A Complete History of Everything

Kyle Lanclos,[a] William T.S. Deich[a]

[a]UCO/Lick Observatory, University of California, Santa Cruz, CA 95064 USA
http://www.ucolick.org

ABSTRACT

This paper discusses Lick Observatory's local solution for retaining a complete history of everything. Leveraging our existing deployment of a publish/subscribe communications model that is used to broadcast the state of all systems at Lick Observatory, a monitoring daemon runs on a dedicated server that subscribes to and records all published messages. Our success with this system is a testament to the power of simple, straightforward approaches to complex problems. The solution itself is written in Python, and the initial version required about a week of development time; the data are stored in PostgreSQL database tables using a distinctly simple schema.

Over time, we addressed scaling issues as the data set grew, which involved reworking the PostgreSQL database schema on the back-end. We also duplicate the data in flat files to enable recovery or migration of the data from one server to another. This paper will cover both the initial design as well as the solutions to the subsequent deployment issues, the trade-offs that motivated those choices, and the integration of this history database with existing client applications.

Keywords: telemetry, database, PostgreSQL, Python, historical record, software architecture, software design, Lick Observatory

1. INTRODUCTION

Determining the cause of subtle problems in complex systems is challenging at the best of times, even when the problem solver has a commanding expertise of the system in question. Successful troubleshooting is aided by, and sometimes requires, detailed instantaneous knowledge of a system's state. In order to provide that knowledge, we created a tool to establish an exhaustive history of all computer-monitored state at Lick Observatory.

Our immediate motivation was to increase the resources available to day-time staff to address problems that occurred during a night's observing. With a complete historical record at their disposal, the problem solver can carefully analyze the changing states that surrounded a specific event, and inspect different parameters without needing to recreate the problem to capture more data. This is particularly valuable for problems that are intermittent or otherwise difficult to recreate, but with monitors that never sleep, the essential data will be captured and available in perpetuity.

In addition to fully describing isolated events, a deep historical record is an essential tool for high-quality trend analysis. By monitoring everything and never throwing away old information, it is possible to capture and analyze trends without waiting to collect any data.

In either case, accurate timestamps are vital to ensure properly sequenced records. At Lick Observatory, all instrument hosts use a set of local Network Time Protocol (NTP) servers, ensuring that system clocks are consistent throughout the observatory and have excellent absolute precision, down to about twenty microseconds.

The effectiveness of the solution described in this paper depends directly on two ideals: everything must be monitored, and all monitors must publish their data via the same mechanism. Though we have not yet achieved either at Lick Observatory, we are closer to these ideals than otherwise; there are sixty-nine such software services at Lick Observatory, covering all but a handful of systems, allowing our complete history of everything an opportunity to prove its utility.

Send correspondence to K.L.: lanclos@ucolick.org; W.T.S.D.: will@ucolick.org

2. APPLICATION DESIGN

When establishing the basic framework of our history daemon (hereafter referred to as the historian), we steered our choices towards simplicity, reliability, and maintainability.[1] In particular, we sought a solution to a discrete problem, and in its implementation, minimal external dependencies and a low impact on our existing build and run-time environments. An earlier attempt to establish a similar tool (`ktlwatch`[2]) for use at both W.M Keck Observatory and Lick Observatory did not succeed in a production capacity, as it did not adhere to these goals.

The prerequisite of a common communications platform is largely a solved problem at Lick Observatory; nearly all of our systems use the Keck Task Library (KTL)[3,4] application programming interface (API) for communication between servers and clients.

Many of the remaining design considerations revolved around efficiency. For the historian, it needed to efficiently process broadcast events, lest it quickly get overwhelmed by a high volume of incoming telemetry. When browsing the captured history, queries for any time span must return results without a large initial delay; when working interactively with command-line tools, users reasonably expect immediate responses, regardless of when the data was initially captured. As a corollary to this last objective, the data must be stored efficiently, so that the simple act of storing new data does not negatively impact interactive extraction of old data.

In addition to efficiency, the next major design consideration was to ensure that the historian stored the *right* data, a subset of *all* data. There are two basic types of telemetry at Lick Observatory: continuously fluctuating numeric values, and state information. States should not change rapidly: if a shutter is open, it is open; when it makes a transition to being closed, it should make that transition once. Continuous numeric values, on the other hand, may change as rapidly as the underlying communications method allows, hundreds or thousands of times per second; consider rapid updates to a deformable mirror in an adaptive optics system as one example. Recognizing these two classes of data, the specific objective was to capture a nearly-complete subset of data, one sufficient to fully describe the systems in question, but which excludes telemetry that is demonstrably high-bandwidth noise. This must be done with care, as one engineer's noise may be another engineer's key data.

