SAND2011-8035C

An Optimized Database for Spacecraft State-of-Health Analysis

Steve Lindsay
Sandia National Laboratories
1515 Eubank Blvd SE
Albuquerque, NM 87185
505-284-6603
srlinds@sandia.gov

Clark Poore
Sandia National Laboratories
1515 Eubank Blvd SE
Albuquerque, NM 87185
505-844-3667
capoore@sandia.gov

Abstract—**This paper describes the design and implementation of an optimized state-of-health (SOH) database. We include performance data for querying measurands with various sample rates and with various change rates across multiple time intervals.**

TABLE OF CONTENTS

1. INTRODUCTION

Spacecraft SOH data is typically downlinked to a ground system and stored in flat files for the duration of the mission. Depending on the active measurand count and their sampling rates, these files can be overwhelmingly large in both size and number. In addition, analyzing individual SOH points and values and performing long-term trend analysis usually requires custom software that is expensive and difficult to write and maintain.

An alternative approach is to ingest these files into a database management system (DBMS) so the resulting measurands can be analyzed and trended with standard tools such as Structured Query Language (SQL). Unfortunately, DBMS implementations for time series data frequently require vast amounts of disk space and computational resources, and even then result in query performance that is intolerably slow.

In this paper, we present our design of a SOH database that is optimized for both storage volume and query performance by leveraging compression and partitioning. We then present our implementation results for a Department of Energy (DOE) remote sensing spacecraft, which transmits over 29,000 measurand values sampled at various rates. We explain how we store the data in both online and offline capacities, how we maintain total-life long-term trending, and how we optimize query performance to retrieve high-resolution data in a minimal amount of time for a given measurand. Our implementation facilitates both short- and long-term health maintenance as well as real-time anomaly resolution within a design based on commercial off-the-shelf (COTS) software.

2. REQUIREMENTS

Like many software endeavors we began with a set of requirements, and we present these in Table 1.

Table 1. High-level Requirements

No.	Description
1	Ensure queries are as responsive as possible
2	Query data in different resolutions to accommodate different user needs
	a. High-resolution data for in-depth analysis and statistical processing
	b. Change-only data for rapid visualization
	c. Summarized data for trend analysis
3	Keep data online and easily accessible
	a. Recent detailed data for as long as possible
	b. Summarized data for entire mission duration

Of particular note is requirement 2, which calls for query resolutions unique to the client making the request. For example, an analyst troubleshooting an ongoing anomaly needs a relatively short interval of what we call high-resolution data: every value of a given measurand as sampled on the spacecraft. Conversely, a project manager needs summarized data to facilitate trend analysis of consumable or limited-life resources; he or she likely wants to see the minimum or maximum values of specific measurands since launch. Either of these users may wish to view a strip chart consisting only of data changes to identify the magnitude and timing of variations. Additionally, a data mining application may require high-resolution data for a longer period of time – months rather than days.

3. DESIGN

To satisfy requirements 2 and 3, we needed to incorporate a compression scheme. We chose to leverage the fact that groups of measurands typically share the same timestamp.

