AirbnbEng Follow
Creative engineers and data scientists building a world where you can belong anywhere.
http://airbnb.io
Feb 23, 2016 · 10 min read

# Data Infrastructure at Airbnb

By James Mayfield, Krishna Puttaswamy, Swaroop Jagadish, and Kevin Long

. . .

## Part 1: Philosophy Behind our Data Infrastructure

At Airbnb we promote a data informed culture and use data as a key input for making decisions. Tracking metrics, validating hypotheses through experimentation, building machine learning models, and mining for deep business insights are all critical to our moving fast and moving smart.

After many evolutionary steps, we now feel that our data infrastructure stack is stable, reliable, and scalable so it seemed like a good opportunity to share our experiences back with the community. Over the next few weeks, we will release a series of blog posts that highlight parts of our distributed architecture and our tool suite. Because open source contributors have provided many of the foundational systems

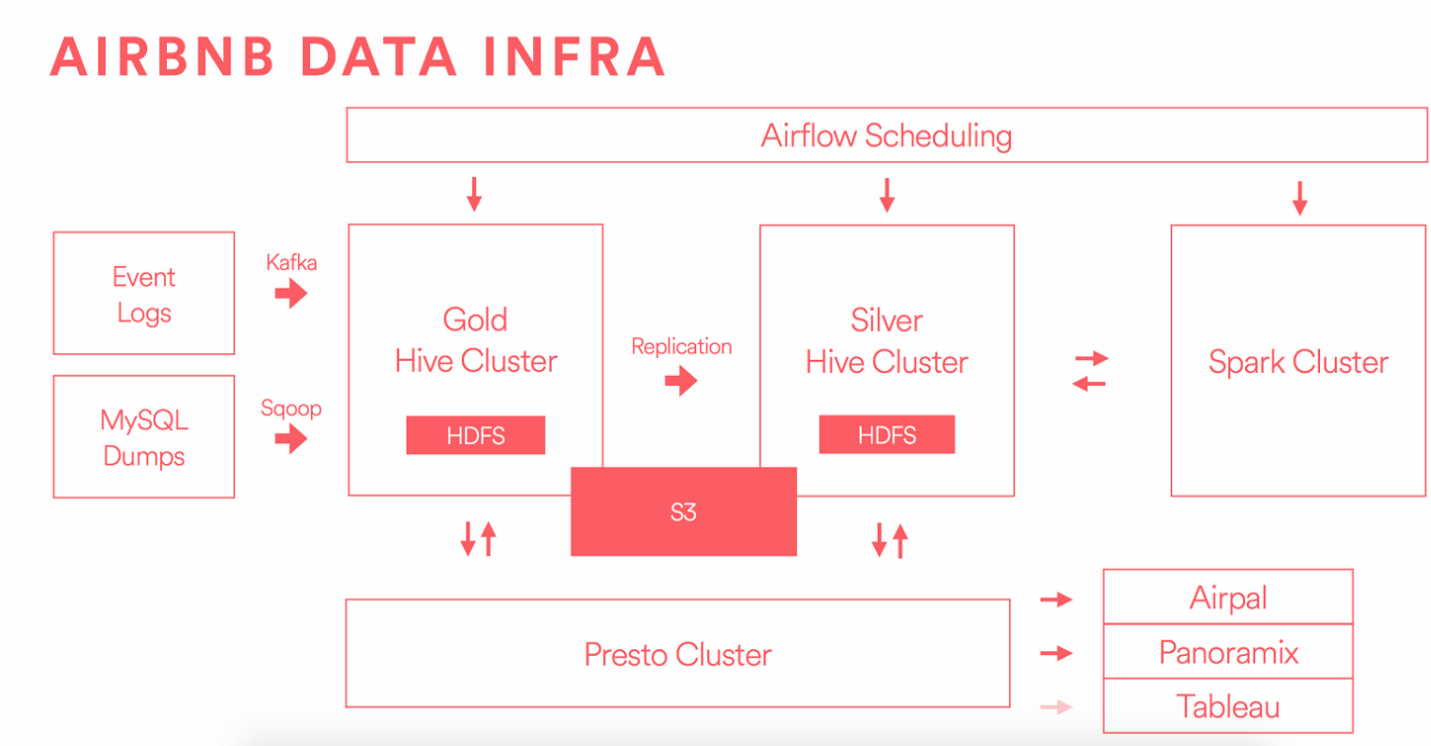
we use each day, it makes us more than happy to contribute back not only useful projects in public GitHub repos, but also to describe the things we've learned along the way.