2.1 Storage of historical data

A relational database was our natural first choice for the primary storage platform. Among the key features driving that decision were: the need for read/write concurrency, from both the client and server perspective; rapid, granular access to data sets of all sizes; a common API to read and write data, and support in multiple languages for that API; and lastly, performance that will scale gracefully for exceedingly large data sets.

As they were a key component in the success of dynamic websites, free relational databases matured immensely over the last fifteen years; the efficiency and reliability of modern database platforms would be time consuming to create from scratch. Widespread API support for common languages and platforms are also available: at Lick Observatory, most of our tools are written in C, Tcl, Python, or Java; our platforms of interest were PostgreSQL or MySQL, both of which are well supported by all our languages of interest. Other database platforms, such as Sybase and Oracle, were not considered, primarily due to ease-of-deployment and cost concerns.

Choosing between PostgreSQL and MySQL was not a difficult or otherwise charged decision. With appropriate attention to tuning parameters, both platforms offer similar performance and stability for our relatively basic needs. There were no platform-specific features that could be leveraged by either our servers or clients to suggest a preference. We selected PostgreSQL for simple reasons: developer preference, and because it was already in widespread use elsewhere at Lick Observatory.

2.2 Simple database schema for live storage

The organization of historical data closely followed the structure of the original data itself. KTL represents data points as *keyword/value pairs*, where each value is a simple data type of boolean, enum, integer, float, double, or string [*]. Each keyword value has *binary* and *ascii* representations; for example, keywords relating to telescope pointing may have a binary floating point value in radians, which is translated internally to a sexagesimal ascii value for the benefit of user interfaces. Multiple keywords are combined into a *service*, which not only serves as an

[*] KTL does support fixed-length numeric array values, but they are rarely used in practice.

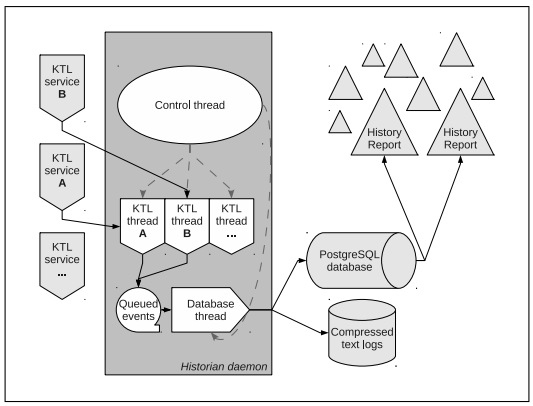

Figure 1. Control and data flow diagram for the historian

organizational construct, but also establishes the structural boundary between different KTL implementations. For example, different KTL services may use different internal communication mechanisms, and they may support subtle variations on the KTL API.

The resulting schema (see figure 2) mirrors this basic structure: an arbitrary database hosts a per-service table; the rows in the per-service table contain a timestamp, keyword name, binary value, and ascii value. In order to maintain the simplicity of the schema, both the binary and ascii values are recorded as text; no attempt was made to record values in a native column type.

time	keyword	binvalue	ascvalue
1317147765.02467	DISPSTA	0	Ready
1317147765.72498	TEMP	16.54	16.5
1317147766.95476	SETPOINT	0.00	0.0
1317147786.35852	TEMP	16.56	16.6
1317147792.94204	TEMP	16.53	16.5
1317147852.01083	TEMP	16.58	16.6

Figure 2. Sample data recorded for a simple temperature controller

Proper indexing on the key columns is absolutely necessary to achieve acceptable performance. In the average case, queries include both the keyword name and a time window; rare queries work individually on either the keyword name or the time alone. In order to maximize performance for the average case, an index was established combining the keyword and time columns. In order to accommodate atypical queries, individual indices on the keyword and time columns were also created. Our original schema was a single table which included the service name as a column; dividing the each service's data into a dedicated table reduced the need for additional indexing.

With the index on both the keyword and time columns, typical small queries on a reasonably modern Linux server (circa 2008, with plenty of physical memory) take 300-700 milliseconds to return; without that index, but with individual indexes on the keyword and time columns, similar queries take 10-30 seconds to complete, a sharp and unacceptable decline in performance.

2.3 Rapid handling of event broadcasts

A key function of KTL leveraged by the historian is asynchronous broadcasts of new keyword values. As new values arrive, the historian will process each event and queue the new values for insertion into the database. Because of the potential for high-frequency broadcasts, it was necessary to engineer the historian to do as little per-event processing as possible, lest the historian be overwhelmed by the quantity of incoming events.

For each event, the historian takes three essential steps: assign a valid timestamp for the event's arrival, acquire the binary and ascii representations of the keyword value, and insert the results into the database.

