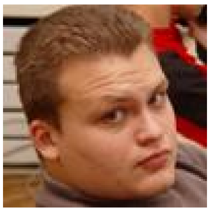
LAMBDA + REACTIVE = CREATIVE



Roman Novik

Big Data Competency Center Expert

Big Data Competency Center Expert,EPAM, raman_novik@epam.com

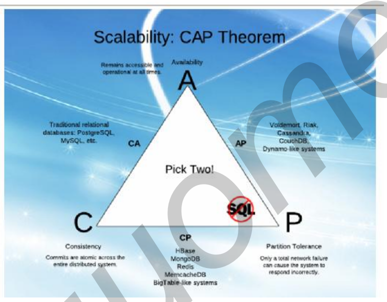
AGENDA

- Motivation
- How to beat the CAP theorem
- Lambda Approach
- Reactive Approach
- Architecture (Lambda, real-world, services, roles, cluster)

CAP THEOREM

by Eric Brewer (Berkley)

A database cannot guarantee consistency, availability, and partition-tolerance at the same time!



AVAILABILITY OR CONSISTENCY?

- We can't sacrifice partition-tolerance as we are talking about distributed and Big Data systems
- So we must make a tradeoff between availability and consistency
- Managing this tradeoff is a central focus of the NoSQL movement
- Consistency = after a successful write, future reads will always take that write into account
- Availability = ability to always read and write to the system
- During a partition, you can only have one of these properties!

CONSISTENCY?

- hmm… There are a lot of awkward issues in case when a database isn't available:
- Buffering writes on some middle machine?
  - There is a risk to lose buffer if middle machine will fail
  - Some inconsistency because of client thinks that data was already committed to database
- Return errors back to client?
  - It's really unsatisfied user experience!

AVAILABILITY?

Eventual consistency - is really "painful" thing to deal with

•Sometimes it's possible to read different result than was written

•Sometimes multiple readers can get different result by the same key

•Updates may not propagate to all replicas of a value

•Difficult strategies like "read repair", "vector clocks" are hard to implement, maintain, and are extremely susceptible to developer's errors

YOU'RE DAMNED IF YOU DO AND DAMNED IF YOU DON'T

Sacrificing consistency = poor user experience and problems with database unavailability

Sacrificing availability = problems with eventual consistency

The CAP theorem is a fact of nature!

IS THERE NO WAY OUT?

There is another way!

Two problems stand out in particular:
- the use of mutable state in databases
- the use of incremental algorithms to update that state

We can't avoid the CAP theorem, but we can isolate its complexity!

ELIMINATION OF THE RESTRICTIONS

- We will try to design new type of distributed data system:
  - it will eliminate the restrictions of the CAP theorem
  - it will be fault-tolerant to machine failures
- But we won't stop there:
  - let's make this data system human fault-tolerant!

WHAT IS A DATA SYSTEM?

- The problem we're trying to solve:
  - what is the purpose of a data system?
  - what is data?
- However, there is such a simple definition:

Query = Function(All Data)

"DATA"

- A piece of data is an indivisible unit that you hold to be true
- It's like an axiom in mathematics
- There are two crucial properties of data
  - data is inherently time based

    Nick / @timestamp1 / lives in Minsk
    Nick / @timestamp2 / lives in Moscow

  - data is inherently immutable

WHAT ABOUT CRUD?

- Do we really need CRUD?
- There only two main operations we can do with data:
  - read existing data
  - add more data
- So, let's turn CRUD to CR!
  - updates don't make sense with immutable data
    Nick / @timestamp1 / lives in Minsk
    Nick / @timestamp2 / lives in Moscow
  - deletes don't make sense with immutable data
    Nick / @timestamp1 / follows Mary
    Nick / @timestamp2 / unfollowed Mary
- Still, purging (or compaction, or "garbage collection") is not a problem in this scenario!

