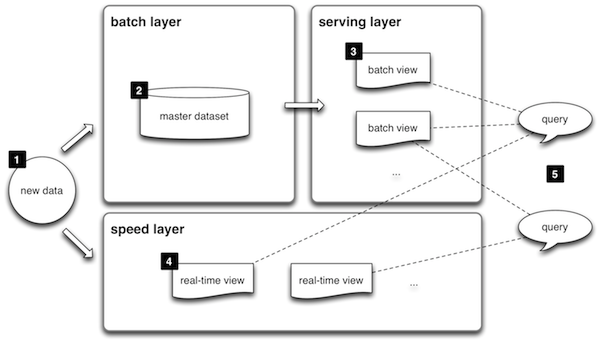
**Why? Problem?**

What is need of the Lambda architecture?

Need to process batch data and real time data the same time.



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.

**Lambda architecture**

**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) 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. The rise of lambda architecture is correlated with the growth of [big data](https://en.wikipedia.org/wiki/Big_data), real-time analytics, and the drive to mitigate the latencies of [map-reduce](https://en.wikipedia.org/wiki/Map-reduce).

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.

**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.[[3]](https://en.wikipedia.org/wiki/Lambda_architecture#cite_note-big-data-3):13 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.

**Speed layer**

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.

**Serving layer**]

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) or [Apache HBase](https://en.wikipedia.org/wiki/Apache_HBase) for speed-layer output, and [Elephant DB](https://github.com/nathanmarz/elephantdb) or [Cloudera Impala](https://en.wikipedia.org/wiki/Cloudera_Impala" \o "Cloudera Impala) for batch-layer output.[

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.

**Lambda architecture in use**

Metamarkets, which provides analytics for companies in the programmatic advertising space, employs a version of the lambda architecture that uses [Druid](https://en.wikipedia.org/wiki/Druid_(open-source_data_store)) for storing and serving both the streamed and batch-processed data.

For running analytics on its advertising data warehouse, [Yahoo](https://en.wikipedia.org/wiki/Yahoo) has taken a similar approach, also using [Apache Storm](https://en.wikipedia.org/wiki/Storm_(event_processor)), [Apache Hadoop](https://en.wikipedia.org/wiki/Hadoop), and [Druid](https://en.wikipedia.org/wiki/Druid_(open-source_data_store)).

The [Netflix](https://en.wikipedia.org/wiki/Netflix) Suro project has separate processing paths for data, but does not strictly follow lambda architecture since the paths may be intended to serve different purposes and not necessarily to provide the same type of views. Nevertheless, the overall idea is to make selected real-time event data available to queries with very low latency, while the entire data set is also processed via a batch pipeline. The latter is intended for applications that are less sensitive to latency and require a map-reduce type of processing.

**Criticism**

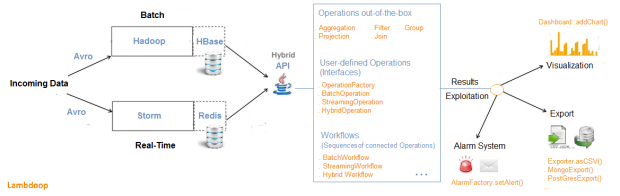
Criticism of lambda architecture has focused on its inherent complexity and its limiting influence. The batch and streaming sides each require a different code base that must be maintained and kept in sync so that processed data produces the same result from both paths. Yet attempting to abstract the code bases into a single framework puts many of the specialized tools in the batch and real-time ecosystems out of reach.

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. Such a streaming framework could allow for collecting and processing arbitrarily large windows of data, accommodate blocking, and handle state.