2.3.1 Timestamps

There are several different places along the KTL handling chain where one could assign a timestamp for a given event: when the event is created, when it is broadcast, when it is received, when it is processed, or anywhere in-between. In keeping with our desire to have the history be as accurate as reasonably possible, we wanted our timestamps to be as closely associated with the creation of the event as we could reasonably accomplish. The prototype of the historian acquired timestamps while processing an event; in a series of development steps, the acquisition of that timestamp moved closer to the event creation time: the next step was to use the timestamps provided by the KTL Python module at the heart of the historian; the mechanism used by KTL Python was subsequently improved such that the timestamp was acquired by the Python/C half of the module; KTL Python then enabled the conditional use of timestamps provided by the per-service client library. For the next optimization, we retooled both the fundamental MUSIC[5] communication layer and the `dtune` KTL client library such that the timestamp is assigned by the software daemon initiating the event broadcast, before the broadcast ever goes out over the network.

These incremental improvements not only improved the accuracy of the timestamp, they reduced the amount of per-event processing performed directly by the historian. While most historical queries are not likely to care about timestamp precision beyond a tenth of a second, the additional precision made it more straightforward to merge this historical record with other precisely timestamped data sources, such as log files.

Figure 2 shows the time column as a UNIX timestamp. This format was chosen chosen both for simplicity, and efficiency: acquiring a UNIX timestamp is a very fast function call in modern operating systems, and converting a UNIX timestamp to a human-readable format is readily supported by all of our languages of interest. In addition, the timestamp format used natively by KTL was also a UNIX timestamp. In our schema, the event timestamp is stored as a basic double precision floating point column type rather than a PostgreSQL timestamp-with-time-zone column type. The additional overhead and query complexity associated with a native timestamp column outweighed the potential benefit of guaranteed microsecond precision of stored timestamps; this is especially true since the system clocks themselves are not precise at the microsecond level.

2.3.2 Values

The values acquired by the historian are minimally processed. This processing includes two basic steps: determining whether the value changed since the last broadcast, and determining whether enough time has elapsed since the previous broadcast for continuously varying numbers.

To answer the first question, the historian caches the ascii value of the most recent broadcast; if the ascii value does not change between broadcasts, the event is not recorded. Besides weeding out redundant broadcasts, this has real value for many of the systems deployed at Lick Observatory: the ascii representation of a keyword is typically set to the maximum usable precision for a numeric value. A complete representation of the system need not include insignificant changes to a numeric parameter.

To answer the second question, the historian caches the timestamp associated with the last queued database insertion for a numeric keyword. If another event arrives before 0.9 seconds elapse, that event is placed on a discard queue. That discarded event will be queued for recording if and only if no further broadcasts occur in the subsequent 0.9 seconds. This time lag ensures that the historian throttles the recording of high-frequency numeric broadcasts to about 1 Hz. No such throttling occurs for non-numeric keywords, whose broadcast events are all considered significant.

2.3.3 Database insertions

When the historian receives an event broadcast, the essential details are appended to a queue of events waiting to be inserted into the database. This decouples the database operations from the asynchronous handling of incoming events, and because the incoming events are timestamped on (or before) arrival, no accuracy is lost by delaying the insertion. The pending queue of database events is common across all KTL services monitored by that historian instance, and is processed by a single background thread. Events are removed from the queue only after they are successfully recorded in the database.

In an effort to retain the simplicity of the design, no attempt was made to leverage transactions or otherwise batch process a large quantity of pending updates; each event is independently inserted. The historian assumes

that database insertions can be processed faster than events arrive; on the database server in use at Lick Observatory, individual insertions take ten to forty milliseconds to process, depending on system load. This suggests that a data rate on the order of one hundred events per second could result in data overflow; in practice, the aggregate load from multiple historian instances regularly exceeds that event rate without overflowing their respective queues.

3. INTEGRATION WITH EXISTING SOFTWARE

The KTL API focuses on the here and now, and does not provide hooks for querying the past. Rather than attempt to bolt such a capacity onto KTL, the focus turned instead to commonly used tools, and how they might be modified to query the recorded keyword history.

3.1 Selecting a prototype

One commonly-used set of command-line tools are the `*show` family of programs. Each individual `*show` command focuses on a different aspect of data presentation: single-value display, continuous scrolling display of values as they are published, columnar display of continuous values, and so on. The `*show` tools are used primarily in shell scripts and for debugging at the system level, and are read-only; write operations in these contexts are generally performed via the discrete `modify` command.

```
$ show -s hamiodine DISPSTA SETPOINT
                       DISPSTA = Ready
                      SETPOINT = 0.0 degrees C

$ cshow -s hamiodine -timestamp TEMP
Tue Sep 27 11:23:06 2011 TEMP = 16.6
Tue Sep 27 11:23:12 2011 TEMP = 16.5
Tue Sep 27 11:24:12 2011 TEMP = 16.6
```

Figure 3. Example usage of `*show` commands with a single KTL service

We provide simple and efficient access to the historian's data by leveraging existing an keyword-query tool, the `gshow` command, a powerful superset of both the `show` and `cshow` commands. If `gshow` is asked to display old data instead of current data, it will make optimized SQL queries on the database, transparent to the user.

