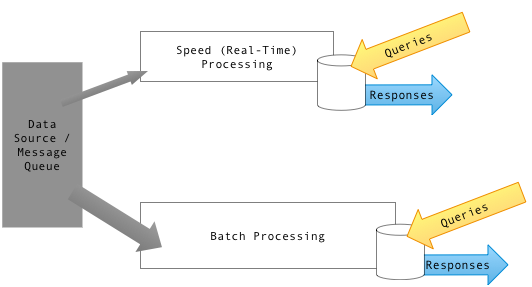
# **Lambda architecture**



Flow of data through the processing and serving layers of a generic lambda architecture

* Generic, scalable and fault-tolerant.

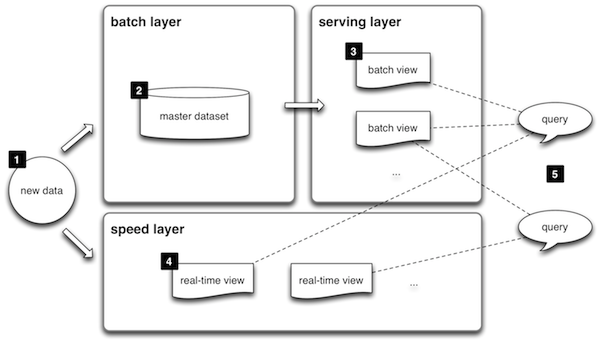
**Lambda architecture** is a [**data-processing**](https://en.wikipedia.org/wiki/Data_processing)**architecture** designed to handle massive quantities of data by taking advantage of both [**batch**](https://en.wikipedia.org/wiki/Batch_processing)**- and**[**stream-processing**](https://en.wikipedia.org/wiki/Stream_processing) methods. This approach to architecture attempts to **balance**[**latency**](https://en.wikipedia.org/wiki/Latency_(engineering))**,**[**throughput**](https://en.wikipedia.org/wiki/Throughput)**, and**[**fault-tolerance**](https://en.wikipedia.org/wiki/Fault-tolerance), both against hardware failures and human mistakes, by using batch processing to provide comprehensive and accurate views of batch data, while simultaneously using **real-time stream** processing to provide views of online data.

Able to serve a wide range of workloads and use cases, and in which **low-latency reads and updates** are required. The resulting system should be **linearly scalable**, and it should scale out rather than up.

Lambda architecture depends on a data model with an append-only, immutable data source that serves as a system of record. It is intended for ingesting and processing timestamped events that are appended to existing events rather than overwriting them. State is determined from the natural time-based ordering of the data.

Motivation for building systems with the lambda architecture as:

* The need for a robust system that is fault-tolerant, both against hardware failures and human mistakes.
* To serve a wide range of workloads and use cases, in which low-latency reads and updates are required. Related to this point, the system should support ad-hoc queries.
* The system should be linearly scalable, and it should scale out rather than up, meaning that throwing more machines at the problem will do the job.
* The system should be extensible so that features can be added easily, and it should be easily debuggable and require minimal maintenance.

366666666666666666666666666666666666666666666666666.ñ

1. All **data** entering the system is dispatched to both the batch layer and the speed layer for processing.
2. The **batch layer** has two functions: (i) managing the master dataset (an immutable, append-only set of raw data), and (ii) to pre-compute the batch views.
3. The **serving layer** indexes the batch views so that they can be queried in low-latency, ad-hoc way.
4. The **speed layer** compensates for the high latency of updates to the serving layer and deals with recent data only.
5. Any incoming **query** can be answered by merging results from batch views and real-time views.

## **Overview**

Lambda architecture describes a system consisting of three layers: batch processing, speed (or real-time) processing, and a serving layer for responding to queries. The processing layers ingest from an immutable master copy of the entire data set.

### Batch layer

The batch layer precomputes results using a distributed processing system that can handle very large quantities of data. The batch layer aims at perfect accuracy by being able to process *all* available data when generating views. This means it can fix any errors by recomputing based on the complete data set, then updating existing views. Output is typically stored in a read-only database, with updates completely replacing existing precomputed views.

[Apache Hadoop](https://en.wikipedia.org/wiki/Hadoop) is the de facto standard batch-processing system used in most high-throughput architectures.

## **Processing Frameworks**

| **Technology** | **Does it fit** | **Maturity** | **Ease of use** | **Language** | **Platforms** | **Comments** |
| --- | --- | --- | --- | --- | --- | --- |
| [Hadoop MapReduce](http://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapreduce/package-summary.html) | ★★★ | ★★★ | ★ | Java | Hadoop | Very low-level, not re-usable |
| [Spark](https://spark.apache.org/docs/latest/) | ★★★ | ★★ | ★★★ | Scala, Java, Python | Spark | In-memory |
| [Hive](http://hive.apache.org/) | ★★★ | ★★★ | ★★★ | HiveQL, Java | Hadoop | Support planned for Tez |
| [Spark SQL](https://spark.apache.org/docs/latest/sql-programming-guide.html) | ★★★ | ★ | ★★ | SQL, Scala, Java, Python | Spark | Successor of Shark |
| [Pig](http://pig.apache.org/) | ★★★ | ★★★ | ★★★ | Pig Latin, Java | Hadoop | Support planned Tez |
| [Spork](https://github.com/mateiz/spork) | ★★★ | ★ | ★★★ | Pig Latin, Java | Spark |  |
| Cascading/Scalding | ★★★ | ★★ | ★★ | Java, Scala | Hadoop |  |
| Cascalog | ★★★ | ★ | ★ | Clojure | Hadoop |  |
| Crunch/SCrunch | ★★★ | ★★ | ★ | Java, Scala | Hadoop | Support planned for Spark and Tez |
| Pangool | ★★★ | ★ | ★ | Java | Hadoop |  |

### Speed layer

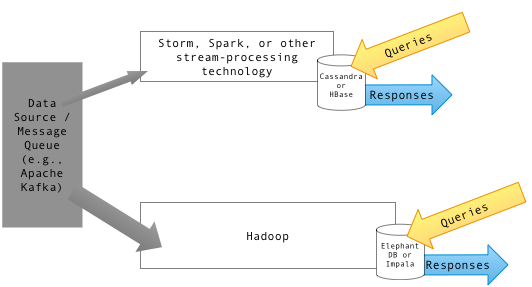


Diagram showing the flow of data through the processing and serving layers of lambda architecture. Example named components are shown.

The speed layer processes data streams in real time and without the requirements of fix-ups or completeness. This layer sacrifices throughput as it aims to minimize latency by providing real-time views into the most recent data. Essentially, the speed layer is responsible for filling the "gap" caused by the batch layer's lag in providing views based on the most recent data. This layer's views may not be as accurate or complete as the ones eventually produced by the batch layer, but they are available almost immediately after data is received, and can be replaced when the batch layer's views for the same data become available.

Stream-processing technologies typically used in this layer include [Apache Storm](https://en.wikipedia.org/wiki/Storm_(event_processor)), [SQLstream](https://en.wikipedia.org/wiki/Sqlstream" \o "Sqlstream) and [Apache Spark](https://en.wikipedia.org/wiki/Apache_Spark). Output is typically stored on fast NoSQL databases.

## **Stream Processing Frameworks**

| **Technology** | **Does it fit** | **Maturity** | **Ease of use** | **Language** | **Comments** |
| --- | --- | --- | --- | --- | --- |
| [Apache Storm](http://storm-project.net/) | ★★★ | ★★★ | ★★ | Clojure | originates from Twitter |
| [Apache Spark Streaming](https://spark.apache.org/streaming/) | ★★★ | ★★ | ★★★ | Scala/Java/Python | originates from AMPLab |
| [Apache Samza](http://samza.incubator.apache.org/) | ★★★ | ★★ | ★ | Scala/Java | originates from LinkedIn |
| [Apache S4](http://incubator.apache.org/s4/) | ★★★ | ★ | ★ | Java | originates from Yahoo! |
| [Spring XD](http://projects.spring.io/spring-xd/) | ★★★ | ★★ | ★★★ | Java | originates from Pivotal |

## **Cloud-based (XaaS) Offerings**

| **Technology** | **Does it fit** | **Maturity** | **Ease of use** | **API** | **Comments** |
| --- | --- | --- | --- | --- | --- |
| AWS [Kinesis](http://aws.amazon.com/kinesis/) | ★★★ | ★★ | ★★ | Java | introduced in 11/2013 |
| Google [Cloud Dataflow](http://googlecloudplatform.blogspot.com/2014/06/sneak-peek-google-cloud-dataflow-a-cloud-native-data-processing-service.html) | ★★ | - | ? | Java | introduced in 06/2014, not yet available |

### Serving layer

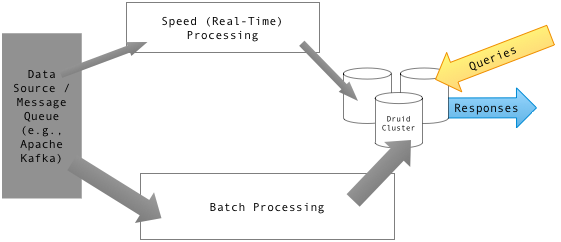


Diagram showing a lambda architecture with a Druid data store.

Output from the batch and speed layers are stored in the serving layer, which responds to ad-hoc queries by returning precomputed views or building views from the processed data.

Examples of technologies used in the serving layer include [Druid](https://en.wikipedia.org/wiki/Druid_(open-source_data_store)), which provides a single cluster to handle output from both layers. Dedicated stores used in the serving layer include [Apache Cassandra](https://en.wikipedia.org/wiki/Apache_Cassandra), [Apache HBase](https://en.wikipedia.org/wiki/Apache_HBase), [MongoDB](https://en.wikipedia.org/wiki/MongoDB) or [Elasticsearch](https://en.wikipedia.org/wiki/Elasticsearch) for speed-layer output, and [Elephant DB](https://github.com/nathanmarz/elephantdb), [Cloudera Impala](https://en.wikipedia.org/wiki/Cloudera_Impala) or [Apache Hive](https://en.wikipedia.org/wiki/Apache_Hive) for batch-layer output.

## **Merge/Low-Latency Databases**

| **Technology** | **Does it fit** | **Maturity** | **Ease of use** | **API Language** | **Comments** |
| --- | --- | --- | --- | --- | --- |
| ElephantDB | ★★★ | ★ | ★ | Clojure |  |
| SploutSQL | ★★★ | ★ | ★★ | Java |  |
| Voldemort (with a ReadOnly backend) | ★★ | ★★ | ★★ | Java |  |
| HBase (bulk loading) | ★★ | ★★ | ★★ | Java |  |
| [Druid](http://druid.io/) | ★★★ | ★★ | ★ | Java | originates from Metamarkets |

## **Optimizations**

To optimize the data set and improve query efficiency, various rollup and aggregation techniques are executed on raw data, while estimation techniques are employed to further reduce computation costs. And while expensive full recomputation is required for fault tolerance, incremental computation algorithms may be selectively added to increase efficiency, and techniques such as *partial computation* and resource-usage optimizations can effectively help lower latency.

## **Criticism**

Criticism of lambda architecture has focused on its inherent complexity and its limiting influence. The batch and streaming sides each require a different code base that must be maintained and kept in sync so that processed data produces the same result from both paths. Yet attempting to abstract the code bases into a single framework puts many of the specialized tools in the batch and real-time ecosystems out of reach.[[11]](https://en.wikipedia.org/wiki/Lambda_architecture#cite_note-11)

In a technical discussion over the merits of employing a pure streaming approach, it was noted that using a flexible streaming framework such as [Apache Samza](https://en.wikipedia.org/wiki/Apache_Samza) could provide some of the same benefits as batch processing without the latency.[[12]](https://en.wikipedia.org/wiki/Lambda_architecture#cite_note-12) Such a streaming framework could allow for collecting and processing arbitrarily large windows of data, accommodate blocking, and handle state.

# Applying the Big Data Lambda Architecture

**A look inside a Hadoop-based project that matches connections in social media by leveraging the highly scalable lambda**

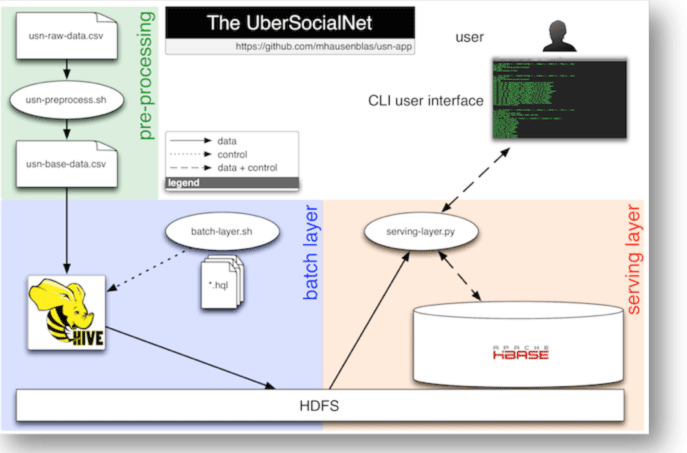
### Scope and Architecture of the Project

In this article, I employ the lambda architecture to implement what I call UberSocialNet (USN). This open-source project enables users to store and query acquaintanceship data. That is, I want to be able to capture whether I happen to know someone from multiple social networks, such as Twitter or LinkedIn, or from real-life circumstances. The aim is to scale out to several billions of users while providing low-latency access to the stored information. To keep the system simple and comprehensible, I limit myself to bulk import of the data (no capabilities to live-stream data from social networks) and provide only a very simple a command-line user interface. The guts, however, use the lambda architecture.

It's easiest to think about USN in terms of two orthogonal phases:

* Build-time, which includes the data pre-processing, generating the master dataset as well as creating the batch views.
* Runtime, in which the data is actually used, primarily via issuing queries against the data space.

