Commentary on SkySpark Core Technologies:

Fantom, Folio and Axon

Introduction

SkyFoundry has invested 10 years of development in the SkySpark software platform. It is uniquely designed from the ground up for the realities of IoT applications in the "Built Environment" and has been proven in deployments across more than 1 Billion sq. ft of facilities worldwide (> 92 Million square meters).

When helping customers with a detailed technical evaluation of SkySpark, we are often asked questions about some of the core technologies that are the foundation of SkySpark. This document provides insight into those technologies and the rationale for their selection and/or development.

Fantom – The Language SkySpark is Written In

Fantom (https://fantom.org/) is an open source developed, portable language that runs on the JVM and in modern web browsers. It features a familiar Java-like syntax, static (and dynamic) typing, elegant system library, closures, immutability, actor concurrency, and much more. It's design constructs and features make it highly suited to applications in control, automation and IoT applications for the built environment.

- Fantom provides 100% portability between Java (server) and JavaScript (browser)
- Fantom provides first class function support (partially available in Java 8, but still no true function types)
- Fantom provides true immutable types (still not supported by any mainstream language)
- Fantom provides actor concurrency model
- Future portability to new runtimes such as WASM WebAssembly - https://webassembly.org/
- SkySpark's code base was started in 2008 (twelve years ago) and features like those listed above were considered essential to achieving a data-centric platform for automation applications.

Note: When comparing to other potential development languages it's useful to note that Java 8 was not released until 2014 (four years after SkySpark 1.0 was released)

Folio – The Database at the Core of SkySpark

There are many databases available, each with advantages and disadvantages for different types of applications and data. Developers will find that the features of conventional time-series or relational databases are not adequate to achieve an effective solution for IoT data applications. SkySpark's Folio database is designed specifically to address the unique needs of collecting, storing, transforming and performing analytics on IoT device data. Folio accomplishes this by seamlessly integrating three database design concepts into one database:

1. Semantics, which includes techniques from both document and graph databases, along with a query/inference engine. This ability to capture descriptive information about the data – it's meaning and the relationships between the data and the devices that produce it is fundamental to IoT data applications.

2. Real-time - sensor data updating in microseconds

3. Time-series data historian

Folio employs techniques from ontological data science to store rich semantic data about the built environment. It combines concepts from document and graph databases, along with an inference engine you might find in RDF databases. Folio leverages Project Haystack as the core ontology, although it's easy for customers to extend this model to their systems.

Folio integrates real-time sensor data with microsecond update times so that data from BACnet, Modbus, etc. can be seamlessly blended into the semantic model. Folio's industrial strength process historian enables it to store and process time series data values efficiently. Folio's blending of these design concepts creates a database technology optimized for the efficient storage, transformation and analytics of device data, including real time data.

- Folio is built to store data grids (dicts); utilizing document-oriented database techniques. (Note: SkySpark follows the Haystack standard for dicts)
- Built for efficient graph traversal (by this we mean representing and identifying relationships between data items)
- Ontology inference engine to provide deep semantics
- Fully integrated transient data for microsecond real-time updates
- Fully integrated process historian for time-series data
- Just In Time (JIT) self-tuning for automatic indexing
- Replication designed specifically for the needs of IoT applications
- Highly efficient data compression – saves disk space, and cost

The following section provides additional information on **Folio**, its design and performance.

What is a Time Series Database?

One of the pivotal features of SkySpark is our time series database, sometimes called a process historian or simply a historian. Many different people are curious about what a time series database is and why we developed our own database technology. In this article I'm going to dive into some of the key design patterns used by the SkySpark historian and why it provides such high-speed performance.

