lightweight tagging of “interesting” time windows for human observation. As an example of this, consider Figure 5 that shows 2 months of voltage data with sufficient information that an operator can identify several voltage sags that warrant further investigation even though they are only 100ms long. This data was fetched in < 10ms.

The agility with which such large quantities of data can be explored lends itself to interactive visualization where an operator can zoom and pan through streams without waiting for queries to complete. The statistical results show more information than would be available from traditional naive or subsampling plotting methods. In Figure 5, each pixel column in the plot covers a time range of roughly 2.4 hours. Although only a single record is plotted per column, the min/mean/max values are used to plot a bracket (in a light color) with the mean as a darker line, increasing the information presented in the graph over a naive plot of the full raw information which would not identify the mean. A subsampled plot does not identify the mean and elides valuable information about transients. The count value is used to plot a *data density* graph above the main plot, useful for identifying missing data and evaluating how many records each pixel represent. Here it shows that each pixel column represents roughly one million points, totalling 747 million points across the width of the plot.

Prior work has considered just-in-time aggregation of data for pixel-perfect visualisation purposes [8], but in our work the query is satisfied without reading the underlying data, enabling a fast response time irrespective of the time range. Figure 6 shows the response time for queries zooming in from a 40 hour wide window to an 8.5 second wide window. The times shown are full-stack round trip times for JSON/HTTP queries, the format used by the plotting web application. It was measured by querying 50 different streams, 100 times each to obtain the variance in the latency. These low-latency responses allow for smooth interactive zooming and panning through terabytes of data, facilitating human interpretation – a utility engineer described his experience of viewing μPMU data on the plotter as “a kid in a candy shop.”

```
def compute(self, changes, inputs,
      params, report):
    output = report.output("phase_difference")
    frame = inputs["phase1"] - inputs["phase2"]
    frame.dropna()
    frame += 180
    frame %= 360
    frame -= 180
    output.insertframe(frame)
    # This specifies the range to be deleted
    output.addbounds(*changed_ranges["phase1"])
```

Fig. 8: The Python kernel for phase difference

3) Versioned store

As the primary data structure is copy-on-write, the database is *monotonic*. New additions to a stream of timeseries data result in new *generations*. Even a deletion does not result in lost data, it is merely unavailable to queries targeting a generation newer than that containing the deletion operation.

This structure allows queries, to be guaranteed reproducible. Combined with software version control systems such as GitHub, this yields perfect provenance and reproducibility of analysis. An algorithm simply needs to retain the generation numbers used to satisfy the queries, and the commit hash of the algorithm, and it can redo the analysis and get the exact results again later.

More importantly, generational storage allows the database to efficiently answer *difference* queries that yield a *changeset* between two generation numbers. A changeset contains a list of **start, end** tuples denoting ranges of time that have been changed. This novel capability enables the efficient composition of stages in a processing pipeline that can react to out of order changes in data without maintaining per-consumer journals. Table I shows how changeset queries in large (9 month) datasets complete in milliseconds[3].

IV. Distillate Framework

The distillate framework provides an environment for materializing analytics streams given an abstract description of a dataflow graph, along with the algorithm kernels that describe each node in the graph. Figure 7 outlines the primary components of the framework.

A. Distiller kernel

The process begins with a researcher developing a kernel. An example kernel that computes phase delta can be found in Figure 8. A kernel is a *stateless, idempotent* block of code that consists of two functions: precompute and compute. The precompute takes a list of time ranges containing changes on the algorithm inputs and returns a list of time ranges that will be required to compute the algorithm's outputs. For many distillates this operation is the identity function, but others require an additional window of data before the change in order to compute the products. An example is the frequency distillate which requires an extra second of data so that the angle deltas can be computed across the whole window.

[3]we would have liked to include the query times for this operation on a conventional relational or time series database but could not construct an implementation that completes in under linear time without sacrificing large amounts of storage

The second part of the kernel is the compute function which is invoked with all the data that has been fetched from the database. It outputs a list of time ranges that need to be erased, and a set of datapoints that need to be inserted.

A key characteristic of the kernel is that it does not maintain any state outside the inputs it receives and is idempotent. This allows multiple chunks of computation to run in parallel, be distributed across machines in a cluster and re-run without unwanted side-effects. It also ensures that the distiller can handle out of order arrival or holes in the data without any additional code in the kernel.