3.2 Sample usage

With `gshow`, querying the database occurs when one or more `-date` arguments are provided at the command line. This usurps the regular control flow in the script; rather than issue requests via the regular KTL machinery, `gshow` queries the history database for the desired parameters, then resumes its normal execution path using the query results as its data source.

```
$ gshow -s hamiodine DISPSTA SETPOINT TEMP -timestamp \
  -date "September 27, 2011 11:23" -date "September 27, 2011 11:24:15"

2011-09-27 11:22:45.0246       DISPSTA = Ready
2011-09-27 11:22:45.7249          TEMP = 16.5 degrees C
2011-09-27 11:22:46.9547      SETPOINT = 0.0 degrees C
2011-09-27 11:23:06.3585          TEMP = 16.6 degrees C
2011-09-27 11:23:12.9420          TEMP = 16.5 degrees C
2011-09-27 11:24:12.0108          TEMP = 16.6 degrees C
```

Figure 4. Example usage of `gshow` command with a specific date range

Note the mismatch in figure 4 between the requested date range and the displayed results: the first three reported values are beyond the beginning of the requested range. `gshow` does not limit itself to reporting events strictly from the requested window, it ensures its results completely cover the designated period of time. In figure 4, the user requested a start time of 11:23 AM; the last recorded change for three of the requested keywords was some fifteen seconds prior, but correctly represented the system state during the requested window. In order to represent a complete picture of the system state, those earlier values were included in the results.

3.3 Query structure

The reporting of older events preceding a continuous stream of updates is analogous to the normal behavior of KTL client libraries. KTL has a *priming* continuous read system call, an initial request for the current value of a keyword combined with a subscription to all future broadcast events for that keyword. When querying the history database, `gshow` performs these priming reads via a distinct query (see figure 5) for each requested keyword.

```
SELECT 'hamiodine' AS service,* FROM hamiodine
WHERE
keyword = 'DISPSTA' AND
time    = ( SELECT MAX(time) FROM hamiodine
            WHERE keyword = 'DISPSTA' AND time < 1317147780
          )
UNION
SELECT ... WHERE keyword = 'SETPOINT' ...
UNION
SELECT ... WHERE keyword = 'TEMP' ...
ORDER BY time
```

Figure 5. Example priming query used by `gshow`, with one `SELECT` per keyword

A second set of queries retrieves all matching events that occurred during the designated time window (see figure 6). Because all of the remaining events are bounded by the same time constraints, the queries work on a per-*service* basis rather than a per-*keyword* basis.

```
SELECT 'hamiodine' AS service,* FROM hamiodine
WHERE
time BETWEEN 1317147780 AND 1317147855
AND
keyword IN ( 'DISPSTA', 'SETPOINT', 'TEMP' )
UNION
SELECT ...
UNION
SELECT ...
ORDER BY time
```

Figure 6. Example query used by `gshow`, with one `SELECT` per service

For both query types, it is important to note that the search clauses are all of the basic form `(time AND keyword)`. Because the queries require both the time and the keyword columns, they leverage the single index that includes both columns (see section 2.2).

4. POST-DEPLOYMENT ISSUES

After deploying the historian for production use, we encountered problems that required modest adjustments to our approach. Some of the issues encountered were anticipated during the design phase, but out of a desire to minimize complexity were not fully addressed until it became clear they could not be ignored.

4.1 Recording noise

Among the hundreds of keywords contained in a single service, it was typical to find a handful of keywords represented 95% of the recorded data. In many cases, this was the natural result of recording parameters that shift significantly over short time scales: for example, wind speed and direction. In others, it was clear that the recorded data was beyond the level of precision available to the system. One easy example was a telescope at rest: when quiescent, the reported position of the telescope should not change in altitude or azimuth. After combing the data for repeat offenders, several several keywords were adjusted to broadcast a level of precision more commensurate with the capabilities of the underlying hardware.