The USN app architecture is shown below in Figure 2:

  
**Figure 2: High-level architecture diagram of the USN app.**

The following subsytems and processes, in line with the lambda architecture, are at work in USN:

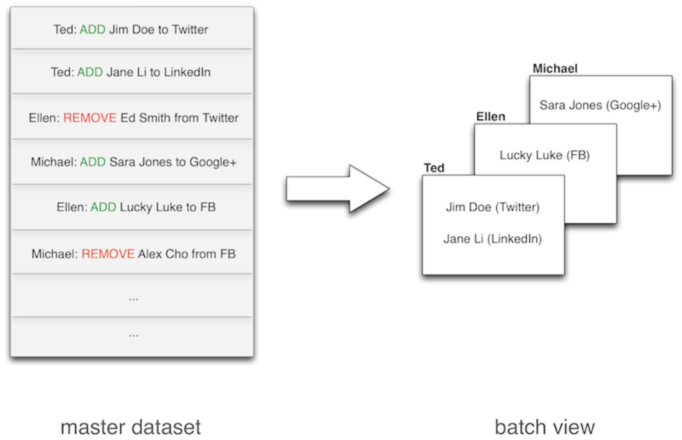
* Data pre-processing. Strictly speaking this can be considered part of the batch layer. It can also be seen as an independent process necessary to bring the data into a shape that is suitable for the master dataset generation.
* The batch layer. Here, a bash shell [script](https://github.com/mhausenblas/usn-app/blob/master/batch-layer/batch-layer.sh) is used to drive a number of [HiveQL](https://cwiki.apache.org/Hive/tutorial.html) queries (see the GitHub repo, in the [batch-layer folder](https://github.com/mhausenblas/usn-app/tree/master/batch-layer)) that are responsible to load the pre-processed input CSV data into HDFS.
* The serving layer. In this layer, we use a Python [script](https://github.com/mhausenblas/usn-app/blob/master/serving-layer.py) that loads the data from HDFS via Hive and inserts it into a HBase table, and hence creating a batch view of the data. This layer also provides query capabilities, necessary in the runtime phase to serve the front-end.
* Command-line front end. The USN app front-end is a bash [shell script](https://github.com/mhausenblas/usn-app/blob/master/usn-ui.sh) interacting with the end-user and providing operations such as listings, lookups, and search.

This is all there is from an architectural point of view. You may have noticed that there is no speed layer in USN, as of now. This is due to the scope I initially introduced above. At the end of this article, I'll revisit this topic.

### The USN App Technology Stack and Data

Recently, Dr. Dobb's discussed [Pydoop: Writing Hadoop Programs in Python](http://www.drdobbs.com/database/pydoop-writing-hadoop-programs-in-python/240156473), which will serve as a gentle introduction into setting up and using Hadoop with Python. I'm going to use a mixture of Python and bash shell scripts to implement the USN. However, I won't rely on the low-level MapReduce API provided by Pydoop, but rather on higher-level libraries that interface with Hive and HBase, which are part of Hadoop. Note that the entire source code, including the test data and all queries as well as the front-end, is available in a [GitHub repository](https://github.com/mhausenblas/usn-app), and it is necessary to follow along with this implementation.

Before I go into the technical details such as the concrete technology stack used, let's have a quick look at the data transformation happening between the batch and the serving layer.

  
**Figure 3: Data transformation from batch to serving layer in the USN app.**

As hinted in Figure 3, the master dataset (left) is a collection of atomic actions: either a user has added someone to their networks or the reverse has taken place, a person has been removed from a network. This form of the data is as raw as it gets in the context of our USN app and can serve as the basis for a variety of views that are able to answer different sorts of queries. For simplicity's sake, I only consider one possible view that is used in the USN app front-end: the "network-friends" view, per user, shown in the right part of Figure 3.

### Raw Input Data

The raw input data is a Comma Separated Value (CSV) file with the following format:

[?](http://www.drdobbs.com/database/applying-the-big-data-lambda-architectur/240162604)

|  |  |
| --- | --- |
| 1  2  3  4 | timestamp,originator,action,network,target,context  2012-03-12T22:54:13-07:00,Michael,ADD,I,Ora Hatfield, bla  2012-11-23T01:53:42-08:00,Ted,REMOVE,I,Marvin Garrison, meh  ... |

The raw CSV file contains the following six columns:

* timestamp is an ISO 8601 formatted date-time stamp that states when the action was performed (range: January 2012 to May 2013).
* originator is the name of the person who added or removed a person to or from one of his or her networks.
* action must be either ADD or REMOVE and designates the action that has been carried out. That is, it indicates whether a person has been added or removed from the respective network.
* network is a single character indicating the respective network where the action has been performed. The possible values are: I, in-real-life; T, Twitter; L, LinkedIn; F, Facebook; G, Google+
* target is the name of the person added to or removed from the network.
* context is a free-text comment, providing a hint why the person has been added/removed or where one has met the person in the first place.

There are no optional fields in the dataset. In other words: each row is completely filled. In order to generate some test data to be used in the USN app, I've created a raw input CSV file from [generatedata.com](http://www.generatedata.com/) in five runs, yielding some 500 rows of raw data.

### Technology Stack

USN uses several software frameworks, libraries, and components, as I mentioned earlier. I've tested it with:

* Apache [Hadoop 1.0.4](http://hadoop.apache.org/releases.html#12+October%2C+2012%3A+Release+1.0.4+available)
* Apache [Hive 0.10.0](http://hive.apache.org/releases.html#11+January%2C+2013%3A+release+0.10.0+available)
* [Hiver](https://github.com/tebeka/hiver) for Hive access from Python
* Apache [HBase 0.94.4](http://www.apache.org/dyn/closer.cgi/hbase/)
* [HappyBase](https://github.com/wbolster/happybase) for HBase access from Python

I assume that you're familiar with the bash shell and have Python 2.7 or above installed. I've tested the USN app under Mac OS X 10.8 but there are no hard dependencies on any Mac OS X specific features, so it should run unchanged under any Linux environment.

### Building the USN Data Space

The first step is to build the data space for the USN app, that is, the master dataset and the batch view, and then we will have a closer look behind the scenes of each of the commands.

First, some pre-processing of the raw data, generated earlier:

[?](http://www.drdobbs.com/database/applying-the-big-data-lambda-architectur/240162604)

|  |  |
| --- | --- |
| 1  2  3 | $ pwd  /Users/mhausenblas2/Documents/repos/usn-app/data  $ ./usn-preprocess.sh < usn-raw-data.csv > usn-base-data.csv |

Next we want to build the batch layer. For this, I first need to make sure that the Hive Thrift service is running:

[?](http://www.drdobbs.com/database/applying-the-big-data-lambda-architectur/240162604)

|  |  |
| --- | --- |
| 1  2  3  4  5 | $ pwd  /Users/mhausenblas2/Documents/repos/usn-app/batch-layer  $ hive --service hiveserver  Starting Hive Thrift Server  ... |

Now, I can run the script that execute the Hive queries and builds our USN app master dataset, like so:

[?](http://www.drdobbs.com/database/applying-the-big-data-lambda-architectur/240162604)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | $ pwd  /Users/mhausenblas2/Documents/repos/usn-app/batch-layer  $ ./batch-layer.sh INIT  USN batch layer created.  $ ./batch-layer.sh CHECK  The USN batch layer seems OK. |

This generates the batch layer, which is in HDFS. Next, I create the serving layer in HBase by building a view of the relationships to people. For this, both the Hive and HBase Thrift services need to be running. Below, you see how you start the HBase Thrift service:

$ echo $HBASE\_HOME  
/Users/mhausenblas2/bin/hbase-0.94.4  
$ cd /Users/mhausenblas2/bin/hbase-0.94.4  
$ ./bin/start-hbase.sh   
starting master, logging to /Users/...  
$ ./bin/hbase thrift start -p 9191  
13/05/31 09:39:09 INFO util.VersionInfo: HBase 0.94.4

As now both Hive and HBase Thrift services are up and running, I can run the following command (in the respective directory, wherever you've unzipped or cloned the GitHub repository):

[?](http://www.drdobbs.com/database/applying-the-big-data-lambda-architectur/240162604)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | $ echo $HBASE\_HOME  /Users/mhausenblas2/bin/hbase-0.94.4  $ cd /Users/mhausenblas2/bin/hbase-0.94.4  $ ./bin/start-hbase.sh  starting master, logging to /Users/...  $ ./bin/hbase thrift start -p 9191  13/05/31 09:39:09 INFO util.VersionInfo: HBase 0.94.4 |

Now, let's have a closer look at what is happening behind the scenes of each of the layers in the next sections.

**A look inside a Hadoop-based project that matches connections in social media by leveraging the highly scalable lambda architecture.**

**The Batch Layer**

The raw data is first pre-processed and loaded into Hive. In Hive (remember, this constitutes the master dataset in the batch layer of our USN app) the following schema is used:

[?](http://www.drdobbs.com/database/applying-the-big-data-lambda-architectur/240162604?pgno=2)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | CREATE TABLE usn\_base (   actiontime STRING,   originator STRING,   action STRING,   network STRING,   target STRING,   context STRING  ) ROW FORMAT DELIMITED FIELDS TERMINATED BY '|'; |

To import the CSV data, to build the master dataset, the shell script batch-layer.sh executes the following HiveQL commands:

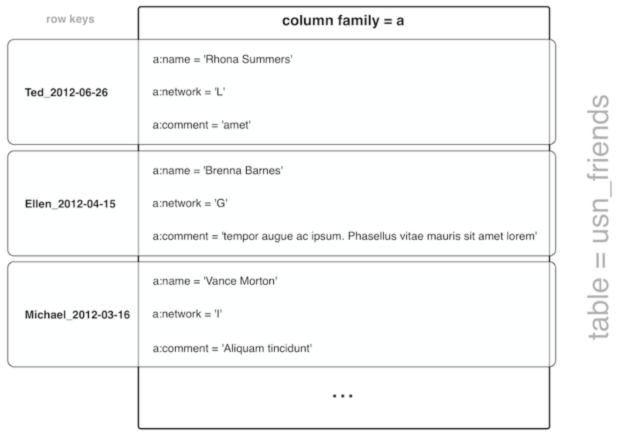
[?](http://www.drdobbs.com/database/applying-the-big-data-lambda-architectur/240162604?pgno=2)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | LOAD DATA LOCAL INPATH '../data/usn-base-data.csv' INTO TABLE usn\_base;    DROP TABLE IF EXISTS usn\_friends;    CREATE TABLE usn\_friends AS  SELECT actiontime, originator AS username, network,         target AS friend, context AS note  FROM usn\_base  WHERE action = 'ADD'  ORDER BY username, network, username; |

With this, the USN app master dataset is ready and available in HDFS and I can move on to the next layer, the serving layer.

**The Serving Layer of the USN App**

The batch view used in the USN app is realized via an HBase table called usn\_friends. This table is then used to drive the USN app front-end; it has the schema shown in Figure 4.

  
**Figure 4: HBase schema used in the serving layer of the USN app.**

After building the serving layer, I can use the HBase shell to verify if the batch view has been properly populated in the respective table usn\_friends:

[?](http://www.drdobbs.com/database/applying-the-big-data-lambda-architectur/240162604?pgno=2)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | $ ./bin/hbase shell  hbase(main):001:0> describe 'usn\_friends'  ...   {NAME => 'usn\_friends', FAMILIES => [{NAME => 'a', DATA\_BLOCK\_ENCODING => 'NONE', BLOOMFILTER => 'N true   ONE', REPLICATION\_SCOPE => '0', VERSIONS => '3', COMPRESSION => 'NONE', MIN\_VERSIONS => '0', TTL =>    '-1', KEEP\_DELETED\_CELLS => 'false', BLOCKSIZE => '65536', IN\_MEMORY => 'false', ENCODE\_ON\_DISK =>    'true', BLOCKCACHE => 'false'}]}  1 row(s) in 0.2450 seconds |

You can have a look at some more queries used in the demo user interface on the [Wiki page](https://github.com/mhausenblas/usn-app/wiki/Serving-Layer) of the GitHub repository.

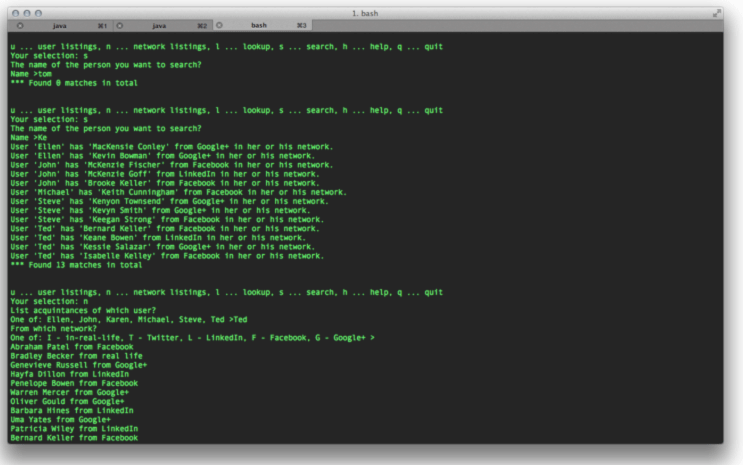
**Putting It All Together**

After the batch and serving layers have been initialized and launched, as described, you can launch the user interface. To use the CLI, make sure that HBase and the HBase Thrift service are running and then, in the main USN app directory run:

[?](http://www.drdobbs.com/database/applying-the-big-data-lambda-architectur/240162604?pgno=2)

|  |  |
| --- | --- |
| 1  2  3  4 | $ ./usn-ui.sh  This is USN v0.0    u ... user listings, n ... network listings, l ... lookup, s ... search, h ... help, q ... quit |

Figure 5 shows a screen shot of the USN app front-end in action:

  
**Figure 5: Screen-shot of the USN app command line user interface.**

The three main operations the USN front-end provides are as follows:

* u ... user listing lists all acquaintances of a user
* n ... network listing lists acquaintances of a user in a network
* l ... lookup listing lists acquaintances of a user in a network and allows restrictions on the time range (from/to) of the acquaintanceship
* s ... search provides search for an acquaintance over all users, allowing for partial match

An example USN app [front-end session](https://github.com/mhausenblas/usn-app#cli-front-end) is available at the GitHub repo for you to study.

**What's Next?**

I have intentionally kept USN simple. Although fully functional, it has several intentional limitations (due to space restrictions here). I can suggest several improvements you could have a go at, using the available [code base](https://github.com/mhausenblas/usn-app) as a starting point.