Some informal philosophies that have come about during our time working on data infrastructure:

- **Look to the open source world:** there are a lot of good resources for data infrastructure in the open source community and we try to adopt those systems. Furthermore, if we build something useful ourselves and it is feasible to give it back to the community, we reciprocate.
- **Prefer standard components and methods**: There are times when it makes sense to invent a completely new piece of infrastructure, but often times this is not a good use of resources. Having intuition about when to build a unique solution and when to adopt an existing solution is important, and that intuition must properly account for the hidden costs of maintenance and support.
- **Make sure it can scale:** we have found that data doesn't grow linearly with the business, but grows superlinearly as technical employees begin building new products and logging new activities on top of the growth of the business.
- **Solve real problems by listening to your colleagues:** empathizing with the data users around the company is an important part of informing our roadmap. In adherence with Henry Ford's mantra, we must balance making faster horses vs. building the automobile—but listen to your customers first.
- **Leave some headroom:** we oversubscribe resources to our clusters in order to foster a culture of unbounded exploration. It is easy for infrastructure teams to get wrapped up in the excitement of maximizing resources too early, but our hypothesis is that a single new business opportunity found in the warehouse will more than offset those extra machines.

. . .

## Part 2: Infrastructure Overview




Here is a simplified diagram showing the major components of our infrastructure stack.

Source data comes into our system from two main channels: instrumentation in source code that sends events through Kafka and production database dumps that are taken using AWS point-in-time-restores on RDS, then delivered through Sqoop.

This source data containing user activity event data and dimensional snapshots is then sent into the Gold cluster where we store the data and begin running our extraction, transform and load jobs. In this step, we apply business logic, make summary tables, and perform data quality checks.

In the diagram above, there are two separate clusters for “Gold” and “Silver” which we will describe in more detail later in this post. The high level reason for the separation is to guarantee isolation of compute and storage resources and it provides disaster recovery assurances if there ever was to be an outage. This architecture provides a Gold environment where the most critical jobs can run with strict guarantees and service level agreements, free from any interference caused by resource intensive ad hoc queries. We treat the Silver cluster as being a

production environment as well, but relax the guarantees made and tolerate bursts of resource intensive queries.

Note that by having two clusters we gain the power of isolation, but it comes at the cost of managing large volume data replication and maintaining synchronicity between dynamic systems. Gold is our source of truth and we copy each bit of data from Gold down to Silver. Data generated on the Silver cluster is not copied back to Gold, and so you can think of this as a one way replication scheme that leaves Silver cluster as a superset of everything. Because much of our analysis and reporting happens from the Silver cluster, it is critical that when new data lands in Gold we replicate it on Silver as soon as possible to keep all user jobs running without delay. Even more critically, if we update a preexisting piece of data on the Gold cluster, we must be alerted to that update and propagate the change down to Silver as well. This replication optimization problem didn't have a good solution in the open source, so we built a new set of tools that we will describe in greater detail in an upcoming post.

