Friends don’t let Friends use the Lambda Architecture

John Hugg - VoltDB Inc.

Recently, there have been a number of systems developed to tackle the intersection of streaming and big data. Nathan Marz has championed the Lambda Architecture as a recipe for combining speed and batch processing to tackle these problems. Today, it’s hard to go to a trendy big data conference without hearing a talk about how a complex, interdependent Lambda system solved a problem in heroic fashion with only a modicum of war scars and hefty AWS bill.

In this talk, we will prove, by counterexample, that the Lambda Architecture is bad and should feel bad. We will examine an recent, popular and non-fictional Lambda conference talk. We will then show how the same problem can be solved using VoltDB in a tiny fraction of the code, with many fewer nodes and with drastically reduced operational complexity. Finally we will argue the results are generally generalizable, i.e. not specific to the one example shown.

What is the Lambda Architecture

Today, high throughput data is often managed in a batch workload. Data is loaded, then processed or queried as a discrete step. The problem with this approach is the latency to understanding is often minutes or hours. The alternative, systems that process streams of data, can provide real time understanding, but are lacking in other ways, primarily robustness.

The Lambda Architecture proposes that both speed/streaming and batch workloads be run in parallel on the same incoming data. The speed layer can achieve faster responses, while the batch layer can be more robust and serve as the system of record for historical data. Queries can be made against both layers, and responses can be merged to provide one view.

An Exemplar Lambda Problem

Ed Solovey of Twitter (formerly Crashlytics) has given several talks on their use of the Lambda Architecture for their Crashlytics service, including a 20 minute presentation from October at the Boston Facebook @Scale Conference (http://youtu.be/56wy_mGEnzQ).

To summarize his problem as presented, every time an end-user uses a smartphone app, a message is sent to Crashlytics with an app identifier and a unique device id. This happens 800,000 times per-second over thousands of apps. App developers pay Crashlytics to tell them how many unique users have used their app each day, with per-day history available for some amount of time going back.

To solve this problem, Crashlytics has chosen the Lambda Architecture. In the speed layer, they have enlisted Kafka for ingestion, Storm for processing, Cassandra for state and Zookeeper for distributed agreement. In the batch layer, tuples are loaded in batches into S3, then are processed with Cascading and Amazon Elastic Map-Reduce.

Note that counting unique devices when there are potentially billions of values is not space efficient and requires at least a hefty hash/tree lookup. Thus Crashlytics has chosen to use a cardinality estimation algorithm, HyperLogLog, in the speed layer. HyperLogLog requires only a few kilobytes per app and can be accurate to a few percent or less. Since the Storm framework exposes a flexible Java interface to developers, integrating a HyperLogLog implementation is reasonably easy to do.

The upside to the Crashlytics solution are laid out by Ed in his presentation well. First, it meets their high performance needs, a requirement that would eliminate many solutions. Second, by using a batch and a speed layer that perform interchangeable computations using different software, there is a built-in cross-checking to ensure valid answers.

The acknowledged downside is complexity. We will assume Zookeeper, Kafka, Storm and Cassandra all require at least three nodes each to run with reasonable safety, as this is a bit of a magic number for distributed systems. Thus the speed layer alone requires twelve machines before it has processed anything. To create just the speed layer, four systems must be interconnected. To use the system, all four systems must be monitored and any problems must be traced to the faulty system or to an interaction between systems. Ed rightly calls out that the speed layer as built is fragile, as failure recovery is complex and error-prone.

A Concrete Alternative

At VoltDB, we've seen a number of Lambda Architecture presentations where we theorized that the presenter could have saved time and money by replacing his or her stack with a VoltDB solution. The Crashlytics use case is well defined, but still not trivial to accomplish with most processing tools. Thus it provided the perfect opportunity to directly compare a VoltDB-based solution with a Lambda solution.

First, we built a data generator, often the hardest part of any project. Next we found an existing HyperLogLog implementation in Java on Github with a permissive license and whipped up a stored procedure that accepts AppId/DeviceId tuples and returns estimated counts. The bulk of the logic is about 30 new lines of code and it can easily scale to Crashlytics goal of 800k ops/sec on a commodity three node cluster, which is at least a factor of four improvement on the Crashlytics speed layer. It took one developer about two days to build the app and the data generator.

The Advantages of ACID Transactions and Integrated State in Stream Processing

For each smartphone app, the VoltDB solution stores the HyperLogLog bits in a blob, but also the current estimate in integral form. ACID transactions are used to update the HyperLogLog bits, compute a new estimate and store it in the row durably and constantly. This means the state can be queried with SQL directly without needing HyperLogLog code to process the rows, opening the schema up to standard reporting tools.

Additionally, an extension of the solution was developed that kept exact counts up to a fixed number, say 1000, then switched to estimated cardinality above that. This handoff between exact and non-exact counts leveraged ACID transactions to simplify the code greatly.

But what about batch? The first version of our app was built to replace the speed layer, but the batch/historical layer was also easy to add. Daily history is essentially a 64-bit long per smartphone app, per day, allowing a distributed in memory system like VoltDB to store years of history. The key to actually storing the correct value is using ACID transactions to detect a daily rollover exactly once and change data in multiple tables transactionally and durably.

Finally all of these enhancements to the core problem can be localized to the stored procedure that processes the tuple. No access to HyperLogLog code is required in client or supporting code. Whether estimates or exact values are used can also be toggled by changing the procedure code only. One of the oft criticisms levied at stored procedures is that they separate business logic between the client and the server. In Lambda architectures where processing is done at multiple sites and data lives in many stores, insulating the core processing code to a single stored procedure and allowing clients to use standard SQL has tremendous value.