* Bigger data: The most obvious point is not the app itself but the data size. Only laughable 500 rows? This isn't Big Data I hear you say. Rightly so. Now, no one stops you generating 500 million rows or more and try it out. Certain processes such as pre-processing and the generating the layers will take longer but there are no architectural changes necessary, and this is the whole point of this USN app.
* Creating a full-blown batch layer: Currently, the batch layer is a sort of one-shot, while it should really run in a loop and append new data. This requires partitioning of the ingested data and some checks. [Pail](https://github.com/nathanmarz/dfs-datastores), for example, allows you to do the ingestion and partitioning in a very elegant way.
* Adding speed layer and automated import: It would be interesting to automate the import of data from the various social networks. For example, Google [Takeout](https://www.google.com/takeout/) allows exporting all data in bulk mode, including G+ Circles. For a stab at the speed layer, one could try and utilize the [Twitter fire-hose](https://dev.twitter.com/docs/streaming-apis) along with Storm.
* More batch views: There is currently only one view (friend list per network, per user) in the serving layer. The USN app might benefit from different views to enable different queries most efficiently, such as time-series views of network growth or overlaps of acquaintanceships across networks.

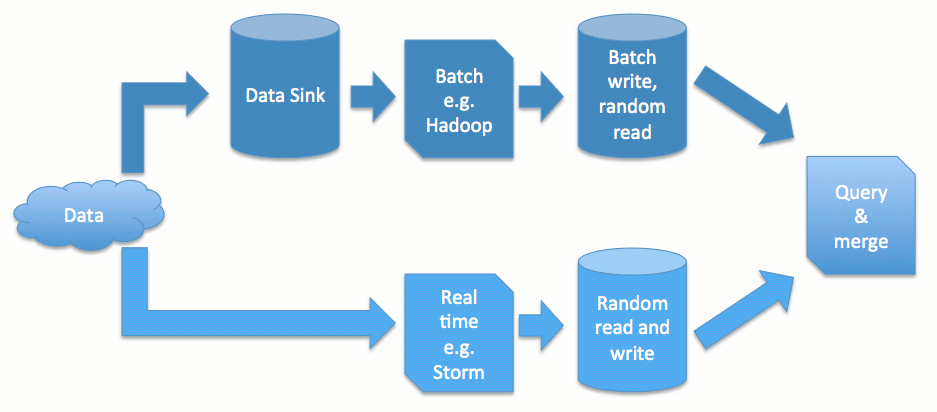
I hope you have as much fun playing around with the USN app and extending it as I had writing it in the first place. I'd love to hear back from you on ideas or further improvements either directly here as a comment or via the GitHub [issue tracker](https://github.com/mhausenblas/usn-app/issues) of the USN app repository.

**Further Resources**

* A must-read for the Lambda Architecture is the [Big Data](http://manning.com/marz/) book by Nathan Marz and James Warren from Manning. The USN app idea actually stems from one of the examples used in this book.
* Slide deck on [A real time architecture using Hadoop and Storm](http://www.slideshare.net/nathan_gs/a-real-time-architecture-using-hadoop-and-storm) from FOSDEM 2013
* A blog post about [an example "lambda architecture" for real-time analysis of hashtags using Trident, Hadoop, and Splout SQL](http://www.datasalt.com/2013/01/an-example-lambda-architecture-using-trident-hadoop-and-splout-sql/)
* Additional batch layer technologies such as [Pail](https://github.com/nathanmarz/dfs-datastores) for managing the master dataset and [JCascalog](https://github.com/nathanmarz/cascalog/wiki/JCascalog) for creating the batch views
* [Apache Drill](http://incubator.apache.org/drill/) for providing interactive, ad-hoc queries against HDFS, HBase, or other NoSQL back-ends.
* Additional speed layer technologies, such as [Trident,](https://github.com/nathanmarz/storm/wiki/Trident-tutorial) a high-level abstraction for doing real-time computing on top of Storm and MapR's [Direct Access NFS](http://www.mapr.com/doc/display/MapR/Accessing+Data+with+NFS) to land data directly from streaming sources such as social media streams or sensor devices.

## **Lambda Architecture explained**

The Lambda Architecture centrally receives data and does as little as possible processing before copying and splitting the data stream to the real time and batch layer. The batch layer collects the data in a data sink like HDFS or S3 in its raw form. Hadoop jobs regularly process the data and write the result to a data store.

[](http://www.semantikoz.com/blog/wp-content/uploads/2014/05/lambda-architecture.png)

Lambda architecture duplicates incoming data and processes them in parallel at different speeds

Since this process is fully batched the data store can have some significant simplification. It should support random reads, i.e. needs some kind of index, however, it can do away with random writing, locking, and consistency issues. This simplifies the store significantly. An example of such a system is ElephantDB.

The problem with batch processing is the time it takes. For example, the above process may take hours or days. In the meantime data has been arriving and subsequent processes or services continue to work with hours or days old information. The real time layer solves this by taking its copy of the data and processing it in seconds or minutes and stores it in a fast random read and write store. This store is more complex since it has to be constantly updated.

The complexity of the real time layer and it’s store is manageable since it only has to store and serve a sliding window of data, which needs to be roughly as long as the batch process takes. Both layers’ results are merged and real time information is replaced in favour of batch layer data. In many cases this enables for the real time process to work with good approximations since its results are replaced by highly precise data within a short period.

## **Lambda Architecture benefits**

The addition of another layer to an architecture has major advantages. Firstly, data can (historically) be processed with high precision and involved algorithms without losing short-term information, alerts, and insights provided by the real time layer. Secondly, the addition of a layer is offset by dramatically reducing the random write storage requirements. The batch write storage provides also the option to switch data at predefined times and version data.

Lastly and importantly, the addition of the data sink of raw data offers the option to recover from human mistakes, i.e. deploying bugs which write erroneous aggregated data from which other architectures can not recover. Another option is to retrospectively enhance data extraction or learning algorithms and apply them on the whole of the historic dataset. This is extremely helpful in agile and startup environments where MVPs push what can be done down the track.

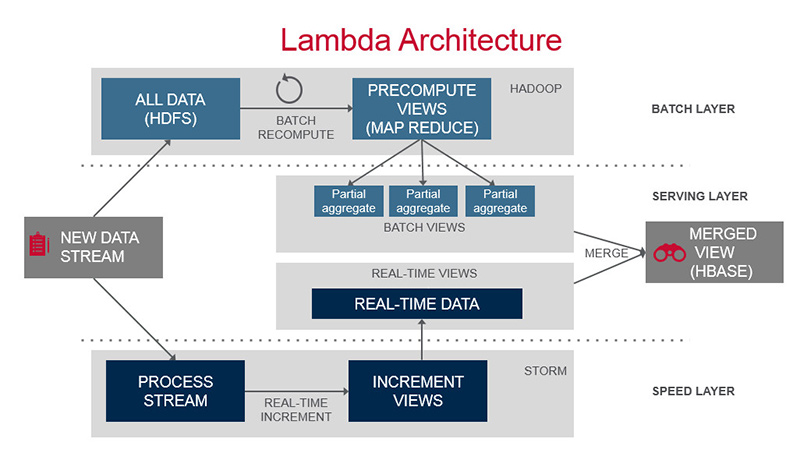
## Making Sense of it All

Building a well-designed, reliable and functional big data application that caters to a variety of end-user latency requirements can be an extremely challenging proposition. It can be daunting enough to just keep up with the rapid pace of technology innovation happening in this space, let alone building applications that work for the problem at hand. “Start slow and build one application at a time” is perhaps the most common advice given to beginners today. However, there are certain high-level architectural constructs that can help you mentally visualize how different types of applications fit into the big data architecture and how some of these technologies are transforming the existing enterprise software landscape.

## Lambda Architecture

Lambda Architecture is a useful framework to think about designing big data applications. Nathan Marz designed this generic architecture addressing common requirements for big data based on his experience working on distributed data processing systems at Twitter.

|  |  |
| --- | --- |
| Some of the key requirements in building this architecture include: Fault-tolerance against hardware failures and human errors Support for a variety of use cases that include low latency querying as well as updates Linear scale-out capabilities, meaning that throwing more machines at the problem should help with getting the job done Extensibility so that the system is manageable and can accommodate newer features easily | [Michael Hausenblas](https://mapr.com/developercentral/lambda-architecture/www.youtube.com/watch?v=rE0KGHbh7ZQ) Michael Hausenblas, Chief Data Engineer at MapR, explains Lambda Architecture |

  
Overview of the Lambda Architecture  
The Lambda Architecture as seen in the picture has three major components.

1. Batch layer that provides the following functionality
   1. managing the master dataset, an immutable, append-only set of raw data
   2. pre-computing arbitrary query functions, called batch views.
2. Serving layer—This layer indexes the batch views so that they can be queried in ad hoc with low latency.
3. Speed layer—This layer accommodates all requests that are subject to low latency requirements. Using fast and incremental algorithms, the speed layer deals with recent data only.

Each of these layers can be realized using various big data technologies. For instance, the batch layer datasets can be in a distributed filesystem, while MapReduce can be used to create batch views that can be fed to the serving layer. The serving layer can be implemented using NoSQL technologies such as HBase, while querying can be implemented by technologies such as Apache Drill or Impala. Finally, the speed layer can be realized with data streaming technologies such as Apache Storm or Spark Streaming.

Up to now, the description of the Lambda Architecture here makes use of the basic capabilities that are pretty much common to all distributions powered by Hadoop. There are somethings you can do, however, with a MapR cluster that improves the basic operation of the Lambda architecture.

For instance, most Storm topologies avoid the use of much persisted state. This is fast and easy, since tuples can be acknowledged as soon as their effect has been impressed on memory. In the Lambda Architecture, this is not supposed to be too big of a deal, since any in-memory state that is lost due to software version upgrades or failures will be repaired within a matter of hours or so as the affected time window ages out of the real-time part of the architecture.

When you have a MapR cluster underneath a Lambda Architecture, however, you can do a bit better than this, so that the times that failures are visible drops to seconds instead of hours.

One way that this works is that MapR allows high-speed streaming data to be written directly to the Hadoop storage layer, while allowing [**stream-processing applications**](https://mapr.com/developercentral/lambda-architecture/assets/mapr_tech-brief_stream_processing.pdf) such as Storm or Spark Streaming to run as an independent service within the cluster. The processing application now becomes more of a subscriber to the incoming data feed. If a failure occurs, and the original application goes down, a new instance of the application can pick up the data stream within seconds of where the original application instance dropped off. An added advantage of this architecture is the availability of streaming data for batch as well as the serving layers.

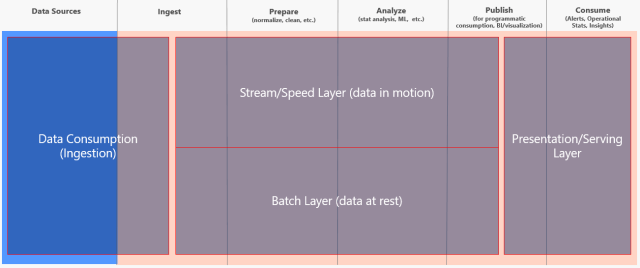
In addition, individual processing elements can delay their acknowledgement of incoming tuples until they have logged the tuple to a log file in the distributed file system. This log file need only persist until the state of the bolt is either persisted or a summary is sent down stream. At that moment, a new log is started.

In the case of failure or orderly exit of a topology, the new version of the bolt can read this log and reconstruct the necessary state of the bolt very quickly. Once the log is read, tuples coming from the spout can be processed as if nothing ever happened. Since all tuples that arrived after the last record in the log have not been acknowledged, the spout will replay them so the bolt will get a complete set of tuples.

Lambda architecture is a data-processing architecture designed to handle massive quantities of data (i.e. “Big Data”) by using both batch-processing and stream-processing methods.  This idea is to balance latency, throughput, scaling, and fault-tolerance by using batch processing to provide comprehensive and accurate views of batch data, while simultaneously using real-time stream processing to provide views of online data.  The two view outputs may be joined before presentation.

This allows for a way to bridge the gap between the historical single version of the truth and the highly sought after “I want it now” real-time solution.  By combining traditional batch processing systems with stream consumption tools the needs of both can be achieved with one solution.

The high-level overview of the Lambda architecture is expressed here:

[](http://www.jamesserra.com/?attachment_id=9666)

A brief explanation of each layer:

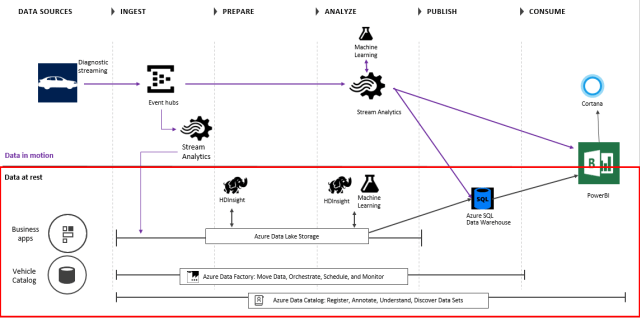
**Data Consumption**: This is where you will import the data from all the various source systems, some of which may be streaming the data.  Others may only provide data once a day.

**Stream Layer**: It provides for incremental updating, making it the more complex layer.  It trades accuracy for low latency, looking at only recent data.  Data in here may be only seconds behind, but the trade-off is the data may not be clean.

**Batch Layer**: It looks at all the data at once and eventually corrects the data in the stream layer.  It is the single version of the truth, the trusted layer, where there is usually lots of ETL and a traditional data warehouse.  This layer is built using a predefined schedule, usually once or twice a day, including importing the data currently stored in the stream layer.

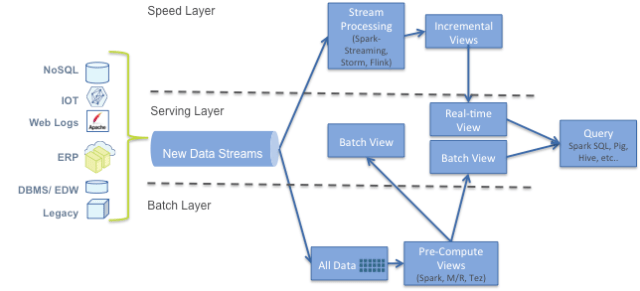
**Presentation Layer**: Think of it as the mediator, as it accepts queries and decides when to use the batch layer and when to use the speed layer.  Its preference would be the batch layer as that has the trusted data, but if you ask it for up-to-the-second data, it will pull from the stream layer.  So it’s a balance of retrieving what we trust versus what we want right now.