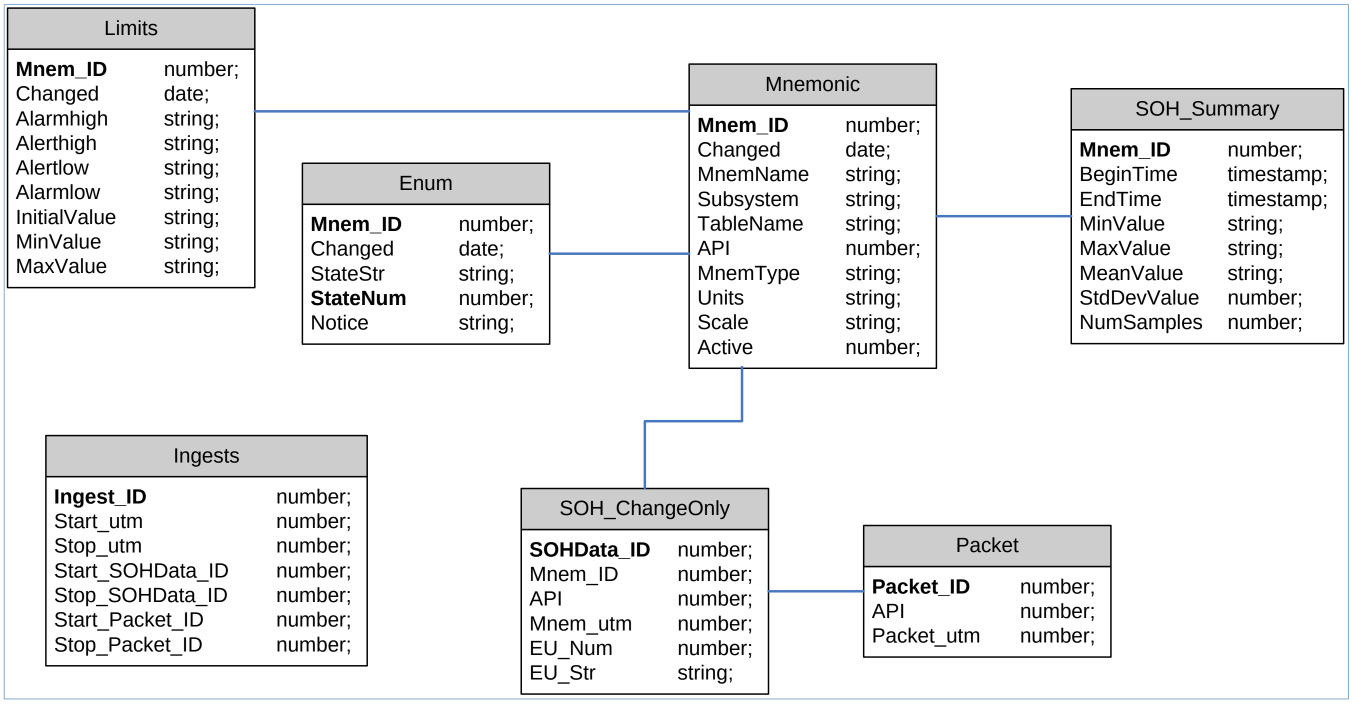


Figure 1. Database Schema

For example, one packet application process identifier (API) in Consultative Committee for Space Data Systems (CCSDS) packetized telemetry, or one minor frame in time division multiplexed (TDM) telemetry, share a common timestamp for their corresponding measurands. We store each item in this grouping as a single record in every case. For individual measurands, we store its first occurrence and then only those values that have changed in relation to the previously stored value. For simplicity, and due to its prevalence in modern systems, we assume the use of CCSDS packetized telemetry for the remainder of this paper.