B. Graph description

The researcher *instantiates* distillers by creating INI files and uploading them to a specific repository on GitHub (shown top left in Figure 7). This file specifies a list of nodes in the dataflow graph, with each node having an algorithm, a set of parameters, input streams and output streams. The act of pushing the configuration file to the repository triggers an automatic inclusion into the schedule and subsequent computation. This design choice allows for accountability and reproducability as there is no out of band control in either the algorithm or the graph description: everything goes via a versioned source control system. The metadata associated with the output products contains the version numbers of both the algorithm and the parameters, allowing the provenance of any point of data to be determined.

C. Scheduler

The scheduler runs on the front end for the three clusters that comprise the system. It receives the graph description via a post-commit hook from GitHub and spawns a process for every node in the graph, passing it the instantiation parameters. These processes are distributed among the cores in the compute cluster. By utilising the versioned query functionality provided by BTrDB, consistent views of the inputs to each node can be maintained, even if other nodes are writing to the same streams in the same time range. For this reason, the node processes are fully parallelizable, requiring no inter-process communication or synchronisation at all. This is the key characteristic that offers scalability for the compute cluster.

D. Node process

The node process begins by pulling the algorithm code from GitHub. All execution trace information is directed to logging infrastructure so that development errors or warnings thrown by the kernel can be observed on the pipeline monitoring dashboard. From the node description, the input streams are determined, and the metadata for those streams is loaded. The BTrDB database is then queried for all time ranges that have changed since the last timeslice that was scheduled for the node, forming a changeset. These changes need not be contiguous, at the end of the stream or in previously empty space – the system is fully generic and will handle arbitrary changes.

The changeset is then *chunked* into pieces that will fit into memory and be efficient working sets for computation. Each chunk is then processed independently. Like the node process, this may also be parallelized, although we have found the parallelism at the scheduler level sufficient to take advantage of

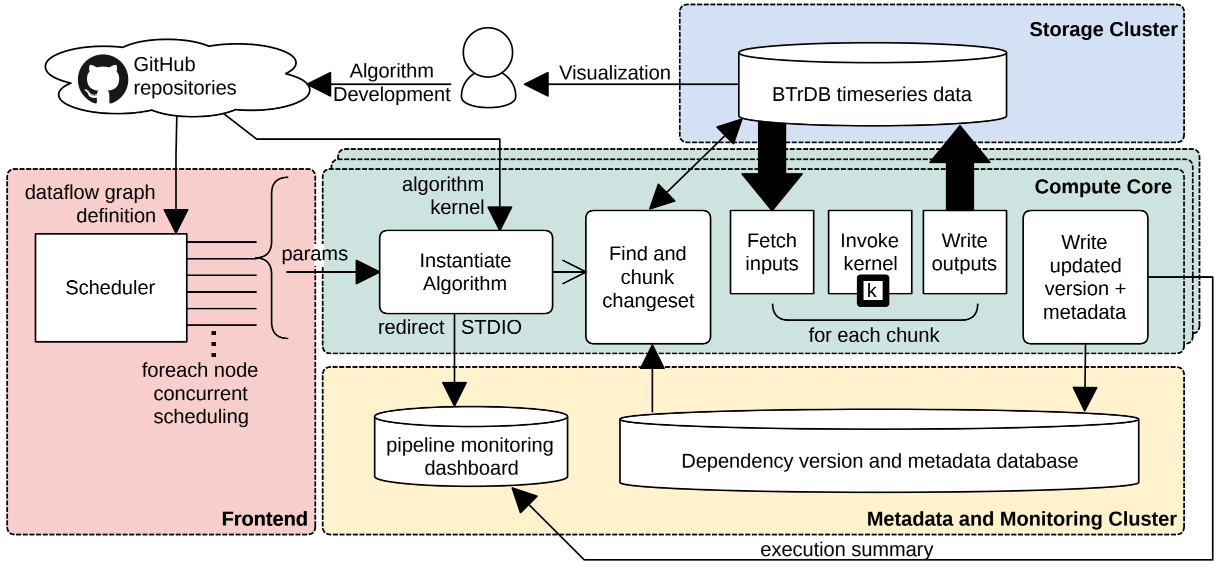

Fig. 7: The distillate framework, showing the three clusters, frontend node and cloud services along with the flow of data

the computational resources we have deployed. For simplicity, the current implementation processes the chunks sequentially.

For each chunk – a timerange containing changes – the framework invokes the prerequisites section of the kernel. It then queries and builds the dataset. The kernel compute function is invoked to process the dataset, returning the time ranges to be erased, and the data points for the products. These results are then inserted into the database.