A lambda architecture solution using Azure tools might look like this, using a vehicle with IoT sensors as an example:

[](http://www.jamesserra.com/?attachment_id=9672)

In the above diagram, Event Hubs is used to ingest millions of events in real-time.  Stream Analytics is used for 1) real-time aggregations on data and 2) spool data into long-term storage (SQL Data Warehouse) for batch.  Machine Learning is used in real-time for anomaly detection on tire pressure, oil level, engine temp, etc, to predict vehicles requiring maintenance.  The data in the Azure Data Lake Storage is used for rich analytics using HDInsight and Machine Learning, orchestrated by the Azure Data Factory (for e.g. aggressive driving analysis over past year).  Power BI and Cortana are used for the presentation layer, and the Azure Data Catalog is the metadata repository for all the data sets.

Using Hadoop technologies might provide a solution that looks like this:

[](http://www.jamesserra.com/?attachment_id=9668)

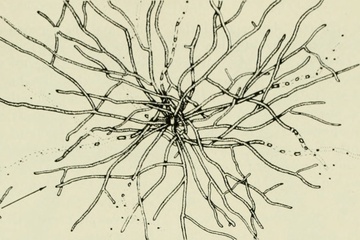
Be aware this is a complicated architecture.  It will need a number of hardware resources and difference code bases for each layer, with each possibly using different technologies/tools.  The complexity of the code can be 3-4 times a traditional data warehouse architecture.  So you will have to weigh the costs versus the benefit of being able to use data slightly newer than a standard data warehouse solution.

# **Questioning the Lambda Architecture**

The Lambda Architecture has its merits, but alternatives are worth exploring.

By [Jay Kreps](https://www.oreilly.com/people/02ea3-jay-kreps)

July 2, 2014

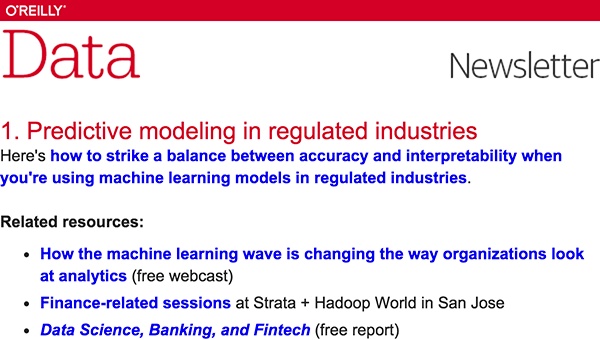
The actinomycetes (source: [Internet Archive Book Images](https://www.flickr.com/photos/internetarchivebookimages/16772552085/))

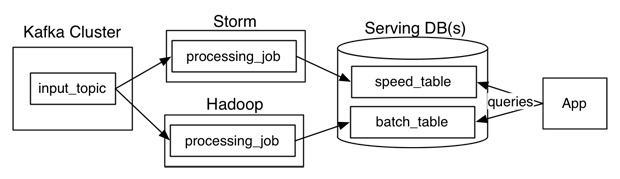
Nathan Marz wrote a popular blog post describing an idea he called the Lambda Architecture (“[How to beat the CAP theorem](http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html)“). The Lambda Architecture is an approach to building stream processing applications on top of MapReduce and [Storm](http://storm.incubator.apache.org/) or similar systems. This has proven to be a surprisingly popular idea, with a dedicated [website](http://lambda-architecture.net/) and an [upcoming book](http://www.manning.com/marz/). Since I’ve been involved in building out the real-time data processing infrastructure at LinkedIn using [Kafka](http://kafka.apache.org/) and [Samza](http://samza.incubator.apache.org/), I often get asked about the Lambda Architecture. I thought I would describe my thoughts and experiences.

## What is a Lambda Architecture and how do I become one?

The Lambda Architecture looks something like this:

**Get O'Reilly's weekly data newsletter**

[](https://www.oreilly.com/ideas/questioning-the-lambda-architecture)



The way this works is that an immutable sequence of records is captured and fed into a batch system and a stream processing system in parallel. You implement your transformation logic twice, once in the batch system and once in the stream processing system. You stitch together the results from both systems at query time to produce a complete answer.

There are a lot of variations on this, and I’m intentionally simplifying a bit. For example, you can swap in various similar systems for Kafka, Storm, and Hadoop, and people often use two different databases to store the output tables, one optimized for real time and the other optimized for batch updates.

The Lambda Architecture is aimed at applications built around complex asynchronous transformations that need to run with low latency (say, a few seconds to a few hours). A good example would be a news recommendation system that needs to crawl various news sources, process and normalize all the input, and then index, rank, and store it for serving.

I have been involved in building a number of real-time data systems and pipelines at LinkedIn. Some of these worked in this style, and upon reflection, it is not my favorite approach. I thought it would be worthwhile to describe what I see as the pros and cons of this architecture, and also give an alternative I prefer.

## What’s good about this?

I like that the Lambda Architecture emphasizes retaining the input data unchanged. I think the discipline of modeling data transformation as a series of materialized stages from an original input has a lot of merit. This is one of the things that makes large MapReduce workflows tractable, as it enables you to debug each stage independently. I think this lesson translates well to the stream processing domain. I’ve written some of my thoughts about capturing and transforming immutable data streams [here](http://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying).

I also like that this architecture highlights the problem of reprocessing data. Reprocessing is one of the key challenges of stream processing but is very often ignored. By “reprocessing,” I mean processing input data over again to re-derive output. This is a completely obvious but often ignored requirement. Code will always change. So, if you have code that derives output data from an input stream, whenever the code changes, you will need to recompute your output to see the effect of the change.

Why does code change? It might change because your application evolves and you want to compute new output fields that you didn’t previously need. Or it might change because you found a bug and need to fix it. Regardless, when it does, you need to regenerate your output. I have found that many people who attempt to build real-time data processing systems don’t put much thought into this problem and end-up with a system that simply cannot evolve quickly because it has no convenient way to handle reprocessing. The Lambda Architecture deserves a lot of credit for highlighting this problem.

There are a number of other motivations proposed for the Lambda Architecture, but I don’t think they make much sense. One is that real-time processing is inherently approximate, less powerful, and more lossy than batch processing. I actually do not think this is true. It is true that the existing set of stream processing frameworks are less mature than MapReduce, but there is no reason that a stream processing system can’t give as strong a semantic guarantee as a batch system.

Another explanation I have heard is that the Lambda Architecture somehow “beats the CAP theorem” by allowing a mixture of different data systems with different trade-offs. Long story short, although there are definitely latency/availability trade-offs in stream processing, this is an architecture for asynchronous processing, so the results being computed are not kept immediately consistent with the incoming data. The CAP theorem, sadly, [remains intact](http://ferd.ca/beating-the-cap-theorem-checklist.html).

## And the bad…

The problem with the Lambda Architecture is that maintaining code that needs to produce the same result in two complex distributed systems is exactly as painful as it seems like it would be. I don’t think this problem is fixable.

Programming in distributed frameworks like Storm and Hadoop is complex. Inevitably, code ends up being specifically engineered toward the framework it runs on. The resulting operational complexity of systems implementing the Lambda Architecture is the one thing that seems to be universally agreed on by everyone doing it.

Why can’t the stream processing system be improved to handle the full problem set in its target domain?One proposed approach to fixing this is to have a language or framework that abstracts over both the real-time and batch framework. You write your code using this higher level framework and then it “compiles down” to stream processing or MapReduce under the covers. [Summingbird](http://github.com/twitter/summingbird) is a framework that does this. This definitely makes things a little better, but I don’t think it solves the problem.

Ultimately, even if you can avoid coding your application twice, the operational burden of running and debugging two systems is going to be very high. And any new abstraction can only provide the features supported by the intersection of the two systems. Worse, committing to this new uber-framework walls off the rich ecosystem of tools and languages that makes Hadoop so powerful (Hive, Pig, Crunch, Cascading, Oozie, etc).

By way of analogy, consider the notorious difficulties in making cross-database ORM really transparent. And consider that this is just a matter of abstracting over very similar systems providing virtually identical capabilities with a (nearly) standardized interface language. The problem of abstracting over totally divergent programming paradigms built on top of barely stable distributed systems is much harder.

## We have done this experiment

We have actually been through a number of rounds of this at LinkedIn. We have built various hybrid-Hadoop architectures and even a domain-specific API that would allow code to be “transparently” run either in real time or in Hadoop. These approaches worked, but none were very pleasant or productive. Keeping code written in two different systems perfectly in sync was really, really hard. The API meant to hide the underlying frameworks proved to be the leakiest of abstractions. It ended up requiring deep Hadoop knowledge as well as deep knowledge of the real-time layer — and adding the new requirement that you understand enough about how the API would translate to these underlying systems whenever you were debugging problems or trying to reason about performance.

These days, my advice is to use a batch processing framework like MapReduce if you aren’t latency sensitive, and use a stream processing framework if you are, but not to try to do both at the same time unless you absolutely must.

So, why the excitement about the Lambda Architecture? I think the reason is because people increasingly need to build complex, low-latency processing systems. What they have at their disposal are two things that don’t quite solve their problem: a scalable high-latency batch system that can process historical data and a low-latency stream processing system that can’t reprocess results. By duct taping these two things together, they can actually build a working solution.

In this sense, even though it can be painful, I think the Lambda Architecture solves an important problem that was otherwise generally ignored. But I don’t think this is a new paradigm or the future of big data. It is just a temporary state driven by the current limitation of off-the-shelf tools. I also think there are better alternatives.

## An alternative

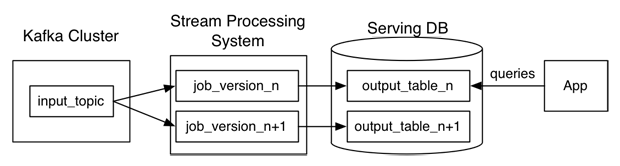
As someone who designs infrastructure, I think the glaring question is this: why can’t the stream processing system just be improved to handle the full problem set in its target domain? Why do you need to glue on another system? Why can’t you do both real-time processing and also handle the reprocessing when code changes? Stream processing systems already have a notion of parallelism; why not just handle reprocessing by increasing the parallelism and replaying history very, very fast? The answer is that you can do this, and I think this it is actually a reasonable alternative architecture if you are building this type of system today.

When I’ve discussed this with people, they sometimes tell me that stream processing feels inappropriate for high-throughput processing of historical data. But I think this is an intuition based mostly on the limitations of systems they have used, which either scale poorly or can’t save historical data. This leaves them with a sense that a stream processing system is inherently something that computes results off some ephemeral streams and then throws all the underlying data away. But there is no reason this should be true. The fundamental abstraction in stream processing is data flow DAGs, which are exactly the same underlying abstraction in a traditional data warehouse (a la [Volcano](http://paperhub.s3.amazonaws.com/dace52a42c07f7f8348b08dc2b186061.pdf)) as well as being the fundamental abstraction in the MapReduce successor [Tez](http://hortonworks.com/hadoop/tez/). Stream processing is just a generalization of this data-flow model that exposes checkpointing of intermediate results and continual output to the end user.

So, how can we do the reprocessing directly from our stream processing job? My preferred approach is actually stupidly simple:

1. Use Kafka or some other system that will let you retain the full log of the data you want to be able to reprocess and that allows for multiple subscribers. For example, if you want to reprocess up to 30 days of data, set your retention in Kafka to 30 days.
2. When you want to do the reprocessing, start a second instance of your stream processing job that starts processing from the beginning of the retained data, but direct this output data to a new output table.
3. When the second job has caught up, switch the application to read from the new table.
4. Stop the old version of the job, and delete the old output table.

This architecture looks something like this:



Unlike the Lambda Architecture, in this approach you only do reprocessing when your processing code changes, and you actually need to recompute your results. And, of course, the job doing the re-computation is just an improved version of the same code, running on the same framework, taking the same input data. Naturally, you will want to bump up the parallelism on your reprocessing job so it completes very quickly.

Maybe we could call this the Kappa Architecture, though it may be too simple of an idea to merit a Greek letter.

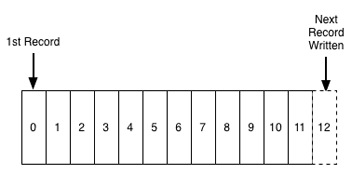
Of course, you can optimize this further. In many cases, you could combine the two output tables. However, I think there are some benefits to having both for a short period of time. This allows you to revert back instantaneously to the old logic by just having a button that redirects the application to the old table. And in cases that are particularly important (your ad targeting criteria, say), you can control the cut-over with an automatic A/B test or [bandit algorithm](http://shop.oreilly.com/product/0636920027393.do) to ensure whatever bug fix or code improvement you are rolling out hasn’t accidentally degraded things in comparison to the prior version.

Note that this this doesn’t mean your data can’t go to HDFS; it just means that you don’t run your reprocessing there. Kafka has good integration with Hadoop, so mirroring any Kafka topic into HDFS is easy. It is often useful for the output or even intermediate streams from a stream processing job to be available in Hadoop for analysis in tools like Hive or for use as input for other, offline data processing flows.

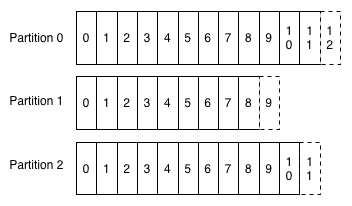
We have [documented](http://samza.incubator.apache.org/learn/documentation/0.7.0/jobs/reprocessing.html) implementing this approach as well as other variations on reprocessing architectures using Samza.

## Some background

For those less familiar with Kafka, what I just described may not make sense. A quick refresher will hopefully straighten things out. Kafka maintains ordered logs like this:



A Kafka “topic” is a collection of these logs:



A stream processor consuming this data just maintains an “offset,” which is the log entry number for the last record it has processed on each of these partitions. So, changing the consumer’s position to go back and reprocess data is as simple as restarting the job with a different offset. Adding a second consumer for the same data is just another reader pointing to a different position in the log.

Kafka supports replication and fault-tolerance, runs on cheap, commodity hardware, and is glad to store many TBs of data per machine. So, retaining large amounts of data is a perfectly natural and economical thing to do and won’t hurt performance. LinkedIn keeps more than a petabyte of Kafka storage online, and a number of applications make good use of this long retention pattern for exactly this purpose.

Cheap consumers and the ability to retain large amounts of data make adding the second “reprocessing” job just a matter of firing up a second instance of your code but starting from a different position in the log.

This design is not an accident. We built Kafka with the intent of using it as a substrate for stream processing, and we had in mind exactly this model for handling reprocessing data. For the curious, you can find more information on Kafka [here](https://kafka.apache.org/documentation.html#introduction).