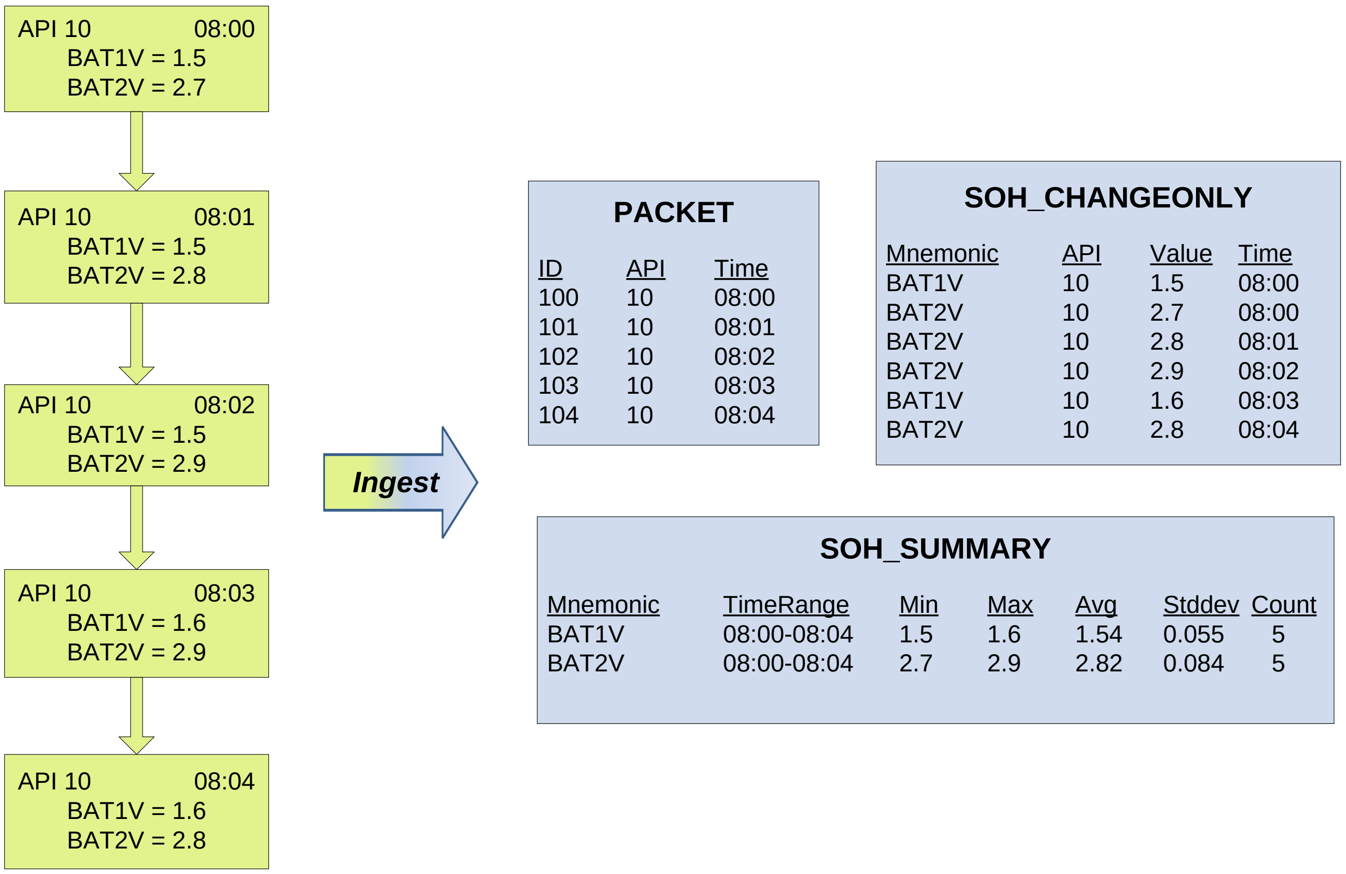
Schema

We chose two tables to represent the above distinction: `packet` and `soh_changeonly`. By separating the data in this manner, our system can retrieve data in ways that make the most sense to the client. To aid in trend analysis, we created a third table called `soh_summary` that stores statistical aggregations to include minimum, maximum, average, and standard deviation values for a given period. We can then populate this table at the end of the chosen period – say, each day – and therefore avoid having to calculate these values on the fly. More importantly, we can keep this summary data online for the life of the system due to its coarser granularity. The `packet` and `soh_changeonly` data, however, is typically much more expansive and potentially requires a finite rolling online window for a multi-year space mission.

Figure 1 shows the complete database schema. The `mnemonic` table contains all relevant information about a measurand and its origin. The `limits` table contains nominal and critical limits and is stored separately since it applies only to a subset of measurands. The `enum` table contains enumerated state information for discrete-type measurands. The `ingests` table provides a way to track ingest activity. Note that we avoid defining an explicit primary key for the SOH tables, as it would only increase execution time and space requirements for the associated index while providing little additional value. We do, however, provide an implicit primary key for these tables in order to efficiently refer to a given row when needed.

Population

Figure 2 illustrates the use of our schema for a simple case of one API with two corresponding measurands: voltage readings for Battery 1 (BAT1V) and Battery 2 (BAT2V). As packets arrive every second, the ingest process stores each one in the `packet` table which also adds an internal numeric ID. The `soh_changeonly` table stores the first occurrence of each measurand followed by its changes. Upon ingest completion, the `soh_summary` table is populated with the corresponding aggregate values.