In other cases, the solution was not so direct. One example: at the Automated Planet Finder (APF) telescope at Lick Observatory, our engineers deployed a small fleet of temperature probes throughout the spectrograph and dome. Most of these sensors were attached to 12-bit analog inputs on a Galil motor controller; when scaled to a five volt range, the analog input is quantified in 0.00122 volt steps. The conversion from a voltage to a temperature for these systems is not at all linear, and in the region of interest a shift from 2.2 volts to 2.20122 volts is a shift of 0.15 degrees Celsius.

$$average_t = average_{t-1} * (1 - factor) + input * factor$$

Figure 7. Exponential smoothing formula used to reduce analog input noise

In order to handle the random sampling variations in these temperature probes, the software was modified to perform smoothing of the raw voltages. In our case, an exponential smoothing factor of 0.07 was used in the formula shown in figure 7 to calm down the noise. With a sampling interval of 300 milliseconds, this established an effective averaging time of about four seconds, and largely eliminated broadcast events in the absence of a genuine trend.

4.2 Automatic reconnect to network services

Considerable work occurred in 2011 and the early part of 2012 to shore up what was already a reliable infrastructure: in addition to modest performance improvements, the KTL platform used at Lick Observatory now features automatic connection recovery, transparently restoring functionality in the event of a network or service failure. Establishing these recovery features was necessary to ensure the captured history would not be marred by long stretches of missing data as a result of a transient infrastructure issue.

Similar to the gradual shift of responsibility with the timestamps, the initial attempt at connection recovery occurred within the historian itself. Doing so provided containment for any unanticipated side effects. After succeeding with the historian, the automated recovery mechanism migrated into the KTL Python layer, allowing all KTL Python applications to benefit from the new functionality. After succeeding with KTL Python, a similar fail-safe mechanism was engineered into the `dtune` KTL client library.

The KTL Python mechanism relies on each KTL service providing one or more *heartbeat* keywords. For the purposes of the recovery feature, the actual content published via the heartbeat keyword is immaterial, the important fact is that a broadcast event occurred at all. Thus, any keyword that reliably broadcasts on a regular frequency can be used as a heartbeat keyword; in practice, heartbeat keywords at Lick Observatory are generally an integer count of the number of seconds a given KTL server has been running. If no broadcasts are received for a heartbeat keyword within a defined interval, the background monitoring thread in KTL Python will attempt an external query against the same KTL service; if that attempt succeeds, all of the internal objects associated with the service are destroyed, recreated, and carefully reintegrated with other persistent objects.

In most circumstances, the `dtune` mechanism activates first, but is only available for KTL services that use the `dtune` KTL client library. Because there are a variety of non-`dtune` KTL client implementations in production use, it was necessary to retain the connection recovery mechanism in KTL Python to ensure the broadest possible coverage.

4.3 Updated client code not installed

Another source of missing records was a result of code development, as opposed to shifts of infrastructure. The software environment at Lick Observatory, while structured, is always in an ongoing state of development: services may receive new keywords, the network location of services may change, or whole new services will be added as new systems come online.

In any of these circumstances, the historian would miss data. This remains an unsolved problem at Lick Observatory; the current plan is to implement a script that will probe a source code checkout for missing updates, update the run time environment with any changes, and intelligently restart any affected instances of the historian. In the absence of an automated solution, it is incumbent upon the developer to update and restart the historian as necessary when the production software environment changes.

4.4 Migrating the database back-end

Early in the deployment of the historian, we needed to migrate the database back-end from its temporary home onto the production server that was purchased specifically for this purpose. Over the six months that the historian was active in testing, it acquired nearly thirty gigabytes of data, as measured in a text database dump. The size of the file alone was problematic to work with; the temporary server did not have enough disk space to store both the raw PostgreSQL binary data as well as the dump file; once acquired, editing the dump file to match the production table layout required careful splitting and subsequent concatenation. To make matters worse, the hard drive on the temporary host developed a rash of sector read errors, which were no doubt uncovered by the extraordinary increase in disk activity due to the attempted migration.

The migration attempt eventually resulted in failure. The resource consumption issues were solvable, though tedious to work around; by contrast, the data corruption resulting from the bad disk sectors proved far too time consuming to work around. Even if a solution were readily available, the migrated data set would have gaps, and it would have been difficult to determine the quality of the remaining data. Both of these concerns, the impracticality of full data dumps and the vulnerability to a single source of failure, prompted the development of a secondary storage mechanism for the historian's data stream.

As mentioned in section 2.3.3, a single thread performs all database insert commands for a single historian, regardless of how many KTL services are covered. That same thread also writes each row of data to a logging object. The object, in turn, is responsible for writing the data to a per-service log file, and to automatically open a new log file on a weekly basis. In order to mitigate the migration concerns, the content of the log file was structured to precisely mimic the text formatting used in PostgreSQL's database dumps; if and when the database back-end needs to migrate to a new host, the secondary log files could be concatenated together and fed to a bulk `COPY` command for rapid ingestion of old data.