Anything that stores data keyed by timestamps could be called a time series database. In SkySpark we are specifically talking about sensor data which is typically timestamp samples of an analog or digital sensor, setpoint, or command. For example, if we have a sensor sampling a zone temperature every 15min, then the data stream would look something like:

```
2011-07-20T12:00-04:00 New_York  72.1°F
2011-07-20T12:15-04:00 New_York  72.3°F
2011-07-20T12:30-04:00 New_York  72.3°F
2011-07-20T12:45-04:00 New_York  72.4°F
2011-07-20T13:00-04:00 New_York  72.2°F
```

Characteristics of a Time Series Database

There are some characteristics of time series sensor data which are important to observe:

- High resolution data can yield an immense volume of timestamp/value pairs. If we store minutely data, we have 525,600 samples per year. 100,000 points with two years of minutely data is 105 billion samples! Although that isn't actually an extremely large system, a traditional database with 100 billion rows would be considered a really big database. So efficient storage and access these types of data volume is a top priority
- Time series data is virtually almost always "append only". Once data is stored, we rarely change it, but we are always adding newly acquired timestamp samples
- Time series data is always accessed by a time range: we want to query data by year, month, week, day, date range, etc.
- Time series data queries must always be sorted: we almost always want to analyze or view data in temporal order
- We often wish to work with "rolled up" aggregations of the data; for example, if working with years' worth of 15min energy data we might only care about total daily or monthly energy usage
- If working with data which spans time zones, then proper handling of time zones is critical. Most analysis of time series data is done in the context of local time (for example comparing energy/equipment use against occupancy times)

Problems with Relational Databases When Applied to Sensor and Device Time-Series Data

A common technique is to store time series data in an off the shelf RDBMS like MySql, MS-SQL, etc. But the problem with these general-purpose databases is they make trade-offs which are extremely un-optimized for the characteristics described above. They are designed for random access updates versus a stream of append only data. And without an explicit index on the tables, a RDBMS makes no guarantees about indexing a time range or reading the data in sorted order.

Consider what happens under the covers when you query a relational database for a single week of data in a table which stores several years of data. Without an index the entire data set has be read off disk, the predicate tested to see if each timestamp matches the filter, and then the data has to be sorted in memory.

If you setup indexing on your tables, then the size of your data on disk explodes! Most relational databases use a general-purpose b-tree structure to index a table. If you have a table with billions of rows, the overhead to maintain the index can quickly add up. For this type of data, the index can exceed the disk size of the data itself.

The other major problem with an RDBMS is that they are designed to query the data in one process, then move all the data over a network connection to another process for computation. If you want to compute monthly roll-ups for years' worth of data, that can be a huge amount of data to pass over the network before computation can even begin. So, let's look at the techniques SkySpark uses to optimize this problem:

Storage

In SkySpark we store data to disk so that it is always indexed by timestamp and always sorted by timestamp. The way we store data means we can index the data by timestamp with virtually no additional indexing overhead. Furthermore, we can utilize the fact that we store samples to disk in temporal order to our advantage by applying a bunch of compression techniques. A typical RDBMS without any indexing is probably going to require at least 12 bytes to store a timestamp and 32-bit floating value. If you add overhead for indexing, then this size might double or triple. In SkySpark this same data is stored with an average of 12 bits (1.5 bytes). We've seen customers switch from a relational database to SkySpark and achieve an order of magnitude in disk space savings. Efficient storage of each timestamp/value sample really matters when we are talking about billions of samples.

Read Performance

When you query a specific time range in SkySpark we know a) where to begin reading from disk and b) that we are reading sorted data straight from disk. And because we have compressed data we are often reading fewer bytes from disk. Since disk access is orders of magnitude slower than RAM access, the number of blocks read from disk before you can start crunching your data is the #1 issue in how well your database performs. In SkySpark we don't spend any time reading index information from disk and we don't have to perform any sorting in memory.
So how fast can we read data? Even on my run-of-the-mill machine and disk drive I consistently get benchmarks for querying 800,000 samples/sec. To crunch a year's worth of 15 minutely data is only 30ms!

Write Performance

SkySpark takes advantage of the fact that most writes to a time-series database are at the end of the time stream. We constantly collect new sensor data and might be writing new data for 100,000 points every minute. SkySpark optimizes for this case and avoids the costly techniques of journaling, log files, and b-tree updates as required by a general-purpose RDBMS.

Note that since the data must be sorted to disk, there can be a performance impact if attempting to write data with timestamps which occur interleaved with data already stored to disk (SkySpark provides a warning when this happens).

Under the covers we use Fantom's actor concurrency model to effectively utilize all the cores of the microprocessor to pre-sort, coalesce, and sanitize the data. The overall result is that SkySpark's historian can typically write data to disk far faster than a general-purpose database.