"QUERY"

- It's is a derivation from a set of data
- It's like a theorem in mathematics
- For example:
  - data
    Nick / @timestamp1 / lives in Minsk
    Nick / @timestamp2 / lives in Moscow
  - query
    What is Nick's current location? => Moscow

HOW TO BEAT THE CAP THEOREM

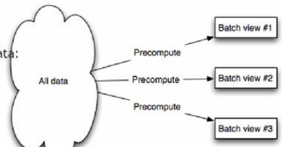
- If we could query the complete dataset within our latency constraints
  - then there would be nothing else to invent
- If not, the CAP theorem still applies
- But the complexity it normally causes is avoided
  - by using immutable data
  - and computing queries from scratch
- If we choose consistency over availability - then not much changes from before
  - periodical inaccessibility of system is still possible
  - but it is option where rigid consistency is a necessity
- If we choose availability over consistency - then the system is eventually consistent without any of the complexities of eventual consistency
  - we always write new data
  - queries always work with fresh data
  - there are no divergent values, "repair reads", "vector clocks"

SUMMARY

- Problem was around the interaction between incremental updates and the CAP theorem
- We can avoid that complexity
  - by rejecting incremental updates
  - by embracing immutable data
  - and computing queries from scratch each time
- Of course, it was just our assumption
  - it's infeasible to compute queries from scratch each time
  - but we found some key properties of what a real solution will look like
- These properties are
  - the system makes it easy to sore and scale an immutable, constantly-growing dataset
  - the primary operation of the system is to add new immutable facts of data
  - the system recomputes queries from raw data
  - the system can use incremental algorithms if latency of such queries is on acceptable level

BATCH COMPUTATIONS

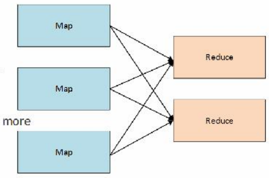
- It's daunting problem to make a some function on whole dataset
- Let's work with outdated (for a few hours) data
- Let's precompute data
- For example, to have latest state of immutable data:
  Nick / @timestamp1 / lives in Minsk
  Nick / @timestamp2 / lives in Moscow
  - becomes
  Nick / @timestamp2 / lives in Moscow
- To build such system we need system that
  - can easily store a large and constantly growing dataset
  - can compute functions on that whole dataset in a scalable way



Precomputation workflow

HADOOP & MAPREDUCE

- Hadoop is exactly what we need!
- Its components:
  - HDFS - distributed fault-tolerance file system
  - Yarn - yet another resource manager
  - Hive - SQL-like façade
  - Parquet, ORC, Avro - data-formats with schema
  - HBase - NoSQL key-value versioned database
  - eco-system (data-governance, security, ETLs etc)
  - MapReduce - default batch processing framework
- MapReduce:
  - programming interface so that the system can do more automatically
  - express jobs as graphs of high-level operators

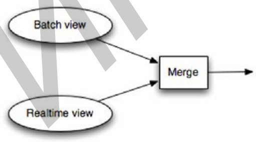


REAL-TIME COMPUTATIONS

We need a real-time system to be launched in parallel with batch system

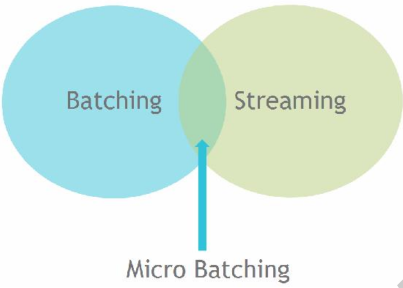
This real-time system will precompute each query function for the last few hours of data

To resolve result query batch and real-time views and merge all results



Computing a query

BATCHING VS STREAMING



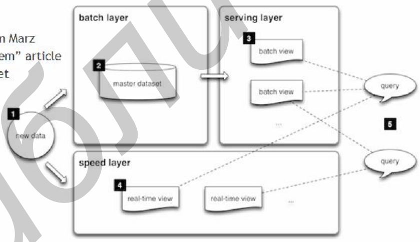
SPARK VS STORM

	Spark Streaming	Core Storm
Hadoop distribution	Hortonworks, Cloudera, MapR	Hortonworks
Implemented in	Scala	Clojure (Lisp like on JVM)
API language	Java, Scala, Python	Java, Scala, Clojure, Python, Ruby
Stack	Spark SQL & Hive integration, Spark MLLib, Spark GraphX	N/A
Processing model	Micro-batching	Record-at-a-time
Coordinator	Zookeeper	Zookeeper
Resource manager	Standalone, Yarn, Mesos	Standalone, Yarn
Latency	Few seconds	Sub-seconds
Delivery semantics	Exactly once	At most once, at least once
Message passing layer	Netty + Akka	Netty
Batch framework integration	Spark	N/A
Fault tolerance	Recovery of lost work. Restart of workers via RM.	Restart of workers and supervisors like nothing happened.
Performance	400000 records / second / node	10000 records / second / node

PUTTING ALL TOGETHER

So, at last...