We have made a big effort to treat HDFS, and more accurately to treat Hive managed tables, as our central source and sink for data. The quality and sanity of the warehouse depend on data being immutable and all derivations of data being reproducible—using partitioned Hive tables is really important for this goal. Furthermore, we discourage the proliferation of different data systems and do not want to maintain separate infrastructure that sits between our source data and our end user reporting. In our experience, these intermediate systems obfuscate sources of truth, increase burden for ETL management, and make it difficult to trace the lineage of a metric figure on a dashboard all the way back to the source data from which it was derived. We do not run Oracle, Teradata, Vertica, Redshift, etc. and instead use Presto for almost all ad hoc queries on Hive managed tables. We are hoping to link Presto directly to our Tableau installation as well in the near future.

A few other things to note in the diagram include Airpal, a web based query execution tool backed by Presto that we built and open sourced. It is our main interface for users to run ad hoc SQL queries against the warehouse and more than 1/3 of all employees have run queries using the tool. Scheduling of jobs happens through Airflow, a platform to programmatically author, schedule, and monitor data pipelines that can run jobs on Hive, Presto, Spark, MySQL, and more—note that we

logically share Airflow across clusters, but physically the Airflow jobs run on the appropriate sets of clusters, machines, and workers.. We built Airflow and open sourced it, as well. The Spark cluster is another processing tool favored heavily by engineers and data scientists working on machine learning, and is useful for stream processing. You can check out more of our ML efforts in the Aerosolve post. S3 is a separate storage system where we can retire data from HDFS for cheap long term storage. Hive managed tables can have their storage altered and pointed to S3 files in order to maintain easy access patterns and metadata management.

. . .

## Part 3: Detailed Look at Our Hadoop Cluster Evolution

This year we undertook a significant migration to go from a poorly architected set of clusters called "Pinky and Brain" and onto the "Gold and Silver" system described above. To set some context for scale, two years ago we moved from Amazon EMR onto a set of EC2 instances running HDFS with 300 terabytes of data. Today, we have two separate HDFS clusters with 11 petabytes of data and we also store multiple petabytes of data in S3 on top of that. With that background, here were the major problem areas and what we did to resolve them:

**A) Running a Unique Hadoop on Mesos Architecture**

Some early Airbnb engineers had a keen interest in a piece of infrastructure called Mesos, which sets out to deploy a single configuration across many servers. We built a single cluster of c3.8xlarge machines in AWS each backed by 3TB of EBS and ran all Hadoop, Hive, Presto, Chronos, and Marathon on Mesos.

To be clear, many companies use Mesos to great effect and implement novel solutions to administer large sets of machines running important infrastructure. But, our small team decided running a more standard, ubiquitous deployment would reduce the time we spent on operations and debugging.

Hadoop on Mesos issues:

- Very little visibility into jobs running and log files

- Very little visibility into the cluster health
- Hadoop on Mesos could only run MR1
- Performance problems caused by task tracker contention
- Cluster under utilization
- High operational load and difficulty reasoning about the system
- Lack of integration with Kerberos for security

***Resolution:*** the answer was simply to move to a "standard" stack. We were happy to learn from the hundreds, or potentially thousands, of other companies that operate large clusters and not try to invent a novel solution to a problem that is not ours to solve.

**B) Remote Reads and Writes**

By storing all our HDFS data in mounted EBS (elastic block storage) volumes, we were sending lots of data over the public Amazon EC2 network in order to run queries. Hadoop was built for commodity hardware and expects local reads and writes on spinning disks, so this was a design mismatch.

Further to the remote nature of reads and writes, we had incorrectly chosen to split our data storage across three separate availability zones, within a single region, in AWS. Furthermore, each availability zone was designated as its own "rack" so the 3 replicas were stored on different racks, therefore remote reads and writes were happening constantly. This was again a design flaw that led to slow data transfers and remote copies happening anytime a machine was lost or a block corrupted.

***Resolution:*** having dedicated instances using local storage, and running in a single availability zone without EBS fixed these issues.