So how fast can we write? On my run-of-the-mill machine, benchmarking yields write rates of 250,000 samples/sec. If working with 15min data, that equates to writing seven years of data every second.

Colocation of Computation

The architecture of SkySpark is designed around the idea that the database engine and computation engine must be co-located in the same OS process. The key observation is that often we wish to crunch large volumes of time series data to compute small answers. Examples: find the total energy consumption for each month, find the day where we had our peak demand in 2010, find periods of time where a sensor exceeded a given limit.

In a relational database, the SQL language is used to query the data. But SQL is not a general-purpose language for analyzing and transforming data. So, one must use SQL to query the raw data, move it over the network to another process, and then crunch the data using a programming environment like Java, C#, etc. The problem with this approach is that you might have to query millions of rows of data over the network just to compute a single number. To work around the inevitable performance problems, developers try to pre-compute these "rollup" versions of the data. But this is a stop-gap solution that doesn't work for ad hoc general analytics because there is an infinite number of ways, we might want to crunch the raw data.

In SkySpark our database and the Axon computation engine are bundled together in the same server. We've designed Axon to be an expressive query language, but also a full-fledged general-purpose programming language. Let's consider a simple example where we want to query the monthly consumption of energy data for the entire year of 2010 using raw 15 minutely data:

```
readAll(kwh).hisRead(2010).hisRollup(sum, 1mo)
```

This expression is evaluated by Axon as follows: read all the records with the kwh tag, pipe that to the hisRead function to read all the raw 15 minutely data for the entire year of 2010, and then pipe that to the hisRollup function to compute the monthly interval sum of the raw data. This entire calculation is executed as the data is read off disk. There is no buffering of raw data in memory, nor are we required to pass any data over a network connection. The design for pipe-lining the computation as we read off disk is a key enabler for SkySpark's high-performance analytics of time series data.

We hope this overview helps explain how the Folio database addresses the challenges of storing and processing analytics on device data.

Axon – The Scripting Language Optimized for Analytics of Sensor and Device Data

The unique characteristics and requirements of working with device data extend to the process of defining and coding analytic rules and algorithms. SkySpark implements a language called Axon that is optimized to address these needs. The Axon language underlies all of SkySpark – it is the language used to define analytic rules and algorithms, it is used to create automated "jobs" that acquire and process and transform data from external sources and to query the Folio database to create reports and views.

Key characteristics of Axon include:

- Extremely simple language (grammar fits on one page). This enables developers to quickly gain competence
- (Mostly) pure functional language – functional languages are the most effective solution for applications that focus on data transformations, which is the core function of analytics. Examples: Transform raw data to patterns (Sparks), transform raw data into KPIs, rollups and other calculations.
- The Axon language designed to manipulate data grids (see Folio section of this document). To do this it shares the same literal syntax as Haystack. For example, Haystack data graphs are valid in Axon which is the same data type/model for representing data in Folio. Example: Numbers stored in Folio always have units as part of the data, and functions to query/transform numbers always work with the units without requiring additional coding by programmers.
- True immutability for all data values – this is a key benefit when working with device data, meaning it never changes the original data as it calculates results. It goes hand in hand with effective data transformation. This makes Axon dramatically easier to debug and understand program logic. It also makes it easier to do parallel processing with multiple threads/cores.
- Requires sandboxed environment for security – Everything SkySpark does in the UI and analytic rules are Axon calls. Because Axon is sandboxed users cannot perform system level calls, and cannot penetrate the security boundary of the Axon engine – For example, they can't access passwords, etc.
- Integrated components syntax. As a custom language Axon provides a first-class syntax for Haystack filters and also for building functional components.
- Querying data inside Axon uses Axon itself versus requiring a separate syntax. And with a built-in component syntax, developers can build packaged rules with predefined bindings and tuning parameters.

We hope this document helps enhance the understanding of SkySpark's foundational technologies. Contact us to learn more at info@skyfoundry.com

ABOUT SKYFOUNDRY

SkyFoundry's mission is to provide software solutions for the age of "the Internet of things". Areas of focus include:

- Building automation and facility management
- Energy management, utility data analytics
- Remote device and equipment monitoring
- Asset management

SkyFoundry products help customers derive value from their investments in smart systems. Contact us to learn more.

https://skyfoundry.com/

info@skyfoundry.com

SkyFoundry