5. HISTORY IN ACTION

Turning the focus away from the general design of the historian, we will now look at a few examples of how the history database contributed to solving real-world problems. The selected examples highlight three fundamentally different ways the historian contributed to solving the problem at hand: direct retention of unique events, instant recall of recent events, and long-term collection of trending data. Each one of these examples also represents a different class of problem: diagnosing a specific hardware fault, resolving a subtle software infrastructure issue, and characterizing the performance of a system.

5.1 Unique events: diagnosing a hardware failure

In early April, research astronomer Robert Kibrick experienced a dome shutter failure at the APF telescope. When the APF dome receives a request to close, both the front and rear shutters are sent to their respective *home* positions via a hardware-controlled sequence of moves. When the rear shutter failed to close, it drove itself away from the home position until it encountered a hardware limit, and the close sequence aborted.

The initial hypothesis was that the physical switch used to determine when the rear shutter was at the home position had failed, and in such a way that the signal was always active. When the rear shutter is at the home position, the first step in the hardware-controlled homing sequence is to move the shutter off the home position.

If that signal never transitions, the programmed sequence does not recognize the failure, and would drive the shutter into a hardware limit. Direct probing of the switch did not confirm this hypothesis: when tested in isolation, the switch appeared to work correctly.

```
2012-04-02 20:05:58.8798      RSATHOME = True
2012-04-02 20:05:58.8798      RSATHOME = False
2012-04-02 20:05:58.9443      RSATHOME = True
2012-04-02 20:05:58.9516      RSATHOME = False
2012-04-02 20:05:59.2509      RSATHOME = True
2012-04-02 20:05:59.2509      RSATHOME = False
2012-04-02 20:05:59.2825      RSATHOME = True
2012-04-02 20:05:59.2826      RSATHOME = False
```

Figure 8. Sample of the thousands of recorded state transitions for the APF rear shutter home signal

In search of an alternate hypothesis, the recorded history of keyword events was inspected for clues. This inspection immediately yielded results: the observed behavior of the system was confirmed, in that the failures of the homing sequence were the result of the limit switch being in the incorrect state; in addition, it was immediately clear that the problem was intermittent, and most commonly associated with specific positions along the rear shutter's range of motion. When otherwise idle, the limit switch signal would occasionally oscillate hundreds of times per second, and would randomly stop on either *True* or *False* (see figure 8). This discovery allowed observatory personnel to refine their search, in that they started looking for an intermittent wiring problem as opposed to a mechanical issue. In the end, it was discovered that the wiring terminals inside the limit switch had corroded due to moisture intrusion.

5.2 Instant recall: isolating a software infrastructure glitch

Technicians at Lick Observatory reported occasional display issues with a specific graphical interface for a specific instrument; the error was strangely limited, in that it only affected a small set of keywords, and was not at all consistent. Worse, the error seemed to evaporate after adding debug statements in key control points, suggesting that the root cause could be an unidentified race condition.

The essential problem was that the `dtune` KTL client library was not reliably communicating a published broadcast to a client application. Multiple applications running on the same host generally all experienced the glitch at the same time; applications running on distinct hosts were less consistent. After some careful comparisons of related keyword events, it became apparent that these glitches were also affecting the history database, and that the recorded history, or lack thereof, could assist with tracking this problem down.

```
2012-05-15 13:26:27.5779      SLITSTA = Ready
2012-05-15 13:26:27.8426      SLITMOO = Off
2012-05-15 13:26:29.6720      SLITSTA = Moving
2012-05-15 13:26:29.8365      SLITMOO = On
2012-05-15 13:26:32.8538      SLITSTA = Ready
2012-05-15 13:26:34.8960      SLITSTA = Moving
2012-05-15 13:26:38.0839      SLITSTA = Ready
2012-05-15 13:26:40.1633      SLITSTA = Moving
2012-05-15 13:26:43.3539      SLITSTA = Ready
```

Figure 9. Absence of proper broadcasts: each state (STA) transition should have a matching motor-on-off (MOO) transition

The debugging statements added to `dtune` produced an enormous amount of output, as they provided an extremely close look at the minute behavior of some routines inside `dtune`, and were triggered on every broadcast of any native integer keywords from the KTL service in question. By querying the history database, it was possible to determine precisely when one of these glitches occurred, and track down that specific time in the voluminous

debug output. In the end, it was determined that `dtune` did not properly handle the case where it received a broadcast for a keyword that the local instance of the KTL client library was not configured to recognize; when events were bundled together into a single broadcast, all events following the one associated with the unknown keyword were corrupted.