Fundamentally, though, there is nothing that ties this idea to Kafka. You could substitute any system that supports long retention of ordered data (for example HDFS, or some kind of database). Indeed, a lot of people are familiar with similar patterns that go by the name [Event Sourcing](http://martinfowler.com/eaaDev/EventSourcing.html) or [CQRS](http://martinfowler.com/bliki/CQRS.html). And, of course, the distributed database people will tell you this is just a slight rebranding of materialized view maintenance, which, as they will gladly remind you, they figured out a long long time ago, sonny.

## Comparison

I know this approach works well using Samza as the stream processing system because we do it at LinkedIn. But I am not aware of any reason it shouldn’t work equally well in Storm or other stream processing systems. I’m not familiar enough with Storm to work through the practicalities, so I’d be glad to hear if others are doing this already. In any case, I think the general ideas are fairly system independent.

The efficiency and resource trade-offs between the two approaches are somewhat of a wash. The Lambda Architecture requires running both reprocessing and live processing all the time, whereas what I have proposed only requires running the second copy of the job when you need reprocessing. However, my proposal requires temporarily having 2x the storage space in the output database and requires a database that supports high-volume writes for the re-load. In both cases, the extra load of the reprocessing would likely average out. If you had many such jobs, they wouldn’t all reprocess at once, so on a shared cluster with several dozen such jobs you might budget an extra few percent of capacity for the few jobs that would be actively reprocessing at any given time.

The real advantage isn’t about efficiency at all, but rather about allowing people to develop, test, debug, and operate their systems on top of a single processing framework. So, in cases where simplicity is important, consider this approach as an alternative to the Lambda Architecture.

## [How to beat the CAP theorem](http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html)

DateTHURSDAY, OCTOBER 13, 2011

The CAP theorem states a database cannot guarantee consistency, availability, and partition-tolerance at the same time. But you can't sacrifice partition-tolerance (see [here](http://codahale.com/you-cant-sacrifice-partition-tolerance/) and [here](http://www.cloudera.com/blog/2010/04/cap-confusion-problems-with-partition-tolerance/)), so you must make a tradeoff between availability and consistency. Managing this tradeoff is a central focus of the NoSQL movement.

Consistency means that after you do a successful write, future reads will always take that write into account. Availability means that you can always read and write to the system. During a partition, you can only have one of these properties.

Systems that choose consistency over availability have to deal with some awkward issues. What do you do when the database isn't available? You can try buffering writes for later, but you risk losing those writes if you lose the machine with the buffer. Also, buffering writes can be a form of inconsistency because a client thinks a write has succeeded but the write isn't in the database yet. Alternatively, you can return errors back to the client when the database is unavailable. But if you've ever used a product that told you to "try again later", you know how aggravating this can be.

The other option is choosing availability over consistency. The best consistency guarantee these systems can provide is "eventual consistency". If you use an eventually consistent database, then sometimes you'll read a different result than you just wrote. Sometimes multiple readers reading the same key at the same time will get different results. Updates may not propagate to all replicas of a value, so you end up with some replicas getting some updates and other replicas getting different updates. It is up to you to repair the value once you detect that the values have diverged. This requires tracing back the history using vector clocks and merging the updates together (called "read repair").

I believe that maintaining eventual consistency in the application layer is too heavy of a burden for developers. Read-repair code is extremely susceptible to developer error; if and when you make a mistake, faulty read-repairs will introduce irreversible corruption into the database.

So sacrificing availability is problematic and eventual consistency is too complex to reasonably build applications. Yet these are the only two options, so it seems like I'm saying that you're damned if you do and damned if you don't. The CAP theorem is a fact of nature, so what alternative can there possibly be?

There is another way. You can't avoid the CAP theorem, but you can isolate its complexity and prevent it from sabotaging your ability to reason about your systems. The complexity caused by the CAP theorem is a symptom of fundamental problems in how we approach building data systems. Two problems stand out in particular: the use of mutable state in databases and the use of incremental algorithms to update that state. It is the interaction between these problems and the CAP theorem that causes complexity.

In this post I'll show the design of a system that beats the CAP theorem by preventing the complexity it normally causes. But I won't stop there. The CAP theorem is a result about the degree to which data systems can be fault-tolerant to machine failure. Yet there's a form of fault-tolerance that's much more important than machine fault-tolerance: human fault-tolerance. If there's any certainty in software development, it's that developers aren't perfect and bugs will inevitably reach production. Our data systems must be resilient to buggy programs that write bad data, and the system I'm going to show is as human fault-tolerant as you can get.

This post is going to challenge your basic assumptions on how data systems should be built. But by breaking down our current ways of thinking and re-imagining how data systems should be built, what emerges is an architecture more elegant, scalable, and robust than you ever thought possible.

### What is a data system?

Before we talk about system design, let's first define the problem we're trying to solve. What is the purpose of a data system? What is data? We can't even begin to approach the CAP theorem unless we can answer these questions with a definition that clearly encapsulates every data application.

Data applications range from storing and retrieving objects, joins, aggregations, stream processing, continuous computation, machine learning, and so on and so on. It's not clear that there is such a simple definition of data systems -- it seems that the range of things we do with data is too diverse to capture with a single definition.

However, there is such a simple definition. This is it:

Query = Function(All Data)

That's it. This equation summaries the entire field of databases and data systems. Everything in the field -- the past 50 years of RDBMS's, indexing, OLAP, OLTP, MapReduce, ETL, distributed filesystems, stream processors, NoSQL, etc. -- is summarized by that equation in one way or another.

A data system answers questions about a dataset. Those questions are called "queries". And this equation states that a query is just a function of all the data you have.

This equation may seem too general to be useful. It doesn't seem to capture any of the intricacies of data system design. But what matters is that every data system falls into that equation. The equation is a starting point from which we can explore data systems, and the equation will eventually lead to a method for beating the CAP theorem.

There are two concepts in this equation: "data" and "queries". These are distinct concepts that are often conflated in the database field, so let's be rigorous about what these concepts mean.

#### Data

Let's start with "data". A piece of data is an indivisible unit that you hold to be true for no other reason than it exists. It is like an axiom in mathematics.

There are two crucial properties to note about data. First, data is inherently time based. A piece of data is a fact that you know to be true at some moment of time. For example, suppose Sally enters into her social network profile that she lives in Chicago. The data you take from that input is that she lived in Chicago as of the particular moment in time that she entered that information into her profile. Suppose that on a later date Sally updates her profile location to Atlanta. Then you know that she lived in Atlanta as of that particular time. The fact that she lives in Atlanta now doesn't change the fact that she used to live in Chicago. Both pieces of data are true.

The second property of data follows immediately from the first: data is inherently immutable. Because of its connection to a point in time, the truthfulness of a piece of data never changes. One cannot go back in time to change the truthfulness of a piece of data. This means that there are only two main operations you can do with data: read existing data and add more data. [CRUD](http://en.wikipedia.org/wiki/Create,_read,_update_and_delete) has become CR.

I've left out the "Update" operation. This is because updates don't make sense with immutable data. For example, "updating" Sally's location really means that you're adding a new piece of data saying she lives in a new location as of a more recent time.

I've also left out the "Delete" operation. Again, most cases of deletes are better represented as creating new data. For example, if Bob stops following Mary on Twitter, that doesn't change the fact that he used to follow her. So instead of deleting the data that says he follows her, you'd add a new data record that says he un-followed her at some moment in time.

There are a few cases where you do want to permanently delete data, such as regulations requiring you to purge data after a certain amount of time. These cases are easily supported by the data system design I'm going to show, so for the purposes of simplicity we can ignore these cases.

This definition of data is almost certainly different than what you're used to, especially if you come from the relational database world where updates are the norm. There are two reasons for this. First, this definition of data is extremely generic: it's hard to think of a kind of data that doesn't fit under this definition. Second, the immutability of data is the key property we're going to exploit in designing a human fault-tolerant data system that beats the CAP theorem.

#### Query

The second concept in the equation is the "query". A query is a derivation from a set of data. In this sense, a query is like a theorem in mathematics. For example, "What is Sally's current location?" is a query. You would compute this query by returning the most recent data record about Sally's location. Queries are functions of the complete dataset, so they can do anything: aggregations, join together different types of data, and so on. So you might query for the number of female users of your service, or you might query a dataset of tweets for what topics have been trending in the past few hours.

I've defined a query as a function on the complete dataset. Of course, many queries don't need the complete dataset to run -- they only need a subset of the dataset. But what matters is that my definition encapsulates all possible queries, and if we're going to beat the CAP theorem, we must be able to do so for any query.

### Beating the CAP theorem

The simplest way to compute a query is to literally run a function on the complete dataset. If you could do this within your latency constraints, then you'd be done. There would be nothing else to build.

Of course, it's infeasible to expect a function on a complete dataset to finish quickly. Many queries, such as those that serve a website, require millisecond response times. However, let's pretend for a moment that you can compute these functions quickly, and let's see how a system like this interacts with the CAP theorem. As you are about to see, a system like this not only beats the CAP theorem, but annihilates it.

The CAP theorem still applies, so you need to make a choice between consistency and availability. The beauty is that once you decide on the tradeoff you want to make, you're done. The complexity the CAP theorem normally causes is avoided by using immutable data and computing queries from scratch.

If you choose consistency over availability, then not much changes from before. Sometimes you won't be able to read or write data because you traded off availability. But for the cases where rigid consistency is a necessity, it's an option.

Things get much more interesting when you choose availability over consistency. In this case, the system is eventually consistent without any of the complexities of eventual consistency. Since the system is highly available, you can always write new data and compute queries. In failure scenarios, queries will return results that don't incorporate previously written data. Eventually that data will be consistent and queries will incorporate that data into their computations.

The key is that data is immutable. Immutable data means there's no such thing as an update, so it's impossible for different replicas of a piece of data to become inconsistent. This means there are no divergent values, vector clocks, or read-repair. From the perspective of queries, a piece of data either exists or doesn't exist. There is just data and functions on that data. There's nothing you need to do to enforce eventual consistency, and eventual consistency does not get in the way of reasoning about the system.

What caused complexity before was the interaction between incremental updates and the CAP theorem. Incremental updates and the CAP theorem really don't play well together; mutable values require read-repair in an eventually consistent system. By rejecting incremental updates, embracing immutable data, and computing queries from scratch each time, you avoid that complexity. The CAP theorem has been beaten.

Of course, what we just went through was a thought experiment. Although we'd like to be able to compute queries from scratch each time, it's infeasible. However, we have learned some key properties of what a real solution will look like:

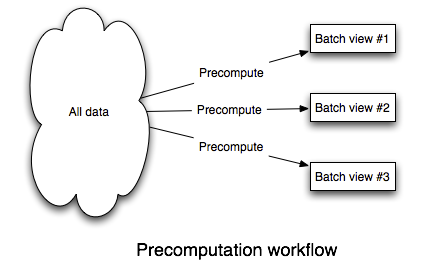
1. The system makes it easy to store and scale an immutable, constantly-growing dataset
2. The primary write operation is adding new immutable facts of data
3. The system avoids the complexity of the CAP theorem by recomputing queries from raw data
4. The system uses incremental algorithms to lower the latency of queries to an acceptable level

Let's begin our exploration of what such a system looks like. Note that everything from here on out is optimization. Databases, indexing, ETL, batch computation, stream processing -- these are all techniques for optimizing query functions and bringing the latency down to an acceptable level. This is a simple but profound realization. Databases are usually made out to be the centerpiece of data management, but really they're one part of a bigger picture.

### Batch computation

Figuring out how to make an arbitrary function on an arbitrary dataset run quickly is a daunting problem. So let's relax the problem a little bit. Let's pretend that it's okay for queries to be out of date by a few hours. Relaxing the problem this way leads to a simple, elegant, and general-purpose solution for building data systems. Afterwards, we'll extend the solution so that the problem is no longer relaxed.

Since a query is a function of all the data, the easiest way to make queries run fast is to precompute them. Whenever there's new data, you just recompute everything. This is feasible because we relaxed the problem to allow queries to be out of date by a few hours. Here's an illustration of this workflow:



To build this, you need a system that:

1. Can easily store a large and constantly growing dataset
2. Can compute functions on that dataset in a scalable way

Such a system exists. It's mature, battle-tested across hundreds of organizations, and has a large ecosystem of tools. It's called [Hadoop](http://hadoop.apache.org/). Hadoop [isn't perfect](http://tech.backtype.com/the-dark-side-of-hadoop), but it's the best tool out there for doing batch processing.

A lot of people will tell you that Hadoop is only good for "unstructured" data. This is completely false. Hadoop is fantastic for structured data. Using tools like [Thrift](http://thrift.apache.org/) or [Protocol Buffers](http://code.google.com/p/protobuf/), you can store your data using rich, evolvable schemas.

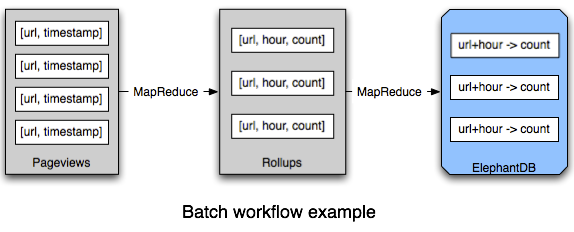
Hadoop is comprised of two pieces: a distributed filesystem (HDFS), and a batch processing framework (MapReduce). HDFS is good at storing a large amount of data across files in a scalable way. MapReduce is good at running computations on that data in a scalable way. These systems match our needs perfectly.

We'll store data in flat files on HDFS. A file will contain a sequence of data records. To add new data, you simply append a new file containing new data records to the folder that contains all the data. Storing data like this on HDFS solves the "Store a large and constantly growing dataset" requirement.

Precomputing queries off of that data is similarly straightforward. MapReduce is an expressive enough paradigm such that nearly any function can be implemented as a series of MapReduce jobs. Tools like [Cascalog](https://github.com/nathanmarz/cascalog), [Cascading](http://cascading.org/), and [Pig](http://pig.apache.org/) make implementing these functions much easier.

Finally, you need to index the results of the precomputation so that the results can be quickly accessed by an application. There's a class of databases that are extremely good at this. [ElephantDB](https://github.com/nathanmarz/elephantdb) and [Voldemort read-only](http://sna-projects.com/blog/2009/06/voldemort-and-hadoop/) specialize in exporting key/value data from Hadoop for fast querying. These databases support batch writes and random reads, and they do not support random writes. Random writes cause most of the complexity in databases, so by not supporting random writes these databases are extraordinarily simple. ElephantDB, for example, is only a few thousand lines of code. That simplicity leads to these databases being extremely robust.