Finally, once all the chunks have been computed without error, the input versions in the metadata database are updated to the ones used for the timeslice of computation that just completed. If a software or hardware error occurs after some but not all chunks have been written, these version numbers will not be updated, causing the scheduler to reschedule all the chunks on a different compute core, achieving eventual consistency.

V. Analysis

A. *Database performance*

Although a full evaluation of the performance of BTrDB is out of the scope of this paper, a summary evaluation of the space efficiency and throughput of the database is required to contextualise the distillate framework performance.

The first figure of merit is the read and write performance. On an Intel Xeon E5-2680v2 based cloud server with 60GB of RAM, described earlier, a single BTrDB node writes 16.7 million scalar points per second and reads 19.8 million scalar points per second under an IO pattern that mirrors that of the chunk loader and distillers. This corresponds to a real-time load of 11500 three phase uPMUs per BTrDB node in the cluster and is several orders of magnitude faster than the time-series databases or relational databases that have so far been used for phasor data storage [5].

The second figure of merit is of storage efficiency. The challenge is to balance the requirements of a multi-resolution store – data carrying internal nodes in a space partitioning copy-on-write tree – with that of warehouse scale archival which requires optimal storage. At present, we have a pilot deployment of thirteen μPMUs and have archived 216,748,666,750 raw data points (not including those produced by the analysis pipeline). This dataset is only 1.1TB as the novel lossless compression algorithm developed for BTrDB uses only **5.46 bytes** per (16 byte) reading on average, giving a compression ratio of **2.93x**. This is significantly better than existing phasor data compression techniques [9].

As our cleaning algorithm heuristics are still under development, the size of our stored distillates is much larger (the results of the whole graph are archived for every version of the cleaning algorithm). These bring the total size of our stored dataset to 3.9 TB. The number of stored points available in the latest versions of the distillates excluding archived deleted regions, is 212,007,633,453 (there are approximately 300 billion archived changed points not in the latest versions). The bytes per reading figure is calculated by taking the total size on disk, including all indexes and statistical copies that serve the multiresolution queries, and dividing it by the total number of readings.

B. *Framework node performance*

To provide a characterisation of the performance of the framework, we performed a microbenchmark of three distillates on two deployments of DISTIL. The first deployment is on a small production setup, where the storage, compute and metadata are all distributed. The CPU is an Intel E5-2670 v2 @ 2.50GHz. The second deployment is on our staging machine, an inexpensive home media server running an Intel i7 4600U @ 2.1GHz. Here, all the components of the system are running on the same box, with the storage attached via USB 3.0. The production cluster has 48 cores per machine, and the staging machine has 8 cores, but in both cases, only a single core is tested – the scalability to multiple cores is completely linear. The most important factor to consider in the performance of the distillate framework is the ratio of time spent processing a changeset to the wall time that the changeset represents, i.e. how much faster than realtime the system is. Primarily this ratio influences how many nodes in the dataflow graph there can be, and therefore how many synchrophasors can be supported by the system.

TABLE I: Compute times for processing a 38 hour changeset on different distillates and deployments

	Distributed			Single Machine		
	Identity	Phase Difference	Reactive/Fundamental Pwr	Identity	Phase Difference	Reactive/Fundamental Pwr
Input/Output streams	1/1	2/1	4/2	1/1	2/1	4/2
Compute changeset	972 μs	1659μs	1180μs	649μs	468μs	47183μs
Query data [s]	69.8	104.4	196.9	38.9	67.3	145.5
Kernel calculation [s]	10.8	22.7	245.5	7.9	15.9	164.3
Delete old data[s]	6.7	6.9	15.8	8.0	6.9	20.6
Insert new data[s]	40.7	39.8	66.5	31.91	30.7	52.8
Changeset / compute time	1064 x	773 x	259 x	1579 x	1135 x	357 x

Table I shows the wall time spent in the various stages of the framework for a selection of three distillers running on two deployments. The first distiller is an identity distiller which simply copies its input to its output, to characterise what fraction of the kernel computation time is from moving data. The second kernel is a phase delta which measures the difference in voltage phase angle between two different μPMUs , performing some validation (the kernel for this distillate is listed in Figure 8). Finally, as a more computationally intensive kernel, the RPFP distillate computes reactive and fundamental power. Both systems and sets of distillates were synthetically exposed to a 38 hour changeset, and the resulting computation was timed.