PACKET

ID	API	Time
100	10	08:00
101	10	08:01
102	10	08:02
103	10	08:03
104	10	08:04

SOH_CHANGEONLY

Mnemonic	API	Value	Time
BAT1V	10	1.5	08:00
BAT2V	10	2.7	08:00
BAT2V	10	2.8	08:01
BAT2V	10	2.9	08:02
BAT1V	10	1.6	08:03
BAT2V	10	2.8	08:04

SOH_SUMMARY

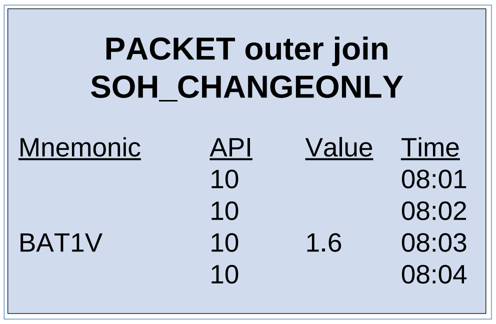
Mnemonic	TimeRange	Min	Max	Avg	Stddev	Count
BAT1V	08:00-08:04	1.5	1.6	1.54	0.055	5
BAT2V	08:00-08:04	2.7	2.9	2.82	0.084	5

Figure 2. Example Population

4. QUERYING

While querying summary and change data is straightforward, querying the high-resolution data is more nuanced. Unfortunately this query cannot be implemented as a simple outer join of packet and `soh_changeonly` for two reasons. First, the requested start time may not align with a value in `soh_changeonly`, so the result set needs the ability to look further back in time than originally requested by the client. Second, the `packet` table has no concept of a measurand value or name, so it must be manually inserted in the results exclusive to the `packet` table – that is, those results that do not reflect a change in value.

These issues are illustrated in Figure 3, which presents an outer join for BAT1V between 8:01 and 8:04 from our example. Because BAT1V logged its most recent change of 1.5 prior to the requested start time, its value is missing from the start of the dataset. This misalignment also results in the omission of its name and any other pertinent information until the first change is encountered at time 8:03.