Lambda architecture:
- originally proposed by Nathan Marz
- "How to beat The CAP theorem" article
- www.lambda-architecture.net



HUMAN FAULT-TOLERANCE & OTHER BENEFITS

Human fault-tolerance
- As we have master dataset with raw data
  - all views can be recalculated
  - new views can be crated any time

Other benefits
- Algorithmic flexibility
  - Schema migrations are easy
  - Easy ad-hoc analysis
  - Self-auditing & keeping whole history (versioning of data rows) by design

It's real "Data Agility" way!

COMPACTION

- Compaction / "Garbage Collection"
- It's batch processing task
- Can be scheduled

IS IT ENOUGH?

- What about valuable events publishing at real-time?
- What about CEP (Complex Event Processing)?
  - fraud detection
  - compliance violations
  - security breaches
  - network outage
  - machine failures
  - application failures
  - operational issues
- What about real-time analytics:
  - online machine learning and predictions
- What about real-time optimization:
  - pricing, customer service, supply chain, offers, bandwidth allocation

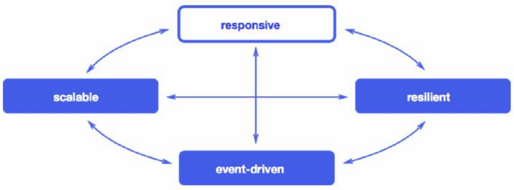
BE REACTIVE! (IN RESPONSE TO DEMAND)

- Customers demand more and more
- Business wants systems that are
  - responsive
  - resilient
  - elastic
  - message driven
  - able to process huge volumes of data

So, be reactive! ☺

REACTIVE MANIFESTO

www.reactivemanifesto.org



REACTIVE STREAMS

www.reactive-streams.org

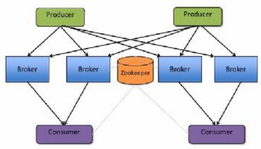
RxJava
RxScala
RxClojure
Rx.NET
RxJS
and others

```
var mouseDown = from evt in FromEvent(image, "MouseDown")
        select GetPosition(image);
var mouseUp = FromEvent(image, "MouseUp");
var mouseMove = from evt in FromEvent(image, "MouseMove")
        select GetPosition(this);

var q = from imageOffset in mouseDown
        from pos in mouseMove.Until(mouseUp)
        select new {
            X = pos.X - imageOffset.X,
            Y = pos.Y - imageOffset.Y };

q.Subscribe(value =>  {
        Canvas.SetLeft(image, value.X);
        Canvas.SetTop(image, value.Y); });
```