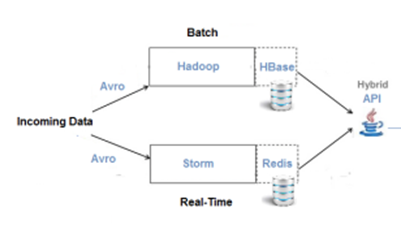
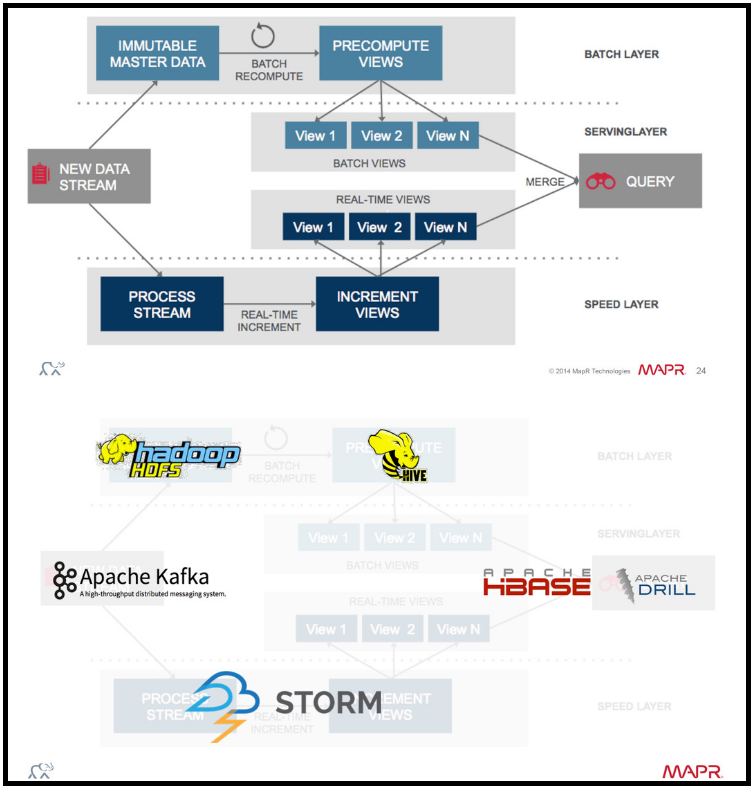
### Unified Lambda

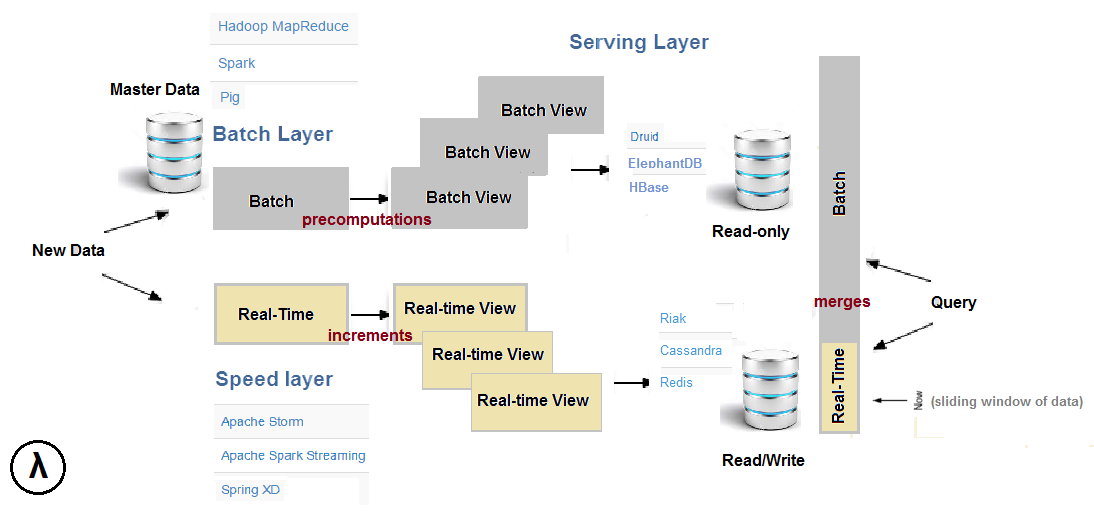
The downside of **λ** is its inherent **complexity**. Keeping  in sync two already complex distributed systems is quite an implementation and maintenance challenge. People have started to look for simpler alternatives that would bring just about the same benefits and handle the full problem set. There are basically three approaches:

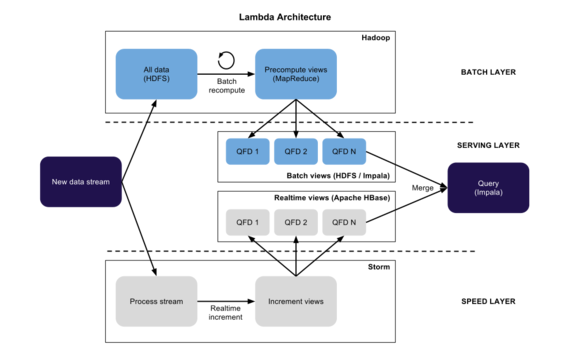
1. Adopt a pure streaming approach, and use a flexible framework such as [Apache Samza](http://samza.incubator.apache.org/) to provide some type of batch processing. Although its distributed streaming layer is pluggable, Samza typically relies on [Apache Kafka](http://kafka.apache.org/). Samza’s streams are replayable, ordered partitions. **Samza** can be configured for batching, i.e. consume several messages from the same stream partition in sequence.
2. Take the opposite approach, and choose a flexible Batch framework that would also allow micro-batches, small enough to be close to real-time, with Apache Spark/Spark Streaming or Storm’s [Trident](https://storm.apache.org/documentation/Trident-tutorial.html). Spark streaming is essentially a sequence of small batch processes that can reach latency as low as one second. Trident is a high-level abstraction on top of Storm that can process streams as small batches as well as do batch aggregation.
3. Use a technology stack already combining batch and real-time, such as Spring “XD”, [Summingbird](https://github.com/twitter/summingbird) or [Lambdoop](http://lambdoop.com/). Summingbird (“Streaming MapReduce”) is a hybrid system where both batch/real-time workflows can be run at the same time and the results merged automatically. The Speed layer runs on Storm and the Batch layer on Hadoop, Lambdoop (**Lamb**da-Ha**doop**, with [HBase](http://hbase.apache.org/), Storm and [Redis](http://redis.io/)) also combines batch/real-time by offering a single API for both processing paradigms:

[](https://tsicilian.files.wordpress.com/2015/01/lambdoop2.png)

The integrated approach (**unified λ**) seeks to handle Big Data’s Volume and Velocity by featuring a **hybrid computation model,**where both batch and real-time data processing are combined transparently. And with a unified framework, there would be only one system to learn, and one system to maintain.







**Comparision between Habse , Cassandara, MongoDB**

**HBase:**

Key characteristics:  
·         Distributed and scalable big data store  
·         Strong consistency  
·         Built on top of Hadoop HDFS  
·         CP on CAP

Good for:  
·         Optimized for read  
·         Well suited for range based scan  
·         Strict consistency  
·         Fast read and write with scalability

Not good for:  
·         Classic transactional applications or even relational analytics  
·         Applications need full table scan  
·         Data to be aggregated, rolled up, analyzed cross rows

Usage Case: Facebook message

**Cassandra:**

Key characteristics:  
·         High availability  
·         Incremental scalability  
·         Eventually consistent  
·         Trade-offs between consistency and latency  
·         Minimal administration  
·         No SPF (Single point of failure) – all nodes are the same in Cassandra  
·         AP on CAP

Good for:  
·         Simple setup, maintenance code  
·         Fast random read/write  
·         Flexible parsing/wide column requirement  
·         No multiple secondary index needed

Not good for:  
·         Secondary index  
·         Relational data  
·         Transactional operations (Rollback, Commit)  
·         Primary & Financial record  
·         Stringent and authorization needed on data  
·         Dynamic queries/searching  on column data  
·         Low latency

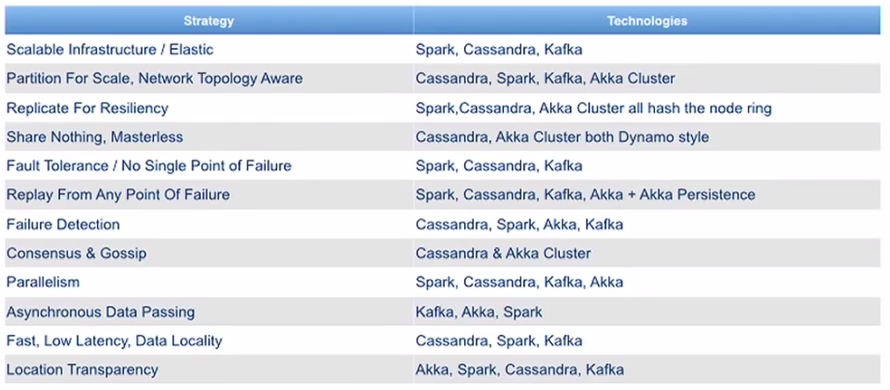
Usage Case: Twitter, Travel portal

**MongoDB:**

Key characteristics:  
·         Schemas to change as applications evolve (Schema-free)  
·         Full index support for high performance  
·         Replication and failover for high availability  
·         Auto Sharding for easy Scalability  
·         Rich document based queries for easy readability  
·         Master-slave model  
·         CP on CAP