Let's look at an example of how the batch system fits together. Suppose you're building a web analytics application that tracks page views, and you want to be able to query the number of page views over any period of time, to a granularity of one hour.



Implementing this is easy. Each data record contains a single page view. Those data records are stored in files on HDFS. A function that rolls up page views per URL by hour is implemented as a series of MapReduce jobs. The function emits key/value pairs, where each key is a [URL, hour] pair and each value is a count of the number of page views. Those key/value pairs are exported into an ElephantDB database so that an application can quickly get the value for any [URL, hour] pair. When an application wants to know the number of page views for a time range, it queries ElephantDB for the number of page views for each hour in that time range and adds them up to get the final result.

Batch processing can compute arbitrary functions on arbitrary data with the drawback that queries are out of date by a few hours. The "arbitrariness" of such a system means it can be applied to any problem. More importantly, it's simple, easy to understand, and completely scalable. You just have to think in terms of data and functions, and Hadoop takes care of the parallelization.

### The batch system, CAP, and human fault-tolerance

So far so good. So how does the batch system I've described line up with CAP, and does it meet our goal of being human fault-tolerant?

Let's start with CAP. The batch system is eventually consistent in the most extreme way possible: writes always take a few hours to be incorporated into queries. But it's a form of eventual consistency that's easy to reason about because you only have to think about data and functions on that data. There's no read-repair, concurrency, or other complex issues to consider.

Next, let's take a look at the batch system's human fault-tolerance. The human fault-tolerance of the batch system is as good as you can get. There are only two mistakes a human can make in a system like this: deploy a buggy implementation of a query or write bad data.

If you deploy a buggy implementation of a query, all you have to do to fix things is fix the bug, deploy the fixed version, and recompute everything from the master dataset. This works because queries are pure functions.

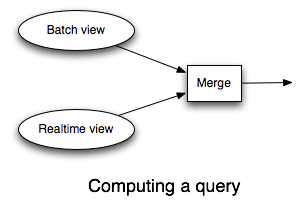
Likewise, writing bad data has a clear path to recovery: delete the bad data and precompute the queries again. Since data is immutable and the master dataset is append-only, writing bad data does not override or otherwise destroy good data. This is in stark contrast to almost all traditional databases where if you update a key you lose the old value.

Note that [MVCC](http://en.wikipedia.org/wiki/Multiversion_concurrency_control) and HBase-like row versioning do not come close to this level of human fault-tolerance. MVCC and HBase row versioning don't keep data around forever: once the database compacts the row, the old value is gone. Only an immutable dataset guarantees that you have a path to recovery when bad data is written.

### Realtime layer

Believe it or not, the batch solution almost solves the complete problem of computing arbitrary functions on arbitrary data in realtime. Any data older than a few hours has already been incorporated into the batch views, so all that's left to do is compensate for the last few hours of data. Figuring out how to make queries realtime against a few hours of data is much easier than doing so against the complete dataset. This is a critical insight.

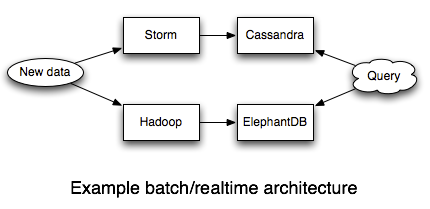
To compensate for those few hours of data, you need a realtime system that runs in parallel with the batch system. The realtime system precomputes each query function for the last few hours of data. To resolve a query function, you query the batch view and the realtime view and merge the results together to get the final answer.



The realtime layer is where you use read/write databases like Riak or Cassandra, and the realtime layer relies on incremental algorithms to update the state in those databases.

The analog to Hadoop for realtime computation is [Storm](https://github.com/nathanmarz/storm/wiki). I wrote Storm to make it easy to do large amounts of realtime data processing in a way that's scalable and robust. Storm runs infinite computations over streams of data and gives strong guarantees on the processing of the data.

Let's see an example of the realtime layer by going back to the running example of querying the number of page views for a URL over a time range.



The batch system is the same as before: a batch workflow based on Hadoop and ElephantDB precomputes the query for everything but the last few hours of data. All that's left is to build the realtime system that compensates for those last few hours of data.

We'll roll up the stats for the last few hours into Cassandra, and we'll use Storm to process the stream of pageviews and parallelize the updates into the database. Each pageview leads to a counter for a [URL, hour] key to be incremented in Cassandra. That's all there is to it -- Storm makes these kinds of things very simple.

### Batch layer + realtime layer, the CAP theorem, and human fault-tolerance

In some ways it seems like we're back to where we started. Achieving realtime queries required us to use NoSQL databases and incremental algorithms. This means we're back in the complex world of divergent values, vector clocks, and read-repair.

There's a key difference though. Since the realtime layer only compensates for the last few hours of data, everything the realtime layer computes is eventually overridden by the batch layer. So if you make a mistake or something goes wrong in the realtime layer, the batch layer will correct it. All that complexity is transient.

This doesn't mean you shouldn't care about read-repair or eventual consistency in the realtime layer. You still want the realtime layer to be as consistent as possible. But when you make a mistake you don't permanently corrupt your data. This removes a huge complexity burden from your shoulders.

In the batch layer, you only have to think about data and functions on that data. The batch layer is really simple to reason about. In the realtime layer, on the other hand, you have to use incremental algorithms and extremely complex NoSQL databases. Isolating all that complexity into the realtime layer makes a huge difference in making robust, reliable systems.

Additionally, the realtime layer doesn't affect the human fault-tolerance of the system. The append-only immutable dataset in the batch layer is still the core of the system, so any mistake can be recovered from just like before.

Let me share a personal story about the great benefits of isolating complexity in the realtime layer. I had a system very much like the one I described here: Hadoop and ElephantDB for the batch layer, and Storm and Cassandra for the realtime layer. Due to poor monitoring on my part, I woke up one day to discover that Cassandra had run out of space and was timing out on every request. This caused my Storm topology to fail and the stream of data to back up on the queues. The same messages kept getting replayed (and kept failing) over and over.

If I didn't have a batch layer, I would have been forced to scale and recover Cassandra. This is non-trivial. Even worse, much of the database was likely inaccurate due to the same messages being replayed many times.

Fortunately, all this complexity was isolated in my realtime layer. I flushed the backed up queues into the batch layer and made a fresh Cassandra cluster. The batch layer ran like clockwork and within a few hours everything was back to normal. No data was lost and there was no inaccuracy in our queries.

### Garbage collection

Everything I've described in this post is built upon the foundation of an immutable, constantly growing dataset. So what do you do if your dataset is so large that it's impractical to store all data for all time, even with horizontally scalable storage? Does this use case break everything I've described? Should you go back to using mutable databases?

No. It's easy to extend the basic model with "garbage collection" to handle this use case. Garbage collection is simply a function that takes in the master dataset and returns a filtered version of the master dataset. Garbage collection gets rid of data that is of low value. You can use any strategy you want for garbage collection. You can simulate mutability by only keeping the last value for an entity, or you can keep a history for each entity. For example, if you're dealing with location data, you may want to keep one location per person per year along with the current location. Mutability is really just an inflexible form of garbage collection (that also interacts poorly with the CAP theorem).

Garbage collection is implemented as a batch processing task. It's something you run occasionally, perhaps once per month. Since garbage collection is run as an offline batch processing task, it doesn't affect how the system interacts with the CAP theorem.

### Conclusion

What makes scalable data systems difficult isn't the CAP theorem. It's a reliance on incremental algorithms and mutable state that leads to complexity in our systems. It's only recently with the rise of distributed databases that this complexity has gotten out of control. But that complexity has always been there.

I said in the beginning of this post that I would challenge your basic assumptions of how data systems should be built. I turned CRUD into CR, split persistence into separate batch and realtime systems, and obsessed over the importance of human fault-tolerance. It took a lot of hard-earned experience over the years to break my old assumptions and arrive at these conclusions.

The batch/realtime architecture has a lot of interesting capabilities that I didn't cover yet. It's worth summarizing some of these now:

1. **Algorithmic flexibility:** Some algorithms are difficult to compute incrementally. Computing unique counts, for example, can be challenging if the sets of uniques get large. The batch/realtime split gives you the flexibility to use the exact algorithm on the batch layer and an approximate algorithm on the realtime layer. The batch layer constantly overrides the realtime layer, so the approximation gets corrected and your system exhibits the property of "eventual accuracy".
2. **Schema migrations are easy:** Gone are the days of difficult schema migrations. Since batch computation is at the core of the system, it's easy to run functions on the complete dataset. This makes it easy to change the schema of your data or views.
3. **Easy ad-hoc analysis:** The arbitrariness of the batch layer means you can run any query you like on your data. Since all data is accessible in one location, this is easy and convenient.
4. **Self-auditing:** By treating data as immutable, you get a self-auditing dataset. The dataset records its own history. I've discussed how important this is for human fault-tolerance, but it's also super useful for doing analytics.

I don't claim to have "solved" the Big Data space, but I've laid down the framework for thinking about Big Data. The batch/realtime architecture is highly general and can be applied to any data system. Rather than give you a fish or a fishing rod, I've shown you how to make a fishing rod for any kind of fish and any kind of water.

There's lots more work to do to improve our collective ability to attack Big Data problems. Here are some key areas of improvement:

1. **Expanded data models for batch-writable, random-read databases:** Not every application is supported by a key/value data model. This is why my team is investing in expanding ElephantDB to support search, document databases, range queries, and more.
2. **Better batch processing primitives**: Hadoop is not the end-all-be-all of batch computation. It can be inefficient for certain kinds of computations. [Spark](https://github.com/mesos/spark) is an important project doing interesting work in expanding the MapReduce paradigm.
3. **Improved read/write NoSQL databases**: There's room for more databases with different data models, and these projects in general will benefit from more maturation.
4. **High level abstractions:** One of the most interesting areas of future work is high level abstractions that map to a batch processing component and a realtime processing component. There's no reason why you shouldn't have the conciseness of a declarative language with the robustness of the batch/realtime architecture.

A lot of people want a scalable relational database. What I hope you've realized in this post is that you don't want that at all! Big data and the NoSQL movement seemed to make data management more complex than it was with the RDBMS, but that's only because we were trying to treat "Big Data" the same way we treated data with an RDBMS: by conflating data and views and relying on incremental algorithms. The scale of big data lets you build systems in a completely different way. By storing data as a constantly expanding set of immutable facts and building recomputation into the core, a Big Data system is actually easier to reason about than a relational system. And it scales.

# Lambda Architecture: A state-of-the-art

Written by [Pere Ferrera Bertran](http://www.datasalt.com/author/pere/) on January 17, 2014 — [No Comments](http://www.datasalt.com/2014/01/lambda-architecture-a-state-of-the-art/#respond)

It’s been some time now since Nathan Marz wrote the [first Lambda Architecture post](http://www.databasetube.com/database/big-data-lambda-architecture/). What has happened since then? How has the community reacted to such a concept? What are the architectural trends in the Big Data space,  as well as the challenges and remaining problems?

### Big Data: Batch processing-only

Despite the attractiveness of a dual batch / real-time architecture, there exists a wide variety of problems in Big Data for which **a batch layer is good enough**, and I think it will continue to be so.

The **consolidated adoption of Hadoop,** together with the dramatic improvement of its available tools, makes it today often the main architectural requirement for solving many Big Data challenges. With SQL-on-Hadoop tools such as [Impala](http://blog.cloudera.com/blog/2012/10/cloudera-impala-real-time-queries-in-apache-hadoop-for-real/) or [Apache Drill](https://incubator.apache.org/drill/) it is no longer the case that data entering Hadoop is not rapidly actionable. Operational applications – e.g. involving recommendations, long-history analytics, client segmentation – benefit from a richer ecosystem, resulting in shorter development cycles; as well as improved hardware with better underlying software – therefore **shorter batch cycle times**. More frameworks to choose ([Spark](http://spark.incubator.apache.org/), [Stratosphere](http://stratosphere.eu/downloads/)) and future paradigms like [Tez](http://incubator.apache.org/projects/tez.html) (remember [Dryad](http://research.microsoft.com/en-us/projects/dryad/)?) complete the batch-processing scenario.

([The story of Spiderio’s architectural evolution came into my mind](http://www.slideshare.net/ashleywbrown/storm-at-spiderio-london-storm-meetup-20130618), switching from real-time to batch at some point.)

### Big Data: Real-time processing-only

Targeted to businesses where real-time is crucial, we are starting to see interesting real-time solutions such as [Druid](http://druid.io/). Stream processing and NoSQL-centric solutions remain a trend (remember [Facebook’s move to HBase](http://highscalability.com/blog/2011/3/22/facebooks-new-realtime-analytics-system-hbase-to-process-20.html)?). Storm improved its default API by adding [Trident](https://github.com/nathanmarz/storm/wiki/Trident-tutorial), which adds exactly-once semantics – and Google announced [Millwheel](http://research.google.com/pubs/pub41378.html), which, as usual, seems to take the state-of-the-art one step further. Add [Spark Streaming](https://www.cs.duke.edu/~kmoses/cps516/dstream.html) to the mix.

**Still challenging to think about the human fault-tolerancy** guarantees of such systems, their complexity and their overall price. And interesting to see how they kind of *meet in the middle*, **incorporating batch storage / indexing** sometimes as part of their architecture.

### “Unified” Lambda Architectures

What seemed once impossible, is starting to be achieved: developing Lambda Architectures using **a “unified” framework** which makes the same code available to both the real-time and batch layer, and combines the results of both layers transparently. Two tools are in the scope for achieving that:

* [**Summingbird**](https://speakerdeck.com/sritchie/summingbird-streaming-mapreduce-at-twitter): Based on “monoids”, supporting Hadoop and Storm as backing infrastructure. Used at Twitter with “Manhattan” as read-only store and Memcached as “real-time” store.
* [**Lambdoop**](http://www.slideshare.net/Datadopter/lambdoop-a-framework-for-easy-development-of-big-data-applications): Based on user-defined “operations”, Avro schemas, supports Hadoop and Storm as backing infrastructure and uses HBase as batch store and Redis as real-time store.