5.3 Trending data: characterizing thermal stability

As part of commissioning the spectrometer at the APF telescope, the instrument scientist wanted to characterize the performance of the system at different temperatures. Part of that work required determining how quickly different systems responded to temperature changes, in addition to looking for correlations between performance degradation and shifts in specific temperature readings.

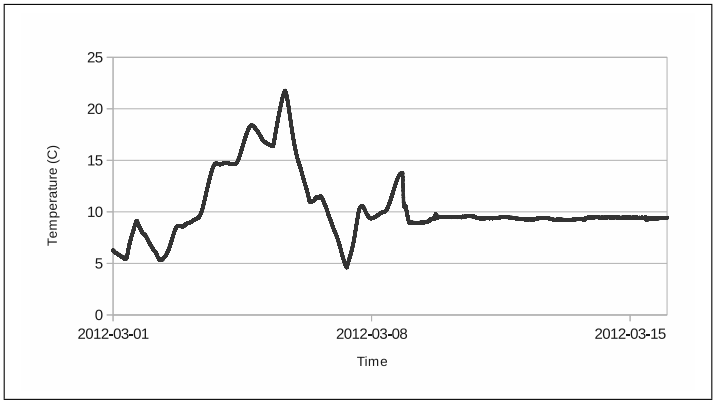

Figure 10. Variation in average APF secondary mirror strut temperature over time

The history database neatly provided the necessary data to perform both of these analyses, without the need for additional one-off monitoring or logging applications, and without parsing any log files. PostgreSQL has the ability to directly output specific queries as comma-separated value (CSV) files, and `gshow` was instrumented to have a similar capability. The instrument scientist was able to import the CSV data into a familiar spreadsheet program, and perform the analysis required. See figure 10 for an example extract of temperature data; in this case, one can see the variation in strut temperature on a day-to-day basis as the internal dome temperature goes uncontrolled, and the comparatively tight thermal variation after reactivating the climate control system.

6. FUTURE ISSUES

6.1 Index performance with large data sets

As the quantity of rows in the history database grows unbounded, there is a natural concern about the performance of the underlying indexing method. The default indexing algorithm for PostgreSQL is a Bayer tree[6] (commonly referred to as a B-tree), the de facto standard for database indexing. The specific Bayer tree variant used by PostgreSQL is a Lehman-Yao high-concurrency B+-tree.[7]

The average time required to look up a value in a Bayer tree is $O(log_b n)$, where b is the Knuth order[8] of the Bayer tree and n is the total quantity of records. The Knuth order for a PostgreSQL B+-tree varies depending on the nature of the data being indexed, and the page size at compile time; for the index on the keyword and time columns, and a default page size of 8,192 bytes, the Knuth order should between one-hundred and two-hundred, depending on how evenly the tree is balanced.

Thus, the index performance should remain very stable as the quantity of rows increases; at Lick Observatory, the performance is very reasonable after six months of data accretion: it will take another fifty years of data accumulation to make a significant change in the observed performance.

6.2 Modifications to database schema

As long as the database performance using monolithic per-service tables remains acceptable, there will be little motivation to pursue modifications to the underlying schema.

In section 2.2, we highlighted the use of a single table for each KTL service. Extrapolating that one step further, one might suggest that the number of indices could be further reduced by having a separate table for each keyword within each service. Doing so would quickly trigger performance issues: since each KTL service typically contains hundreds of keywords, there would be thousands of individual files stored in the PostgreSQL data directory. The performance of an average UNIX filesystem will start to bog down with that many files in a single directory.

More effective, given the concerns highlighted above about the unbounded increase in quantity of records, would be an automated rollover of database tables based on temporal boundaries, perhaps annually. The simple schema with monolithic tables allows the use of very straightforward SQL queries to retrieve data; if the data were divided by year, the complexity of those queries would increase in the case where the desired data set spanned multiple years.

6.3 Running out of disk space

A more pressing concern is the consumption of disk space. One example service at Lick Observatory encapsulates most of the parameters associated with the spectrometer for the APF telescope. There are just under six hundred keywords in this service, covering diverse hardware components such as servo motors, temperature probes, and calibration lamps.

In an average week, this example service generates 9.5 million unique broadcast events. That average week consumes 426.3 megabytes of disk space in the uncompressed secondary log alone; compressed using the LZMA2 algorithm, that log file shrinks down to 77.7 megabytes, better than a 5:1 compression ratio. The heavyweight storage is on the PostgreSQL back-end itself, where that same average week consumes 1,435.3 megabytes of disk space. Of that, 821.4 megabytes (57.6%) are consumed by indices alone; the total on-disk footprint could be reduced by about a third by dropping the rarely used individual indices on the keyword and time columns.

