About the speakers

Luigi Troiano (M.Eng. 2000, Ph.D. 2004) is assistant professor and researcher at University of Sannio since 2006, where he is lecturing in Software Engineering and Intelligent Systems. He obtained Laurea (master degree) in IT Engineering at University of Naples "Federico II" in 2000, and Ph.D. in IT Engineering at University of Sannio in 2004. His research interests are mainly related to Computational Intelligence and Intelligent Systems, focusing on how to apply mathematical models and advanced algorithms to real-world problems. His competences refer to conception, data analysis, algorithm experimentation and validation, implementation of software systems and solutions, using the state of art of technologies. He earned an industrial experience working in Italy and abroad for multinational companies such as Pirelli Trelleborg (1999), Siemens ICN (2000-2001) and TotalFinaElf E&P Paris (2001-2002).

Maria Carmela Vitelli (Bc.Eng. 2010, M.Eng. 2012) is a PhD student at University of Sannio. Se obtained Laurea (master degree) in IT Engineering at University of Sannio in Benevento in 2012. Her research is focused on Algorithms and Tools for Big Data Analysis with possible applications to fault detection, isolation and recovery (FDIR). Investigation is aimed at scaling machine learning techniques to the massive ammount of data collected by telemetry sensors in order to identify patterns (generally unusual and sparse) which could provide early signals on something going to fail in the system. In particular, the approach she is investigating relies on Bayesian probabilistic reasoning and data mining techniques.

Enabling Big Data Architectures for the KAGRA Project

Luigi Troiano and Maria C. Vitelli
University of Sannio, Italy

Graduate School of Science, University of Osaka — February 17th, 2015

Outline

- **Big Data**
- Hadoop & Map-Reduce
- HDFS vs. NFS
- Evolving towards a Big Data architecture
- FPGA co-processing
- Conclusions

Big Data

- Very large, loosely structured data set that defies traditional storage
- Human and machine generated data
- Multiple sources
- Huge volumes of data that cannot be handled by traditional database or warehouse systems
- Mostly unstructured and grows at high velocity
- Big data doesn't always mean huge data, it means "*difficult*" data

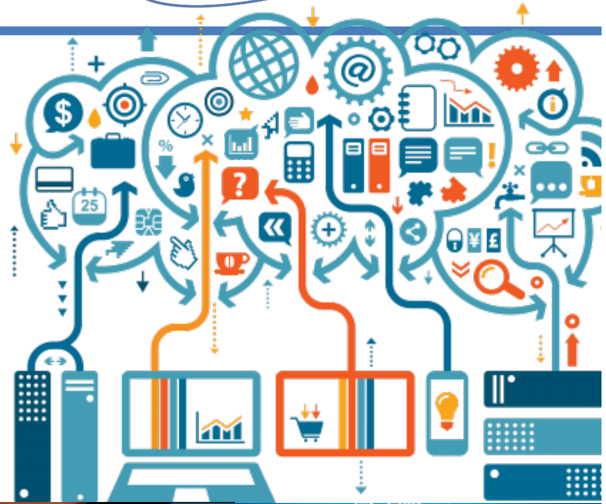
The 4 Vs

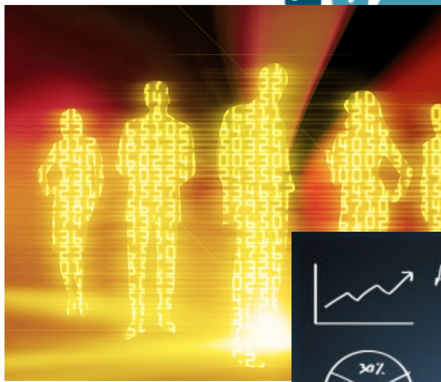
- **Volume:** Data is too big to scale out
- **Velocity:** Decision window is small
- **Variety:** Multiple formats challenge integration
- **Veracity:** Same data, different interpretations

Big Data Application Domains

- Healthcare
- The public sector
- Retail
- Manufacturing
- Personal-location data
- Finance

…and science





Università degli Studi del Sannio

What is Big Data used for?