With Lambdoop not yet released and Summingbird being a very recent release, the usefulness of these frameworks is yet to be seen. **It might be the case that they are too complex or restrictive**, or that they evolve naturally into something very usable. By now the kind of operations that one can implement using these frameworks seem somehow restricted by the nature of the frameworks themselves, and the final data serving one can expect is restricted to pure **key/value serving**.

### “Free” Lambda Architectures

What still seems to me the most common case of Lambda Architecture, is a “literal” implementation of it where each layer operates independently and **results are merged depending on the business nature**.

When things like complex documents and **search-engines** come into play, it becomes harder to reason about an hypothetic “unified” framework that would provide the guarantees and semantics of a batch-processing layer and the immediateness of a speed layer. [The guys at Trovit made their system coordinate a batch and speed layer using Zookeeper](http://www.slideshare.net/sturlese/batch-andnrtsturlese).

As another example, at Datasalt we implemented [analytics for Ad Networks](http://www.datasalt.com/2014/01/ad-networks-analytics-using-hadoop-and-splout-sql/) using [Splout SQL](http://sploutsql.com/)as the batch read-only datastore. This allows us to precalculate less things and calculate real-time aggregations on-the-fly i.e. **ad-hoc drill-downs**. Tied with Splout, it is very easy to deploy a real-time SQL layer that compliments it using well-proven technologies such as MySQL, keeping in mind that the hardest problems are already solved by the batch layer.

I also remember [the guys at GameDuell somehow managed to mix Hadoop and VoltDB bi-directionally](http://www.youtube.com/watch?v=VVmSR--Yt5E&feature=youtu.be) with quick, few-minutes model re-calculations.

### The future

With things like [Yahoo’s Storm-Hadoop for YARN](http://www.infoq.com/news/2013/06/StormHadoop), and Big Data tools being nowadays easier and easier to install and manage (e.g. Cloudera Manager or Apache Ambari making life much easier) it will become less of a pain to deploy an open-source Big Data architecture, implement batch-processing and colocate it with real-time processing. What’s more, as stated, many cases will still be easily solvable just by batch processing. And as we have seen, newer real-time processing frameworks may already provide re-indexing / re-processing semantics underneath.

Because of the many possible use cases, data natures and serving requirements, it is unclear to me whether frameworks like Summingbird or Lambdoop will be key to implementing Lambda Architectures. But who knows, **maybe the best is yet to come!**

What constitutes a good architecture for real-time processing, and how do we select the right one for a project? In two blog posts we will discuss the qualities of the two popular choices Lambda and Kappa, and present concrete examples of use cases implemented using the respective approaches.

In this first post, we describe the two architectures in more detail: their differences, the technologies we can use to realize them as well as the tipping points that will make us decide on using one or the other.  
  
Over the past few years, there have been many discussions about how to design a good real-time data processing architecture. A good real-time data processing architecture needs to be fault-tolerant and scalable; it needs to support batch and incremental updates, and must be extensible.

One important milestone in these discussions was Nathan Marz, creator of Apache Storm, describing what we have come to know as the Lambda architecture. The Lambda architecture has proven to be relevant to many use-cases and is indeed used by a lot of companies, for example Yahoo and Netflix. But of course, Lambda is not a silver bullet and has received some fair criticism on the coding overhead it can create.

In the summer of 2014, Jay Kreps from LinkedIn posted an article describing what he called the Kappa architecture, which addresses some of the pitfalls associated with Lambda. Kappa is not a replacement for Lambda, though, as some use-cases deployed using the Lambda architecture cannot be migrated.

It can be challenging to accurately evaluate which architecture is best for a given use-case and making a wrong design decision can have serious consequences for the implementation of a data analytics project.  
Now let’s get into greater detail about the two data processing architectures.

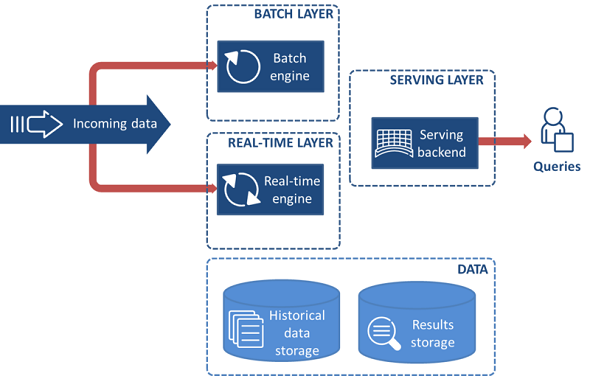


Figure 1 Lambda architecture

The Lambda Architecture, shown in Figure 1, is composed of three layers: batch, speed, and serving.

The batch layer has two major tasks: (a) managing historical data; and (b) recomputing results such as machine learning models. Specifically, the batch layer receives arriving data, combines it with historical data and recomputes results by iterating over the entire combined data set. The batch layer operates on the full data and thus allows the system to produce the most accurate results. However, the results come at the cost of high latency due to high computation time.

The speed layer is used in order to provide results in a low-latency, near real-time fashion. The speed layer receives the arriving data and performs incremental updates to the batch layer results. Thanks to the incremental algorithms implemented at the speed layer, computation cost is significantly reduced.

Finally, the serving layer enables various queries of the results sent from the batch and speed layers.

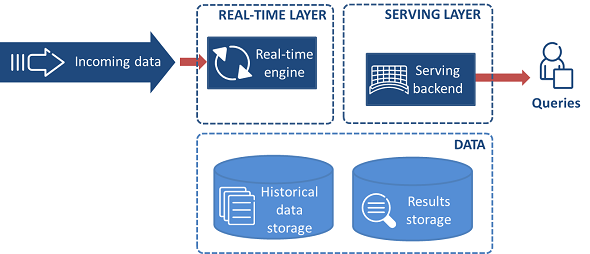


Figure 2 Kappa architecture

The Kappa architecture is shown in Figure 2. One of the important motivations for inventing the Kappa architecture was to avoid maintaining two separate code bases for the batch and speed layers. The key idea is to handle both real-time data processing and continuous data reprocessing using a single stream processing engine. Data reprocessing is an important requirement for making visible the effects of code changes on the results. As a consequence, the Kappa architecture is composed of only two layers: stream processing and serving. The stream processing layer runs the stream processing jobs. Normally, a single stream processing job is run to enable real-time data processing. Data reprocessing is only done when some code of the stream processing job needs to be modified. This is achieved by running another modified stream processing job and replying all previous data. Finally, similarly to the Lambda architecture, the serving layer is used to query the results.

The two architectures can be implemented by combining various open-source technologies, such as Apache Kafka, Apache HBase, Apache Hadoop (HDFS, MapReduce), Apache Spark, Apache Drill, Spark Streaming, Apache Storm, and Apache Samza.

For example, data can be ingested into the Lambda and Kappa architectures using a publish-subscribe messaging system, for example Apache Kafka. The data and model storage can be implemented using persistent storage, like HDFS. A high-latency batch system such as Hadoop MapReduce can be used in the batch layer of the Lambda architecture to train models from scratch. Low-latency systems, for instance Apache Storm, Apache Samza, and Spark Streaming can be used to implement incremental model updates in the speed layer. The same technologies can be used to implement the stream processing layer in the Kappa architecture.

Alternatively, Apache Spark can be used as a common platform to develop the batch and speed layers in the Lambda architecture. This way, much of the code can be shared between the batch and speed layers. The serving layer can be implemented using a NoSQL database, such as Apache HBase, and an SQL query engine like Apache Drill.

So when should we use one architecture or the other? As is often the case, it depends on some characteristics of the application that is to be implemented. Let’s go through a few common examples:

A very simple case to consider is when the algorithms applied to the real-time data and to the historical data are identical. Then it is clearly very beneficial to use the same code base to process historical and real-time data, and therefore to implement the use-case using the Kappa architecture.

Now, the algorithms used to process historical data and real-time data are not always identical. In some cases, the batch algorithm can be optimized thanks to the fact that it has access to the complete historical dataset, and then outperform the implementation of the real-time algorithm. Here, choosing between Lambda and Kappa becomes a choice between favoring batch execution performance over code base simplicity.

Finally, there are even more complex use-cases, in which even the outputs of the real-time and batch algorithm are different. For example, a machine learning application where generation of the batch model requires so much time and resources that the best result achievable in real-time is computing and approximated updates of that model. In such cases, the batch and real-time layers cannot be merged, and the Lambda architecture must be used.

In our previous blog post, we briefly described two popular data processing architectures: Lambda architecture and Kappa architecture. In this post, we present two concrete example applications for the respective architectures: Movie recommendations and Human Mobility Analytics.

#### **Movie recommendations**

Movie recommender systems are an important part of modern media delivery platforms, as they enable a personalized service experience by suggesting relevant movies to users. The ability to deliver accurate and diversified recommendations on time is key for user retention, and thus for revenue generation.

Movie recommender systems typically base their recommendations on a combination of implicit and explicit feedback collected from users. Examples of implicit feedback are clicks, movie views, and location. Explicit feedback is typically collected in the form of movie ratings. Predictions are used to generate lists of personalized movie recommendations.

The design and implementation of a movie recommender system is a challenging task, since there needs to be a balance between accuracy and responsiveness. Accuracy is important to really make the movie predictions relevant. Responsiveness is important in order to provide near real-time recommendations when the users interact with the movie delivery platform.

To enable accurate and responsive movie recommendations, scalable prediction methods are needed. Moreover, such methods need to be implemented within a scalable data processing system. Collaborative filtering approaches via Matrix Factorization (MF) have shown to produce good results in generating predictions at scale, read more [here](http://spark.apache.org/docs/latest/mllib-collaborative-filtering.html). Production systems implementing collaborative filtering based on MF typically have the following requirements:

* Training of large MF models from scratch.
* Incremental updating of MF models.

The former requirement is necessary in order to train the initial MF model. The latter requirement is necessary to provide responsiveness to the arrival of new user preferences, e.g., ratings, clicks. Training from scratch for every new user preference would cause significant computation cost. Algorithms such as [Alternating Least Squares](http://dx.doi.org/10.1109/MC.2009.263) (ALS) can be used to train MF models from scratch in a distributed fashion, and other algorithms exist to incrementally update MF models.

The Lambda Architecture is a good candidate to build a MF-based recommender system, because it fulfills two important requirements: (a) a batch layer for initial model training; and (b) incremental updates via the speed layer. A batch layer enables accurate predictions while the speed layer allows for real-time updating, which is key to responsiveness. The algorithms used in the batch and real-time layer are different, which prevents us from using the same codebase.

This year we have implemented a movie recommender system for one of our personalization projects, using the Lambda architecture. Implementing the Lambda architecture is known to be a non-trivial task, as it requires the integration of several complex distributed systems, like Apache Kafka, Apache HDFS, or Apache Spark; as well as machine learning libraries, for example Apache Mahout or Spark MLlib. We have therefore tried to reuse as much code as possible.

[Cloudera Oryx](http://oryx.io/) is an existing open-source implementation of the Lambda architecture that we decided on adopting. Oryx is based on several technologies that we were already using, including Apache Kafka and Apache Spark. Moreover, it is designed to serve as a framework for implementing machine learning applications with real-time requirements. Actually, a movie recommender is one such application that is shipped with Oryx. The Oryx architecture, shown in Figure 1, is based on four layers: data transport, batch, speed, and serving.

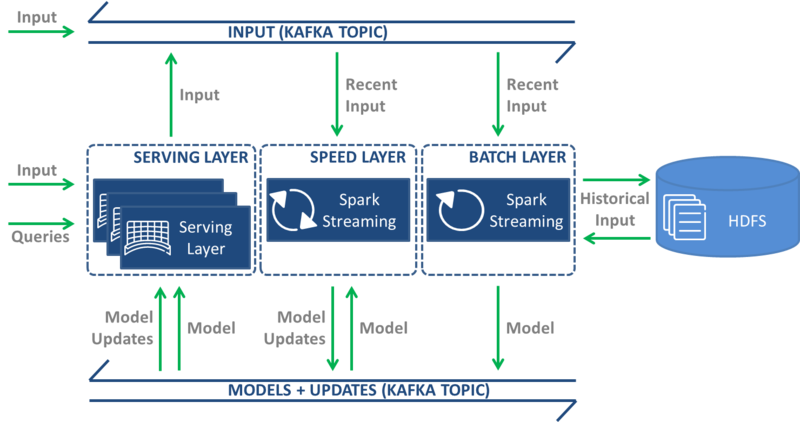


Figure 1 Cloudera Oryx architecture.

The data transport layer receives and moves data between the layers. Two types of data are moved in the movie recommender application: (1) user-movie preferences; and (2) initial MF model and MF model updates. The data transport layer is implemented using the Apache Kafka publish-subscribe messaging system. Apache HDFS is used to persist the MF model in PMML format along with all historical data (tuples of user id, movie id, rating, and timestamp).

The batch layer uses a Spark Streaming job with a very long time interval (hours). It saves the data captured in the most recent time interval to HDFS, merges it with the historical data on HDFS, and starts the MF model building process. For the movie recommender application, the ALS algorithm of [Apache Spark MLlib](http://spark.apache.org/docs/latest/mllib-collaborative-filtering.html) is used to build the MF model. The resulting model is stored on HDFS and published on the data transport layer.

The speed layer uses a Spark Streaming job with a very short time interval (seconds). It receives the full model from the data transport layer and performs incremental/online model updates as new data arrives. For example, as users rate movies, the ratings are instantly incorporated into the model, thus allowing near real-time recommendations. The updated model is published on the data transport layer.

Finally, the serving layer is implemented using an embedded web server. The speed layer receives models – initial and updates – over the data transport layer and stores them in-memory. A REST API is provided to interact with the system. Two types of interactions are supported to ingest data and consume models. Examples of data ingestion include new user-movie preferences, and examples of model consumption include model queries such as the N most popular movies.

#### **Human mobility analytics**

As a second example, we will now look at a use-case developed a while back in Ericsson Research, called real-time human mobility analytics (rtHMA). This use-case is built around the idea that mobile networks generate a lot of location tagged data, which can be mined to provide high-level patterns of how people move around in a city or country.

Data typically consists of a timestamp, a hashed identifier, and a location identifier that points to a mobile network cell. The location is not as precise as GPS coordinates, but sufficient for an approximate position – airport, particular suburb, stadium, theme park, shopping mall, and so on.

What rtHMA does is consume this data in real-time and output resulting movement patterns in the form of Origin/Destination matrices.