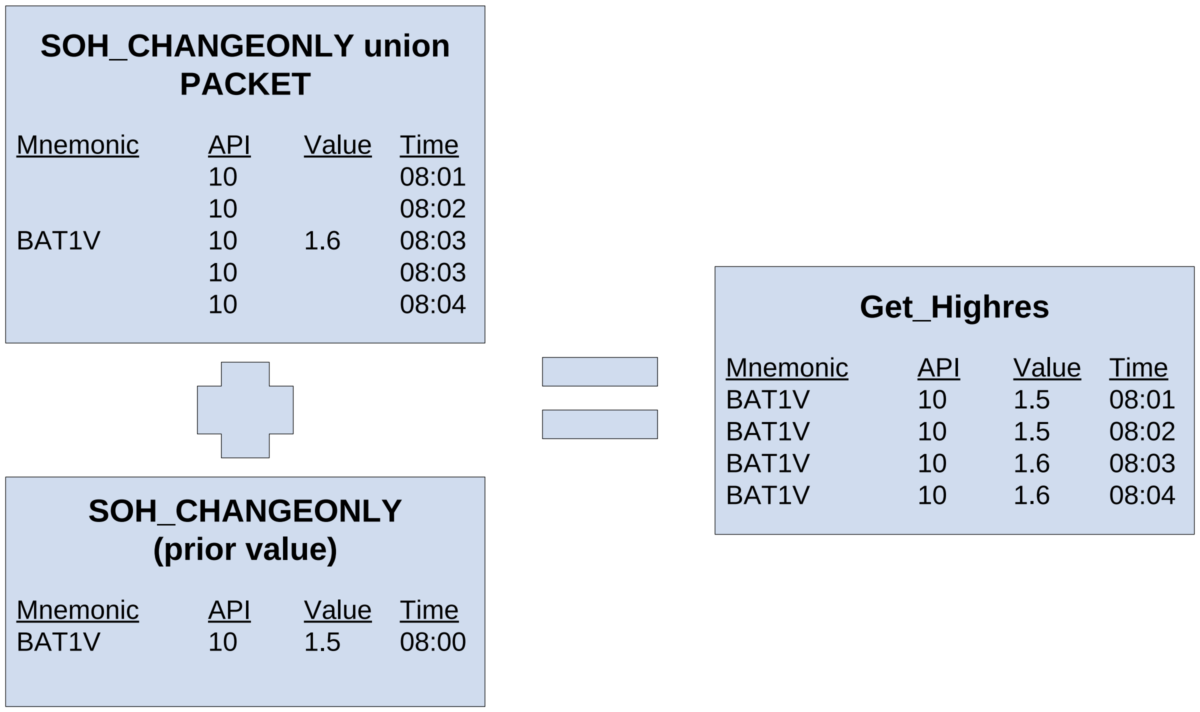
PACKET outer join SOH_CHANGEONLY

Mnemonic	API	Value	Time
	10		08:01
	10		08:02
BAT1V	10	1.6	08:03
	10		08:04

Figure 3. Outer Join - Incorrect

As an alternative to the outer join, we implemented a pipelined function called `get_highres` that takes three parameters – mnemonic ID, start time, and end time – and returns a result set. The function executes a union query on `soh_changeonly` and `packet` for the applicable time range, ordered by time ascending. If the first row does not correspond to a `soh_changeonly` record, it runs a separate query to seed the resultset with the most recent measurand value prior to the start time. It then iterates through the union, discarding redundant `packet` rows while matching nonredundant `packet` rows with the previous measurand value from `soh_changeonly`. For added efficiency, it executes the union query as a bulk collect.

This union query and transformation is shown in Figure 4, producing the fully complete result set shown on the right-hand side of the figure. Note that we leverage our `get_highres` function for populating `soh_summary` and for more sophisticated analysis activity, since it represents the true uncompressed values as sampled on the spacecraft.



SOH_CHANGEONLY union PACKET

Mnemonic	API	Value	Time
	10		08:01
	10		08:02
BAT1V	10	1.6	08:03
	10		08:03
	10		08:04

SOH_CHANGEONLY (prior value)

Mnemonic	API	Value	Time
BAT1V	10	1.5	08:00

Get_Highres

Mnemonic	API	Value	Time
BAT1V	10	1.5	08:01
BAT1V	10	1.5	08:02
BAT1V	10	1.6	08:03
BAT1V	10	1.6	08:04

Figure 4. Union Query - Correct

5. PERFORMANCE CONSIDERATIONS

To make our schema manageable and efficient, we incorporated table partitioning. We chose to partition the `soh_changeonly` and `packet` tables by date-range not only because users most frequently query current data, but also because it provides a natural way to archive old data to make space for new data. When available data file space cannot accommodate a partition expansion, our system exports the oldest partitions to a separate disk area and deletes the obsolete data files. These partition exports remain available if an analyst needs to query outdated high-resolution data, at which point they can be imported on a case-by-case basis.

We designed our system so the partition interval of the high-resolution tables can be changed over time. For example, if a partition interval of seven days is too large and results in bloated data files and long-running partition scans for queries, this interval can be reset to a fewer number of days. For the `soh_summary` table, however, we kept the partition interval fixed at one year. Our rationale is that space missions are typically measured in years, and even though most summary queries range from beginning of life to current time, it still provides a way to divide the potentially millions of rows into a manageable set of underlying data files.

Ingest performance depends greatly on the location of the logic for determining whether a new value for a given measurand represents a change from its previous value. The two choices are internal to the database, or external in a separate module. For the internal method, the ingest module sends every new value to the database, which then examines it for change prior to insertion. In the external method, the ingest module filters redundant values and sends only changes to the database. We chose the external method to minimize both database table lookups and network traffic.

6. IMPLEMENTATION

We deployed our system on a Dell PowerEdge R710 with 48G RAM and two 6-core hyper-threaded Intel Xeon CPUs clocked at 3.33 GHz, running 64-bit Red Hat Enterprise Linux Server 5.5 as the operating system. We attached two Dell PowerVault MD1220 storage arrays for 12 TB of total disk space, with 16 disks comprising the data file storage area using a 1MB stripe size. We configured the data file storage area to use the ext3 file system to leverage both direct and asynchronous I/O. We chose Oracle 11g as our DBMS due to its strong reputation for handling large amounts of data and its highly configurable parallel query execution environment. We implemented our ingest module in C++ to maximize run-time performance, and our administration utilities in Perl to leverage the rich set of available database interface modules [1].