**C) Heterogeneous Workload on Homogeneous Machines**

Looking at our workload, we found that there were distinct requirements for our architectural components. Our Hive/Hadoop/HDFS machines required tons of storage, but didn't need much RAM or CPU. Presto and Spark were thirsty for memory and processing power, but didn't need much storage. Running c3.8xlarge

instances backed by 3TB EBS was proving to be very expensive as storage was our limiting factor.

***Resolution:*** Once we migrated off the Mesos architecture, we were able to choose different machine types to run the various clusters, for example using r3.8xlarge instances to run Spark. Amazon happened to be releasing their new generation of "D-series" instances at the time we were evaluating a shift, which made the transition even more desirable from a cost perspective. Moving from 3TB of remote storage per node on the c3.8xlarge machines to to 48TB of local storage on d2.8xlarge machines was very appealing and will save us millions of dollars over the next three years.

**D) HDFS Federation**

We had been running a federated HDFS cluster with Pinky and Brain where the data was held in shared physical block pools but the sets of mappers and reducers were isolated per logical cluster. This led to an awesome end user experience where any piece of data could be accessed by either Pinky queries or Brain queries, but unfortunately we found that federation was not widely supported and was considered experimental and unreliable by experts.

***Resolution:*** moving to a fully distinct set of HDFS nodes, and not running federation, gave us the true isolation of clusters at the machine level, which also provided better disaster recovery coverage.

**E) System Monitoring was Burdensome**

One of the most serious problems of having a unique infrastructure system was being forced to create custom monitoring and alerting for the cluster. Hadoop, Hive, and HDFS are complicated systems, prone to bugs across their many knobs and dials. Trying to anticipate all failure states and set reasonable pager thresholds proved quite challenging and it also felt that we were solving a solved problem.

***Resolution:*** we signed a support contract with Cloudera to gain from their expertise in architecting and operating these large systems, and most importantly to reduce our maintenance burden by using the Cloudera Manager tool. Linking it into our Chef recipes has greatly reduced our monitoring and alerting burden and we are happy to

report that we spend very little time on system maintenance and alerting.

**Conclusion**

After evaluating all the errors and inefficiencies with our old cluster setup, we set out to systematically resolve these problems. It was a long process to migrate petabytes of data and hundreds of user jobs without disrupting service for our colleagues; we will author a new post on that subject alone and release some of our tooling to the open source community.

Now that the migration is complete, we have drastically reduced the number of incidents and outages in our platform. It is not difficult to imagine the number of bugs and the number of issues we dealt with while running our custom stack on immature platforms, but the system is now well-worn and largely stable. The other benefit is that when we hire new engineers to join the team, onboarding is streamlined because the systems are familiar to what other companies have adopted.

Lastly, because we had the chance to architect things fresh in the new Gold and Silver setup, we had an opportunity to spin up all new instances and add IAM roles to manage security in a sensible fashion. This has meant a much more sane access control layer on top of the cluster, and integrates with how we manage all our machines.

The fact that we were able to dramatically cut costs and at the same time increase performance was awesome. Here are a few stats:

- Disk read/write improved from 70–150MB/sec to 400+ MB/sec
- Hive jobs ~2X faster in CPU and wall clock time
- Network can be fully saturated on our machines, when applicable
- Read throughput is ~3X better
- Write throughput is ~2X better
- Cost is reduced by 70%

. . .

## Part 4: Thanks and Praises

Big thanks to the team of engineers who built the original foundation of data infrastructure at Airbnb and to the folks who have been steadily working to improve and stabilize the systems. I was happy to write this blog post but the real credit goes to Nick Barrow-Williams, Greg Brandt, Dan Davydov, Wensheng Hua, Swaroop Jagadish, Andy Kramolisch, Kevin Long, Jingwei Lu, Will Moss, Krishna Puttaswamy, Liyin Tang, Paul Yang, Hongbo Zeng, and Jason Zhang.

. . .

**Check out all of our open source projects over at airbnb.io and follow us on Twitter: @AirbnbEng + @AirbnbData**