Generally they are used in Business context for:

- Reports
- Diagnosis
- Decisions

…. In Science:

- **Modeling**
- **Induction**

Some examples:

- The 1000 Genomes Project is aimed to find most genetic variants that have frequencies of at least 1% in the populations studied. The genome of each human being is 100 GB long.

- Jack Gallant at UC Berkeley was able to recover what people were seeing by directly observing activity in their brains by using big data and statistical methods.

- The Large Hadron Collider (LHC) at CERN in Switzerland started to take data in 2009. The amount of data collected by CERN is about 25 PB a year.

Big Data Architecture Requirements

- Built to run on a cluster of machines
- Scale horizontally
- Handle unstructured/semi-structured data
- Provide storage and computing

Outline

- Big Data
- **Hadoop & Map-Reduce**
- HDFS vs. NFS
- Evolving towards a Big Data architecture
- FPGA co-processing
- Conclusions

hadoop

What is Apache Hadoop

Apache Hadoop is an open source platform for data storage and processing that is

- ✓ Scalable
- ✓ Fault tolerant
- ✓ Distributed

Other features:

- ◉ Flexibility to store and mine any type of data
- ◉ Excels at processing complex data
- ◉ Scales economically

Hadoop

HDFS + MapReduce

- Horizontal scalability
- Commodity hardware
- Fault tolerance

- Programming framework
- Organize multiple computers in a cluster in order to perform needed calculations
- Fault tolerance

hadoop

Map Reduce

MapReduce

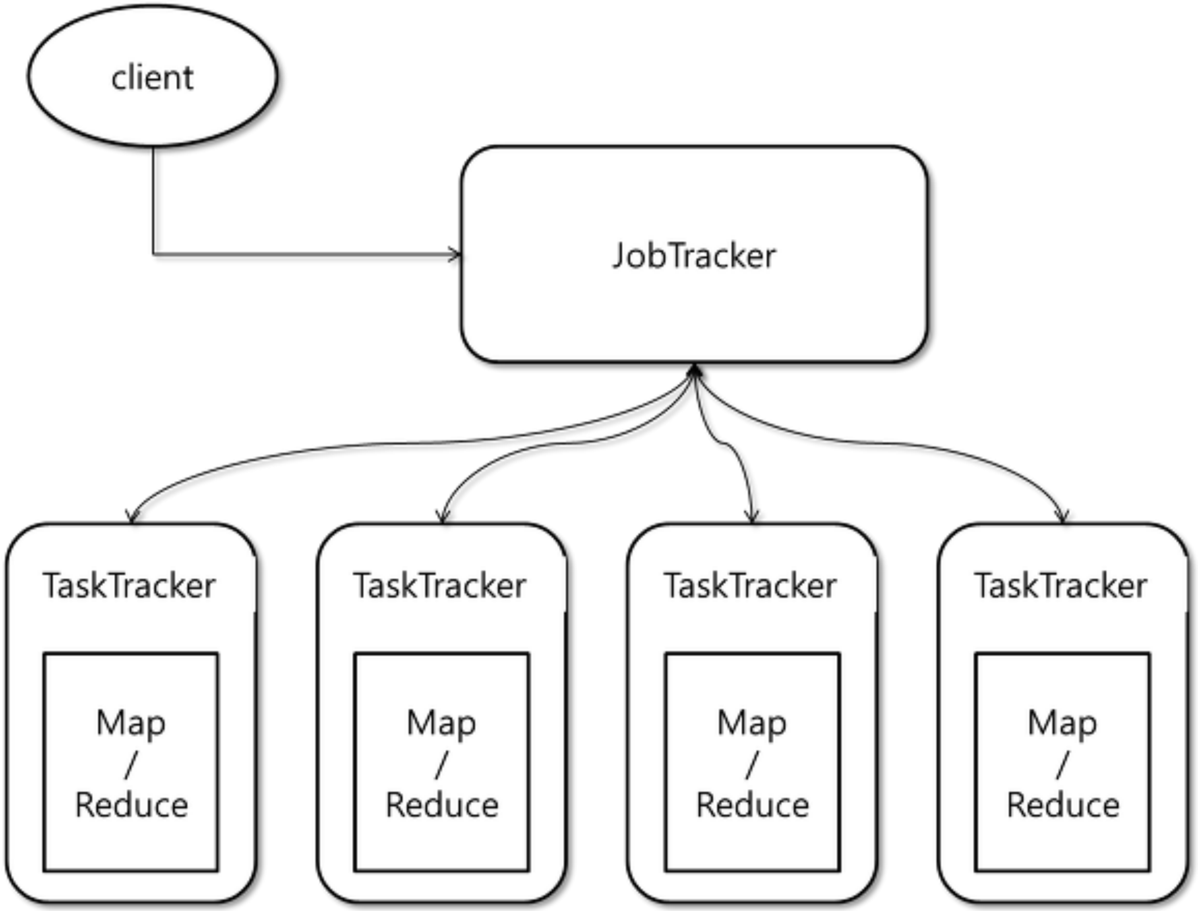
- A programming model for data processing
- Consists of two phases: ***Map*** and ***Reduce***
  - Take a large problem and divide it into sub-problems
  - Perform the same function on all sub-problems
  - Combine the output from all sub-problems
- Each phase has ***key-value pairs*** as input and output