Good for:  
·         RDBMS replacement for web applications  
·         Semi-structured content management  
·         Real-time analytics and high-speed logging, caching and high scalability  
·         Web 2.0, Media, SAAS, Gaming

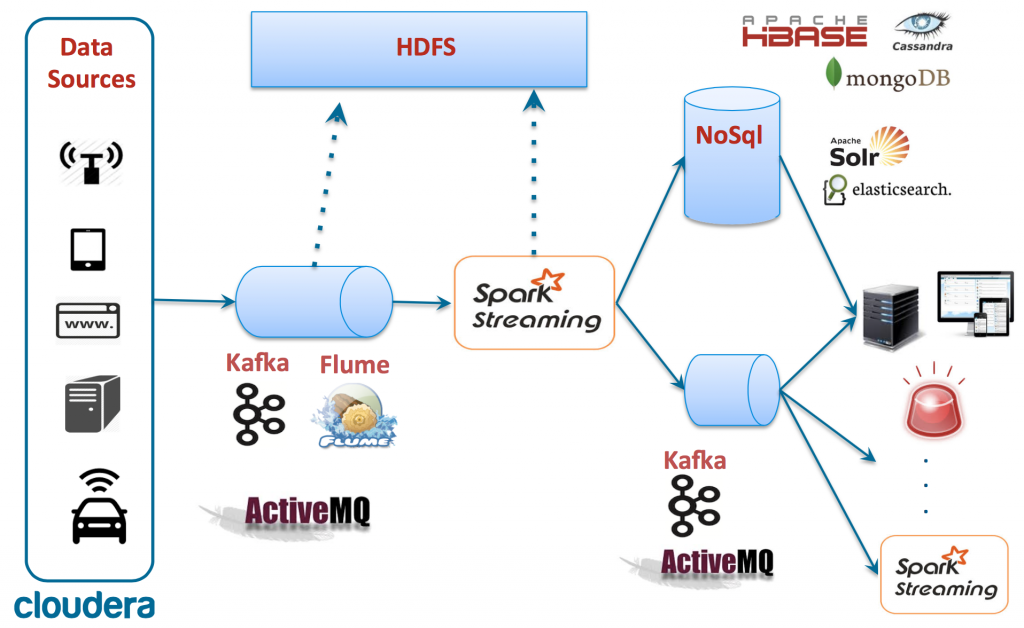
Not good for:  
·         Highly transactional system  
·         Applications with traditional database requirements such as foreign key constraints



## Canonical stream processing architecture

The increasing demand for RT processing requires systems that enables it. A canonical RT/ stream processing architecture is typically made of a following components.

* **Data sources** – source of data streams. This is the raw data and so the source of truth. A data source could be a sensor network, or a mobile application, or a web client, or a log from a server, or even a thing from Internet of Things.
* **Message bus** – reliable, high-throughput and low latency messaging system. [Kafka](http://kafka.apache.org/) and [Flume](http://flume.apache.org/) are obvious choices. There are many other options as well, like, [ActiveMQ](http://activemq.apache.org/), [RabbitMQ](https://www.rabbitmq.com/), etc. [Kafka](http://kafka.apache.org/) is definitely gaining lots of popularity right now.
* **Stream processing system** – a computation framework capable of doing computations on data streams. There a few stream processing frameworks out there, [Spark Streaming](https://spark.apache.org/streaming/), [Storm](https://storm.apache.org/), [Samza](http://samza.apache.org/), [Flink](https://flink.apache.org/), etc. Spark Streaming is probably the most popular framework for stream processing right now.
* **NoSql store** – processed data is of no use if it does not serve end applications or users. End applications like to do lots of fast read and writes. [HBase](http://hbase.apache.org/), [Cassandra](http://cassandra.apache.org/), [MongoDb](https://www.mongodb.org/) are popular choices.
* **End applications** – these are the application which consume the processed data/result streams.

[](http://ingest.tips/wp-content/uploads/2015/06/Screen-Shot-2015-06-14-at-11.12.19-AM.png)