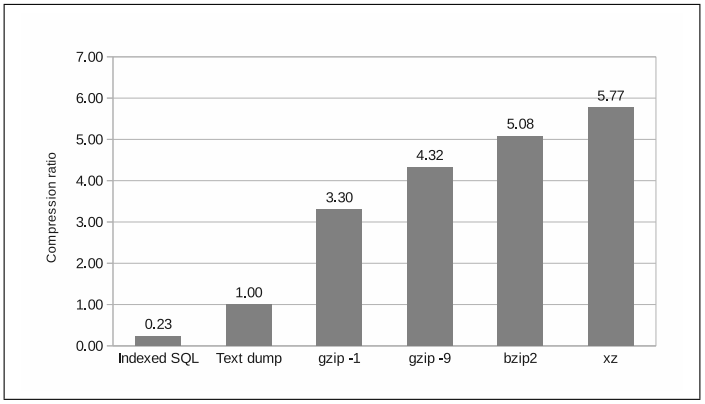

Figure 11. Example compression ratios for long-term storage of a single week's data

The APF spectrograph is not yet in production use; if it were, the average weekly broadcast total would be much higher. Assuming there are ten such services monitored by the historian, that rounds to fifteen gigabytes of data per week, 780 gigabytes per year. At that rate, the mirrored 2.7 terabyte disks in the production server will conservatively handle three years of data before requiring an infrastructure upgrade. Dropping the indices mentioned above would allow another year and a half of breathing room.

7. CONCLUSION

The complete history of all KTL keywords at Lick Observatory was initially little more than an interesting exercise: the idea had potential, and would not be time-intensive to develop and test, but there was no expectation for it to yield immediate results.

An accurate window into the past proved far more practical than expected. As problems came up, the historian's role in the problem-solving process changed from tentative curiosity, to respectful inclusion, to requiring the historian as an essential workflow component. By adhering to our basic principles of simplicity and reliability, a small time investment produced a useful new capability, and practical results closely followed the initial deployment.

In deploying the historian, we realized our two initial goals: improving the resources available to isolate and remedy both systemic and transient issues, and enabling the analysis of long-term trends in system performance. In addition to those discrete goals, the deployment of the historian motivated substantive improvements not only to specific software systems, but to the general software architecture used at Lick Observatory. The historian acted as an enforcer for important best practices: adherence to standards, rigorous use of version control systems, and efficient use of resources. Another unanticipated bonus was the historian's value for *current* engineering tasks. Since history is recorded right up to the present moment, there is less second-guessing of which data are of immediate interest; the emphasis shifts to *retrieving* data, rather than on *recording* data.

We look forward to realizing further unanticipated benefits as Lick Observatory's complete history of everything unrelentingly accretes all available data.

REFERENCES

[1] Lanclos, K. et al., "Realizing Software Longevity over a System's Lifetime," *Software and Cyberinfrastructure for Astronomy* **7740**(3E), SPIE (2010).

[2] Allen, S. L. and Clarke, D. A., "Three Instruments in Four Years: The UCO/Lick Data-Driven Toolkit," in [*Astronomical Data Analysis Software and Systems IX*], Manset, N., Veillet, C., and Crabtree, D., eds., **216**, 339 (2000).

[3] Conrad, A. R. and Lupton, W. F., "The Keck Keyword Layer," in [*Astronomical Data Analysis Software and Systems II*], R.J. Hanisch, R.J.V. Brissenden, & J. Barnes, ed., *Astronomical Society of the Pacific Conference Series* **52**, 203–207 (Jan. 1993).

[4] Lupton, W. F. and Conrad, A. R., "The Keck Task Library (KTL)," in [*Astronomical Data Analysis Software and Systems II*], R.J. Hanisch, R.J.V. Brissenden, & J. Barnes, ed., *Astronomical Society of the Pacific Conference Series* **52**, 315–320 (Jan. 1993).

[5] Stover, R. J., "MUSIC a Multi-User System for Instrument Control," *Lick Observatory Technical Reports* **54** (1989).

[6] Comer, D., "Ubiquitous B-Tree," *ACM Comput. Surv.* **11**, 121–137 (June 1979).

[7] Lehman, P. L. and Yao, S. B., "Efficient Locking for Concurrent Operations on B-Trees," *ACM Transactions on Database Systems* **6**, 650–670 (1981).

[8] Knuth, D. E., [*Sorting and Searching*], vol. 3 of *The Art of Computer Programming*, Addison-Wesley Professional, second ed. (May 1998).

Kyle Lanclos is a member of the Scientific Programming Group at UCO/Lick Observatory. He received his bachelor's in computer science with a minor in astrophysics from the University of California at Santa Cruz in June 2000, and began working with UCO/Lick in December 2000.