Our implementation supports a DOE remote sensing spacecraft providing more than 29000 measurands divided into 728 unique packet APIs. The data rates for each API range from 1 sample per day to 64 Hz, with an average data rate of 0.3 Hz. Our initial tests with lab data showed we could expect several million rows of `packet` data daily, with `soh_changeonly` requiring a row count 5x-10x that of `packet`. As a result, we chose a partition interval of 1 day.

We made several internal database configuration changes to maximize performance [2]. First and foremost, we configured the database to run in noarchivelog mode to eliminate the overhead of logging table insertions. While this decision makes disaster recovery less elegant, our safety net lies in the fact that we can always revisit the SOH flat files to repopulate lost data.

Next, we configured Oracle's parallel execution environment. After experimenting with different settings for the *parallel_degree_policy* initialization parameter, we found that a setting of manual (versus automatic) best leveraged our partition-based architecture. Surprisingly, the automatic setting resulted in no query parallelization, even when we lowered *parallel_min_time_threshold* to a value of one. Finally, we configured the *filesystemio_options* parameter to `SetAll` to ensure the database supported both direct and asynchronous I/O, consistent with the ext3 file system configuration [3].

With separate data and index tablespaces, this implementation produces 732 data files annually. We created three locally partitioned indexes: one for the `packet` table and two for `soh_changeonly`. Taking into account on-orbit data for the six months of operations corresponding to February through July of 2011, our row counts and corresponding file sizes are shown in Table 2.

Table 2. Row Counts

	Row counts				**File sizes**			
Object type	**Min**	**Max**	**Avg**	**Sum**	**Min**	**Max**	**Avg**	**Sum**
Data	1.7M	31.3M	14.9M	5.4 B	48.4 MB	1.68 GB	769.5 MB	278.6 GB
Index	1.7M	31.2M	18.8M	10.2 B	86.8 MB	1.68 GB	1.01 GB	547.8 GB

7. Results

We first analyzed the space savings of our compression schema. A typical day on orbit creates 3M rows in `packet` and 27M rows in `soh_changeonly`, and an aggregate total of 150M distinct values from `get_highres` for all active measurands. In other words, if instead we had chosen to insert into a single table every value sampled for every measurand, we would have inserted 150M rows daily. From a row count perspective, which is closely tied to query performance, this translates into a savings of 80%. To quantify space savings on the file system, we note that the `packet` and `soh_changeonly` tables require three and six fields respectively, while the single table would require five fields. By measuring the daily counts of populated fields, our compression scheme requires 162M fields versus 750M fields for the every-value approach. Our space savings is therefore 78%.

We then timed our database with respect to insertions and retrievals. Oracle's statistic monitoring utilities show that each row insert operation, implemented via a stored procedure call from the ingest client, takes an average of 0.5 milliseconds (0.0005 sec) to complete. Empirically, after accounting for ingest parallelization and process overhead, we require 2 hours of wall clock time to ingest 24 hours of SOH data, or a data-generation to data-consumption ratio of 12 to 1.

However, retrieval time is arguably the most important metric, as it dictates how quickly an analyst can troubleshoot a time-critical anomaly. Because retrieval via `get_highres` is a union query of both `packet` and `soh_changeonly`, we expected the required time to depend on the number of rows in these tables for a given measurand. We also expected the retrieval time to be dependent on the number of partitions accessed.

With these factors in mind, we designed benchmarking scenarios that covered three periods: one day (one partition), one week (7 partitions), and one month (28-31 partitions). We also chose three different packet rates for each scenario: 4 packets/sec, 1 packet/sec, and 0.2 packets/sec (or one packet every five seconds). Finally, we chose measurands with change counts that varied between 2 per day and 86000 per day (or one change per second). To avoid biasing our results due to query caching, we queried a given time period no more than once for any given packet API, and ultimately we queried data dispersed throughout each of the six months between February and July 2011. We used the timing command in Oracle's sqlplus utility to invoke the `get_highres` procedure and return only the first row of data. In this manner, we avoided including network or output latency in the results.