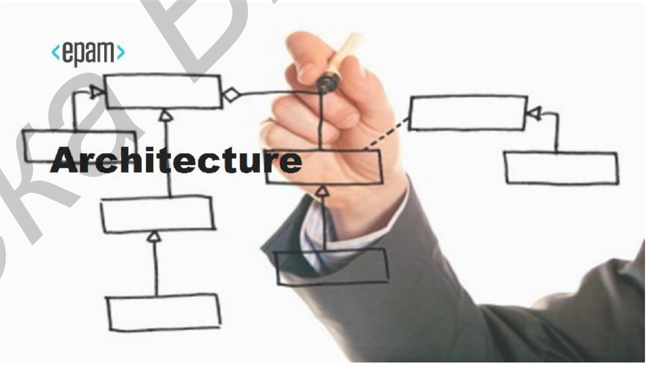
KAFKA

Kafka is really reactive Message Queue
fast, scalable, durable

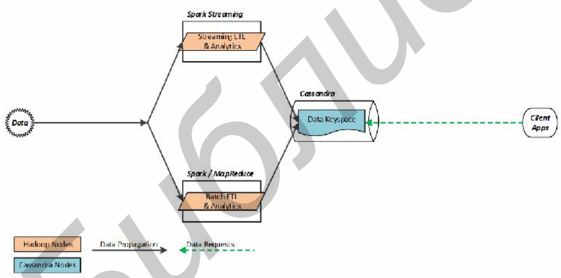
Kafka Architecture



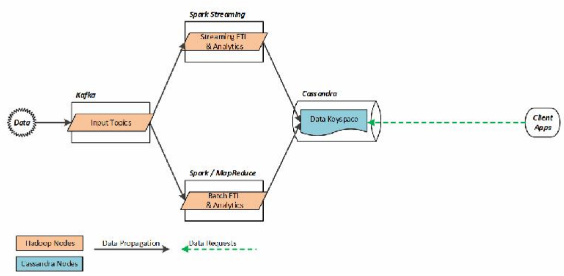
	Kafka
Hadoop distribution	Hortonworks
Implemented in	Scala
API language	Java, Scala
Stack	N/A
Processing model	Record-at-a-time
Coordinator	Zookeeper
Resource manager	Standalone, Yarn
Latency	Low milliseconds
Delivery semantics	At most once, at least once, exactly once
Message passing layer	Fairly straight-forward NIO server
Batch framework integration	MapReduce, Spark and others have connectors for Kafka
Fault tolerance	Sequential, write-ahead, partitioned message log. Paritions are replicated across a configurable number of servers.
Performance	2 million writes per second (on 3 cheap machines)



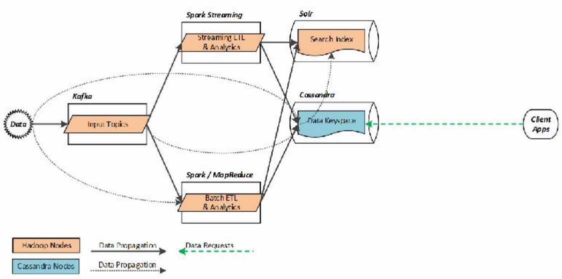
ARCHITECTURE - LAMBDA



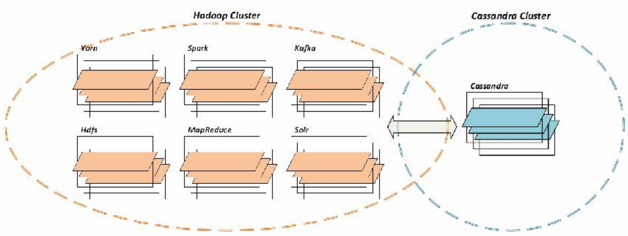
ARCHITECTURE - REALITY



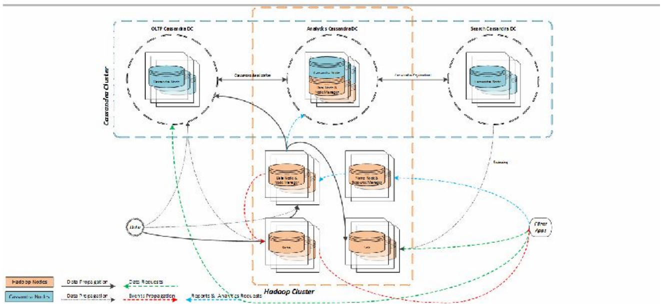
ARCHITECTURE - SERVICES



ARCHITECTURE – ROLES



ARCHITECTURE - CLUSTER



SOLR ON YARN

www.lucidworks.com/blog/solr-yarn
www.github.com/LucidWorks/yarn-proto
issues.apache.org/jira/browse/SOLR-6743

RECOMMENDED RESOURCES

O'Reilly, 2015, Hadoop - The Definitive Guide, 4ed
O'Reilly, 2015, Learning Spark - Lightning Fast Data Analysis
Packt, 2015, Learning Apache Kafka, 2ed
Packt, 2015, Learning Apache Cassandra
Packt, 2015, Real-time Analysis with Storm and Cassandra
Nathan Marz's blog: http://nathanmarz.com
Databricks & DataStax & Hortonworks blogs