The first thing to note is that even on a single core, most distillates operate at well beyond 250x realtime. Secondly, note the alacrity of the changeset calculation query – an operation that requires either a costly linear scan or an untenably large index in existing databases. Finally, as expected, the query and insert times scale with the valency of the node. Algorithms with more inputs and outputs have increased data volume and take longer to compute. As an aside, we are not sure why there is such a disparity between the performance of cores on the two systems, it is likely that it is due to CPU power scaling, as even the kernel compute time differs, and this stage contains no disk or network IO, nor traffic to the database. As these results scale linearly with an increase in cores, an average multicore commodity server such as a dual socket Xeon E5-2670v3 (with 48 cores) can handle more than 54000 phase difference nodes and still be faster than real-time. This amounts to 18000 μPMU pairs – more than the number of synchrophasors installed in North America. A moderately sized cluster could handle in-depth analytics for extremely large scale deployments with ease. The results for the staging machine were included here to show that despite being engineered for warehouse scale analysis of tens of thousands of synchrophasors, the system scales down well to a single machine that could be deployed on-site.

VI. CONCLUSION

This paper presents DISTIL , a system for distributed analysis and storage of extremely large volumes of synchrophasor data. Data from μPMUs is transmitted using commodity communication infrastructure across the Internet, to a central processing location in a cloud or operator datacenter, removing the need for dedicated phasor data concentrator hardware. The processing infrastructure builds on BTrDB, a novel timeseries database offering read or write performance in excess of 16 million points per second per cluster node – more than two orders of magnitude faster than existing solutions. Persistent multiversioning enables data provenance determination and reproducibility of analysis. The distillate framework allows complex analysis pipelines to be developed by writing and connecting simple, concise kernels of analysis code representing nodes in a dataflow graph that are automatically scheduled into execution on a processing cluster. Issues of error recovery and change propagation into derivative streams are handled transparently. We have evaluated DISTIL in a pilot study, analyzing data from 13 deployed next-generation μPMUs and archiving more than 730 billion raw and derived datapoints. This dataset occupies only 3.9TB, due to novel compression techniques that yield a 2.93x compression ratio. Visualization of the data can be done interactively, where terabytes of data can be navigated smoothly and seamlessly due to the rapid multiresolution capabilities of BTrDB. We demonstrate that the framework is capable of scaling to complex analytics on tens of thousands of μPMUs using cost-effective commodity hardware, a previously untenable goal.

ACKNOWLEDGMENT

This research is sponsored in part by the U.S. Department of Energy ARPA-E program (DE-AR0000340), National Science Foundation CPS-1239552, and Fulbright Scholarship program.

REFERENCES

[1] "North american synchrophasor iniative," https://naspi.org, 5 2015.

[2] M. Patel, S. Aivaliotis, E. Ellen *et al.*, "Real-time application of synchrophasors for improving reliability," *NERC Report, Oct*, 2010.

[3] A. von Meier, D. Culler, A. McEachern, and R. Arghandeh, "Microsynchrophasors for distribution systems," in *IEEE 5th Innovative Smart Grid Technologies Conference, Washington, DC*, 2014.

[4] E. M. Stewart, S. Kiliccote, C. M. Shand, A. W. McMorran, R. Arghandeh, and A. von Meier, "Addressing the challenges for integrating microsynchrophasor data with operational system applications," 07/2014 2014.

[5] T. Goldschmidt, A. Jansen, H. Koziolek, J. Doppelhamer, and H. Breivold, "Scalability and robustness of time-series databases for cloud-native monitoring of industrial processes," in *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*, June 2014, pp. 602–609.

[6] EndPoint, "Benchmarking top nosql databases," Endpoint, Tech. Rep., apr 2015, http://www.datastax.com/wp-content/themes/datastax-2014-08/files/NoSQL_Benchmarks_EndPoint.pdf.

[7] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn, "Rados: a scalable, reliable storage service for petabyte-scale storage clusters," in *Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing'07*. ACM, 2007, pp. 35–44.

[8] U. Jugel, Z. Jerzak, G. Hackenbroich, and V. Markl, "M4: A visualization-oriented time series data aggregation," *VLDB. VLDB Endowment*, 2014.

[9] P. Top and J. Breneman, "Compressing phasor measurement data," in *North American Power Symposium (NAPS), 2013*, Sept 2013, pp. 1–4.