Some terminology

- ***Job***: unit of work that the client wants to be performed. It consists of the input data, the MapReduce program and configuration information
- Hadoop runs the job dividing it into ***tasks***: map tasks and reduce tasks
- There are two types of nodes that control the job execution process: a ***jobtracker*** and a number of ***tasktracker***
- Hadoop divides the input to a MapReduce job into fixed-size pieces called ***splits***
- ***Data locality optimization***: run the map task on a node where the input data resides in HDFS
- Map task write their output to the local disk, not to HDFS

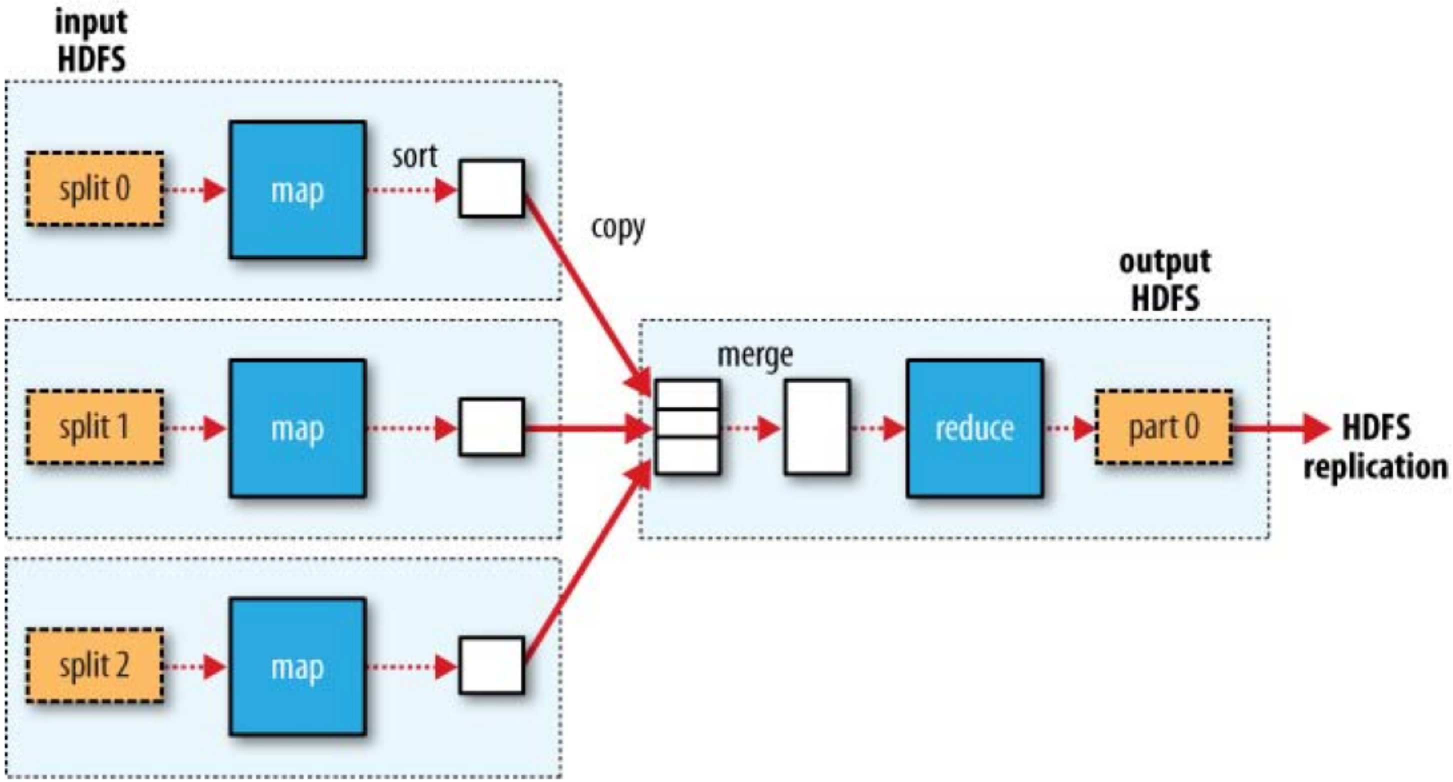
Design of MapReduce

- JobTracker
- TaskTracker



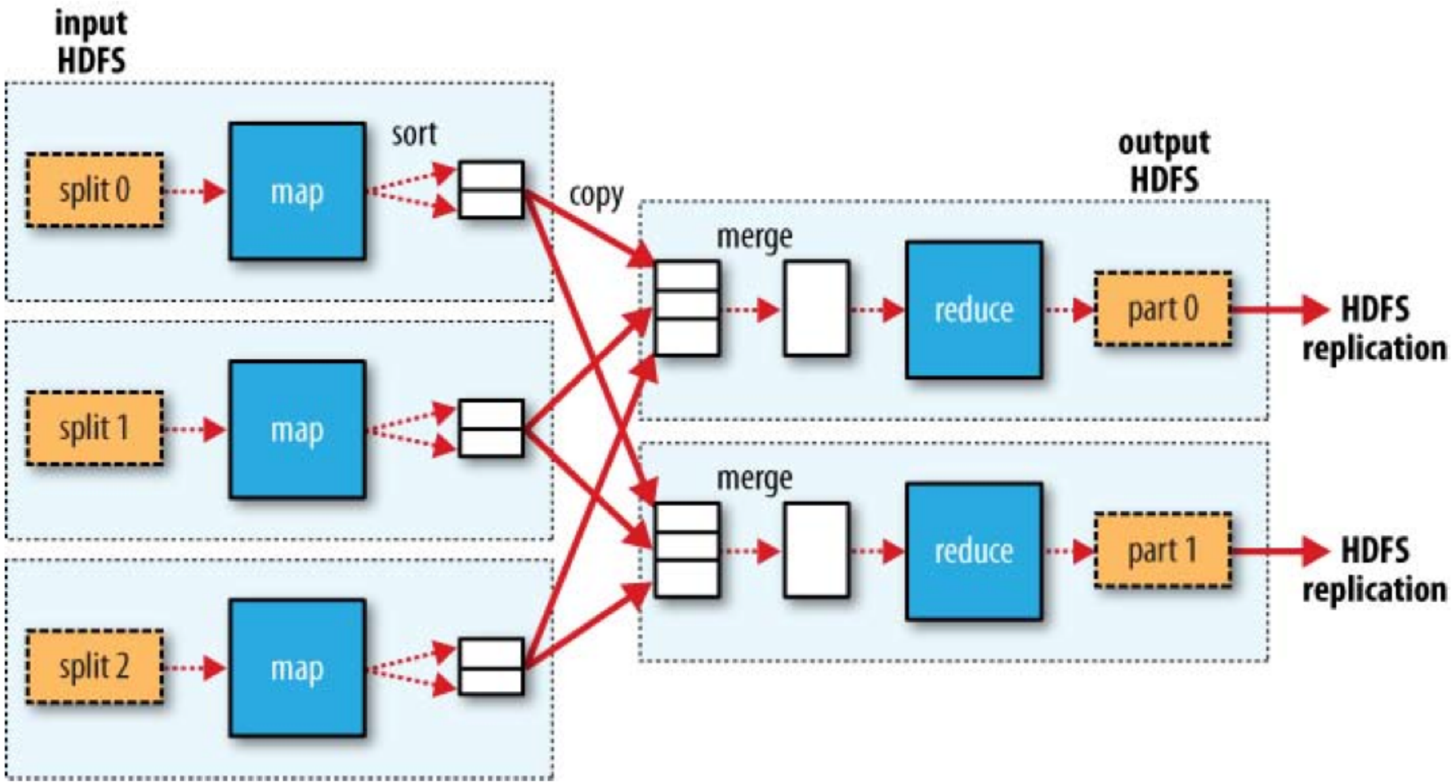
Data Flow (1/3)