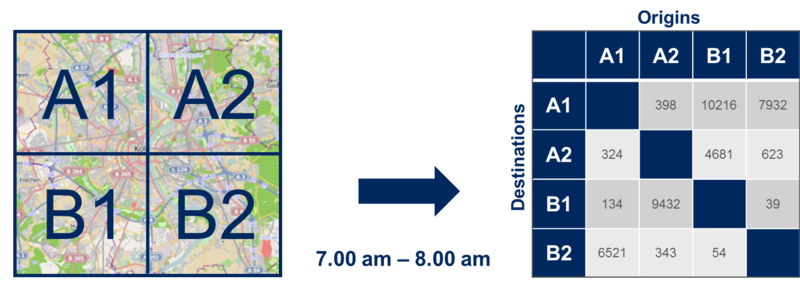


Figure 2 Example of Origin/Destination Matrix.

An Origin/Destination (O/D) matrix is a tool used by some industries, for example transportation, to model mobility demand at city or country level. In each row and column of the matrix, we have the complete list of the locations we want to study – airport, suburb A, suburb B, and so on – and in each cell of the matrix, we have the number of people who traveled from the corresponding column location to the corresponding row location during a specified period of time. Figure 2 shows a city split in 4 different locations: A1, A2, B1 and B2. By reading the O/D matrix on the right side, we can see that 324 people traveled from location A1 to location A2 between 7AM and 8AM.

The algorithm that parses trips out of a stream of location data has been implemented in an Apache Storm topology. Data is fed into the topology in real-time through a TCP socket (a message bus like Apache Kafka could also have been used). Once the trips are parsed, results are saved into a relational database (Postgres) and can be queried using a REST API that we implemented in Python using Django.

This whole pipeline is also able to analyze historical data. To do so, the only step needed is to have historical data in one or several files, and to pipe the content of these files into a netcat instance pointing at the Storm topology socket. From the Storm topology perspective, there is no difference between historical and real-time data. If historical data is replayed, historical data results are overwritten in the database. If Kafka had been used, similar results would have been obtained by having the topology process all retained data of a pre-defined topic.

Before the aforementioned Storm topology was implemented, we actually had an earlier implementation of the algorithm in Apache Hadoop. The process was similar except that data had to be loaded from HDFS. When implementing the Storm version, our initial plan was to keep the Hadoop implementation when working with stored data, as opposed to real-time streams. However, once we realized that the Storm implementation was capable of processing historical data just as fast as the Hadoop one, we simply deprecated it and only kept Storm as a pure Kappa implementation of the use-case.

In this blog post we have presented two example applications for Lambda and Kappa architectures, respectively. As can be seen from our discussion, there is no one-size-fits-all solution for all applications. The movie recommender application clearly benefits from having batch and speed layers in order to achieve batch and incremental model training. This is natural as different algorithms are used for the two layers – training from scratch in the batch setting and incremental training in the speed layer. On the other hand, the rtHMA application does not require distinct algorithms, and hence can be easily implemented using a single layer. We can conclude that the big data processing architecture choice is application dependent and needs to be well thought through.

# [kappa-architecture.com](http://milinda.pathirage.org/kappa-architecture.com/)

Repository dedicated to Kappa Architecture. I collect and publish articles, tutorials, talks, projects and examples related to Kappa Architecture.

## What is Kappa Architecture?

Kappa Architecture is a software architecture pattern. Rather than using a relational DB like SQL or a key-value store like Cassandra, the canonical data store in a Kappa Architecture system is an append-only immutable log. From the log, data is streamed through a computational system and fed into auxiliary stores for serving.

Kappa Architecture is a simplification of [Lambda Architecture](https://en.wikipedia.org/wiki/Lambda_architecture). A Kappa Architecture system is like a Lambda Architecture system with the batch processing system removed. To replace batch processing, data is simply fed through the streaming system quickly.

## But why?

Kappa Architecture revolutionizes database migrations and reorganizations: just delete your serving layer database and populate a new copy from the canonical store! Since there is no batch processing layer, only one set of code needs to be maintained.

## Says who?

The idea of Kappa Architecture was first described in an [article](http://radar.oreilly.com/2014/07/questioning-the-lambda-architecture.html) by [Jay Kreps](https://www.linkedin.com/in/jaykreps) from LinkedIn. Then came the talk [“Turning the database inside out with Apache Samza”](https://www.youtube.com/watch?v=fU9hR3kiOK0) by [Martin Kleppmann](http://martin.kleppmann.com/) at 2014 [StrangeLoop](https://thestrangeloop.com/) which inspired this web site.

### TURNING THE DATABASE INSIDE OUT WITH APACHE SAMZA

### RESOURCES

* [Questioning the Lambda Architecture](https://www.oreilly.com/ideas/questioning-the-lambda-architecture)
* [Apache Kafka and the Next 700 Stream Processing Systems](https://www.youtube.com/watch?v=9RMOc0SwRro)
* Article by Jay Kreps: [The Log: What every software engineer should know about real-time data’s unifying abstraction](https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying)
* Presentation: [Discovering Kappa Architecture the hard way](http://novoj.github.io/reveal.js/kappa-architecture.html#/)
* Linux Foundation Presentation: [Kappa Architecture: Our Experience](http://events.linuxfoundation.org/sites/events/files/slides/ASPgems%20-%20Kappa%20Architecture.pdf)
* [Liquid: Unifying Nearline and Offline Big Data Integration](http://www.cidrdb.org/cidr2015/Papers/CIDR15_Paper25u.pdf) (Summary of Liquid paper can be found [here](http://blog.acolyer.org/2015/02/04/liquid-unifying-nearline-and-offline-big-data-integration/).)

## Tools

### LOG DATA STORES

An append-only immutable log store is the canonical store in a Kappa Architecture (or Lambda Architecture) system. Some log databases:

* [Apache Kafka](http://kafka.apache.org/)
* [DistributedLog](http://distributedlog.io/)

### STREAMING COMPUTATION SYSTEMS

In Kappa Architecture, data is fed from the log store into a streaming computation system. Some distributed streaming systems:

* [Apache Samza](http://samza.apache.org/)
* [Apache Storm](http://storm.apache.org/)
* [Apache Spark](http://spark.apache.org/)
* [Amazon Kinesis](https://aws.amazon.com/kinesis/)
* [Kafka Streams](http://kafka.apache.org/documentation.html#streams)
* [Apache Flink](https://flink.apache.org/)
* [Onyx](http://www.onyxplatform.org/)
* [Hazelcast Jet](http://jet.hazelcast.org/)

### SERVING LAYER STORES

The purpose of the serving layer is to provide optimized responses to queries. These databases aren’t used as canonical stores: at any point, you can wipe them and regenerate them from the canonical data store. Almost any database, in-memory or persistent, might be used in the serving layer. This also includes special-purpose databases, e.g. for full text search.

# **From Lambda to Kappa: A Guide on Real-time Big Data Architectures**

When it comes to real-time big data architectures, today… there are choices. Today, there is more than just Lambda on the menu of choices, and in this blog series, I’ll discuss a couple of these choices and compare them using relevant use cases.  So, how do you select the right architecture for our real-time project? Let’s get started.

[Download >> Talend Open Studio for Data Integration](https://www.talend.com/download/talend-open-studio/?qt-product_tos_download_new=1&utm_medium=bloglink&utm_source=talend&utm_campaign=bloglink)

## **Real-Time Requirements**

Before we dive into the architecture, let’s discuss some of the requirements of real-time data processing systems in big data scenarios.

The most obvious of these requirements is that data is in motion.  In other words, the data is continuous and unbounded.  It’s really about **when** you are analyzing this data that matters. If you are looking for answers against the current snapshot of data or have specific low-latency requirements, then you’re probably looking at a  real-time scenario.

In addition, there are very often business deadlines to be met. After all, if there were no consequences to missing deadlines for real-time analysis, then the process could be batched. These consequences can range from complete failure to simply degradation of service.

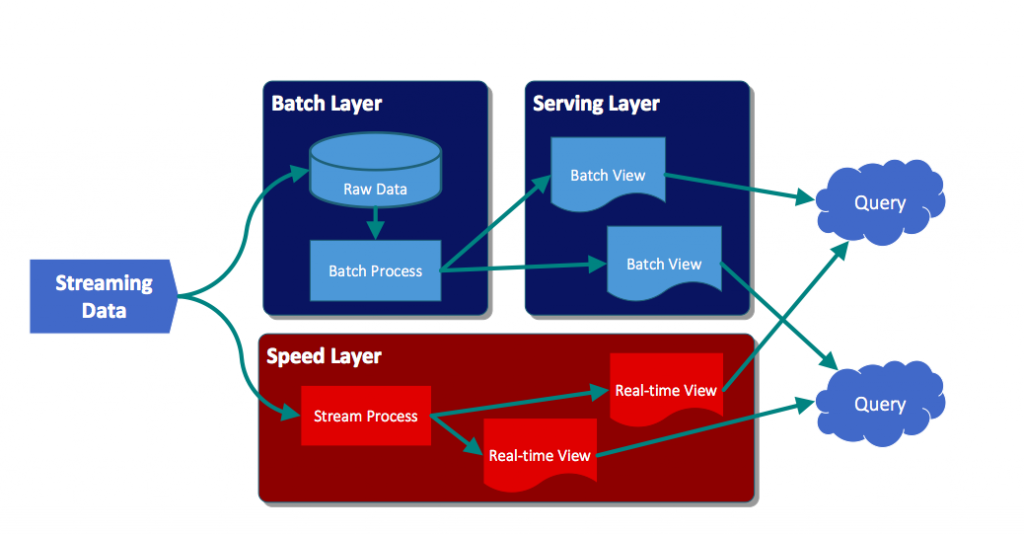
Since we are talking about big data, we also expect to push the limits on volume, velocity and possibly even variety of data.

Real-time data processing often requires qualities such as scalability, fault-tolerant, predictability, resiliency against stream imperfections, and must be extensible.

## **New Architectures for the New Data Era**

To address this need, new architectures were born… or in other words, necessity is the mother of invention.

The Lambda Architecture, [attributed to Nathan Marz](https://www.manning.com/books/big-data), is one of the more common architectures you will see in real-time data processing today.  It is designed to handle low-latency reads and updates in a linearly scalable and fault-tolerant way.



The data stream entering the system is dual fed into both a **batch and speed layer**.

The **batch laye**r stores the raw data as it arrives, and computes the batch views for consumption. Naturally, batch processes will occur on some interval and will be long-lived. The scope of data is anywhere from hours to years.

The **speed layer** is used to compute the real-time views to compliment the batch views.

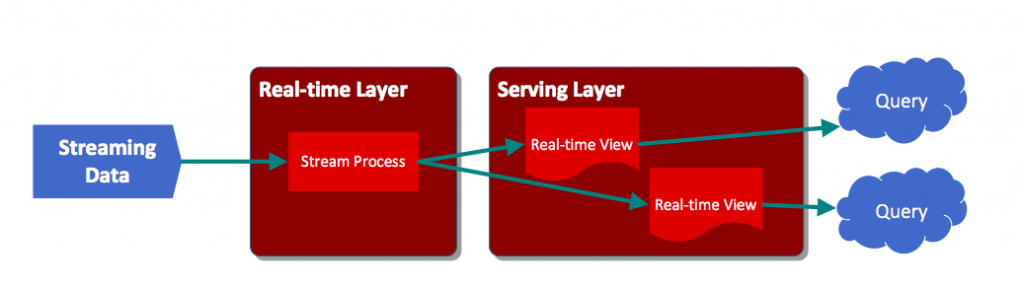
Any query may get a complete picture by retrieving data from both the batch views and the real-time views.  The queries will get the best of both worlds.  The batch views may be processed with more complex or expensive rules and may have better data quality and less skew, while the real-time views give you up to the moment access to the latest possible data. As time goes on, real-time data expires and are replaced with data in the batch views.

One additional benefit to this architecture is that you can replay the same incoming data and produce new views in case code or formula changes.

The biggest detraction to this architecture has been the need to maintain two distinct (and possibly complex) systems to generate both batch and speed layers. Luckily with Spark Streaming (abstraction layer) or Talend (Spark Batch and Streaming code generator), this has become far less of an issue… although the operational burden still exists.

Next, we’ll discuss the **Kappa Architecture**.

The [Kappa Architecture was first described by Jay Kreps](http://milinda.pathirage.org/kappa-architecture.com/).  It focuses on only processing data as a stream. It is not a replacement for the Lambda Architecture, except for where your use case fits. For this architecture, incoming data is streamed through a real-time layer and the results of which are placed in the serving layer for queries.



The idea is to handle both real-time data processing and continuous reprocessing in a single stream processing engine. That’s right, reprocessing occurs from the stream. This requires that the incoming data stream can be replayed (very quickly), either in its entirety or from a specific position.  If there are any code changes, then a second stream process would replay all previous data through the latest real-time engine and replace the data stored in the serving layer.

This architecture attempts to simplify by only keeping one code base rather than manage one for each batch and speed layers in the Lambda Architecture. In addition, queries only need to look in a single serving location instead of going against batch and speed views.

The complication of this architecture mostly revolves around having to process this data in a stream, such as handling duplicate events, cross-referencing events or maintaining order- operations that are generally easier to do in batch processing.

## **One Size May Not Fit All**

Many real-time use cases will fit a Lambda architecture well.  The same cannot be said of the Kappa Architecture. If the batch and streaming analysis are identical, then using Kappa is likely the best solution. In some cases, however, having access to a complete set of data in a batch window may yield certain optimizations that would make Lambda better performing and perhaps even simpler to implement.

There are also some very complex situations where the batch and streaming algorithms produce very different results (using machine learning models, expert systems, or inherently very expensive operations that must be performed differently in real-time) which would require using Lambda.

So, that covers the two most popular real-time data processing architectures.  The next articles in this series will dive deeper into each of these and we’ll discuss concrete use cases and the technologies that would often be found in these architectures.

<http://lambda-architecture.net/platforms/2013-12-25-lambdoop>

<http://lambda-architecture.net/architecture/2013-12-25-issues-in-combined-static-and-dynamic-data-management>

<http://lambda-architecture.net/architecture/2013-12-11-a-real-time-architecture-using-hadoop-and-storm-devoxx>

<http://milinda.pathirage.org/kappa-architecture.com/>

<http://novoj.github.io/reveal.js/kappa-architecture.html#/>

<http://events.linuxfoundation.org/sites/events/files/slides/ASPgems%20-%20Kappa%20Architecture.pdf>