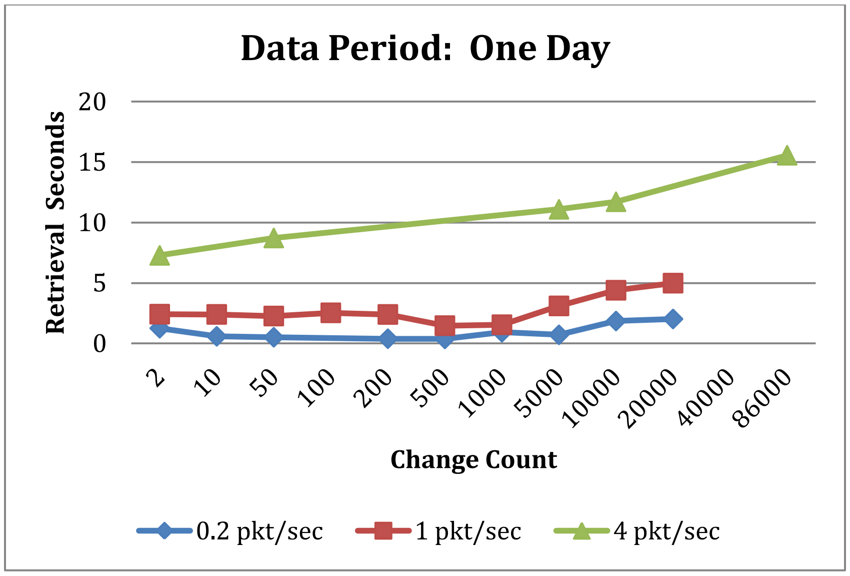


Figure 5. Retrieval Times - One Day

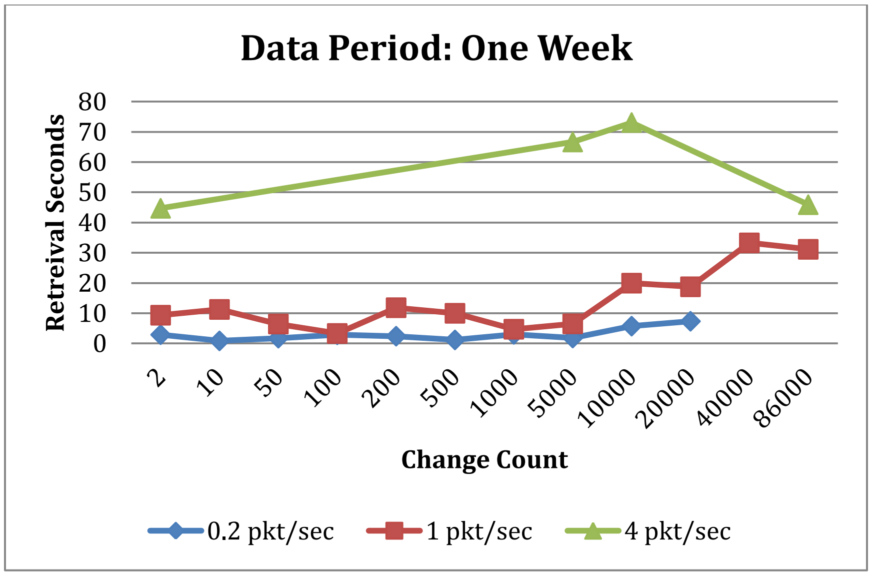


Figure 6. Retrieval Times - One Week

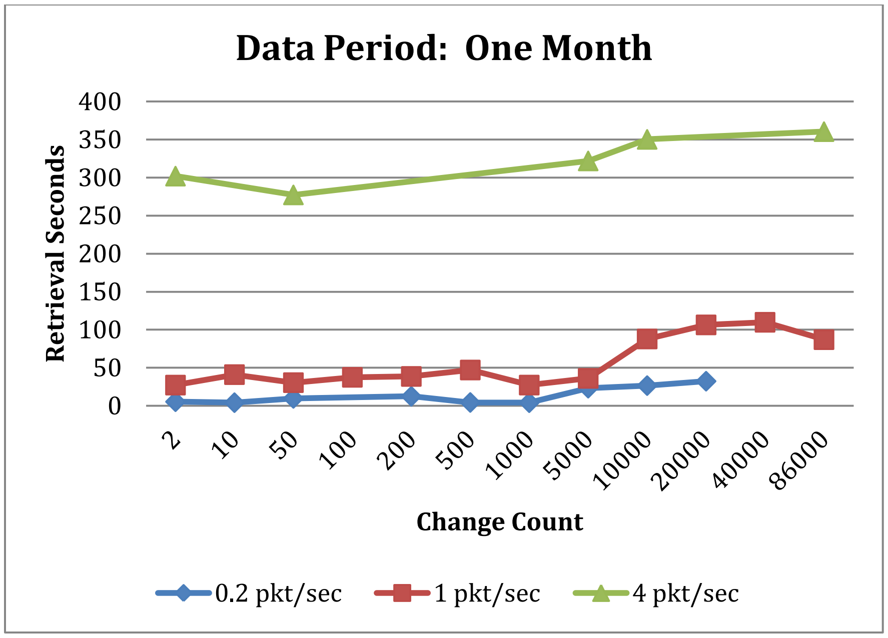


Figure 7. Retrieval Times - One Month

Figure 5 through Figure 7 show our results for these three data periods and the various change counts of the associated measurands. For a single day, a client can expect to wait between 1 and 5 seconds for packet rates up to 1 packet/sec, and between 5 and 16 seconds for the higher rate of 4 packets/sec. For a one-week period, these waits translate to between 1 and 33 seconds for rates up to 1 packet/sec, and between 44 and 73 seconds for the higher rates. To retrieve an entire month of high-resolution data, the query times for the lower packet rates range between 4 and 110 seconds, while for the higher rate data this range varies from 4 to 6 minutes.

8. Conclusions

We were pleased to find that all our queries for 24 consecutive hours of data returned in only a few seconds for our low and medium packet rates, and at most 16 seconds for our highest rates. We were also pleased to see that a query for an entire week of data was still under a minute for low- and medium-rate data, and right around a minute for high-rate data. In our experience, these are the queries that an analyst will most often require during critical anomaly troubleshooting, where the future viability of a multi-million dollar orbiting asset hangs in the balance.

The longer period of one month understandably requires a longer query time: approximately one minute for our low- and medium-rate data, and approximately five minutes for our high-rate data. We feel these times are acceptable, especially since the operational need for such a query is relatively low. After all, if an analyst needs to characterize a measurand trend for a given month, he or she will most likely query the `soh_summary` table, which will return its results in just a few seconds even after years on orbit.

9. Summary

Our SOH database satisfied all of its initial requirements and continues to provide valuable results to the DOE mission it serves. The next step is to fully leverage its capabilities by discovering hidden trends and relationships between measurands and hopefully warn of on-board issues and anomalies before they occur. Toward this end, data mining and other predictive analytic techniques appear to be excellent candidates, and we look forward to exploring these possibilities as our data set grows.

References

[1] Perl documentation web site: perldoc.perl.org.

[2] Oracle® Database Administrator's Guide 11g Release 2 (11.2), Oracle Corporation, April 2011

[3] Oracle® Database Performance Tuning Guide 11g Release 2 (11.2), Oracle Corporation, July 2011.

BIOGRAPHIES

***Steve Lindsay** received a B.S. in Computer Science from The University of Kansas in 1989, and an M.S. in Computer Science from the Air Force Institute of Technology in 1994. He served in the U.S. Air Force for 9 years as a software engineer and systems analyst, and he has been with Sandia National Laboratories for more than 13 years. During his 20 years in the space domain, he has worked on numerous satellite ground station initiatives for both the Department of Defense and the Department of Energy in roles varying from software development, system engineering, and data management.*

***Clark Poore** received a B.S. in Computer Science from the University of New Mexico in 1999, and an M.S. in Computer Science from the New Mexico Institute of Mining and Technology in 2002. He has been a software engineer with Sandia National Laboratories for more than 13 years. He has worked on several satellite ground station programs in software design and development with an emphasis in management, visualization, and data mining of the large volumes of data generated by satellite systems.*