- MapReduce data flow with a single reduce task



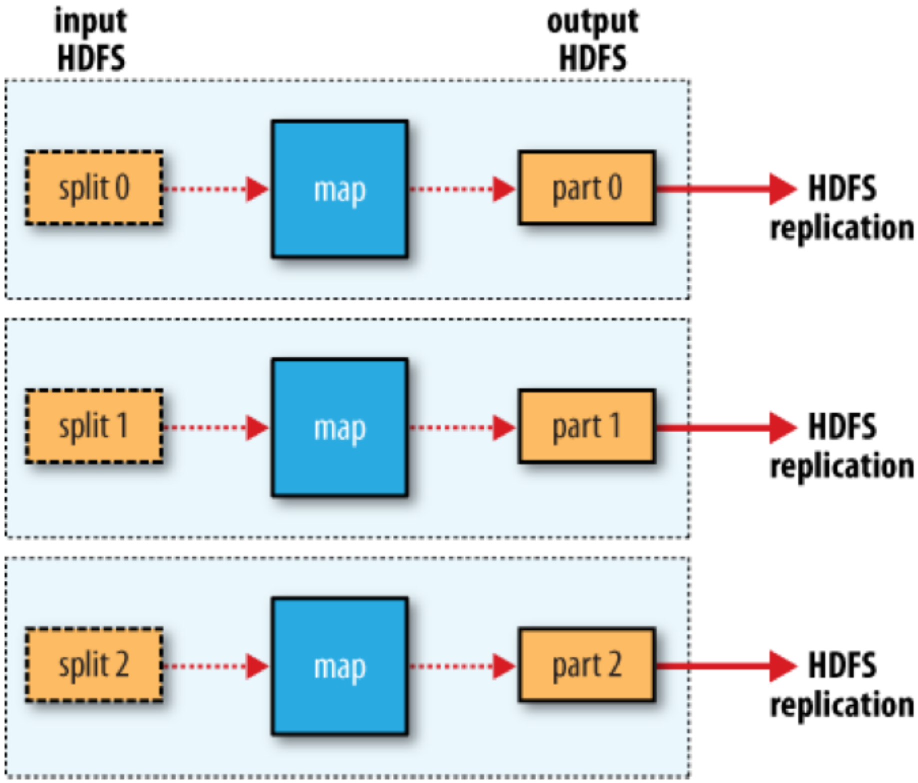
Data Flow (2/3)

- MapReduce data flow with multiple reduce tasks
- Map tasks *partition* their output
- Data flow between map and reduce tasks is called "*the shuffle*"



Data Flow (3/3)

- MapReduce data flow with no reduce tasks
- Appropriate when processing can be carried out entirely in parallel



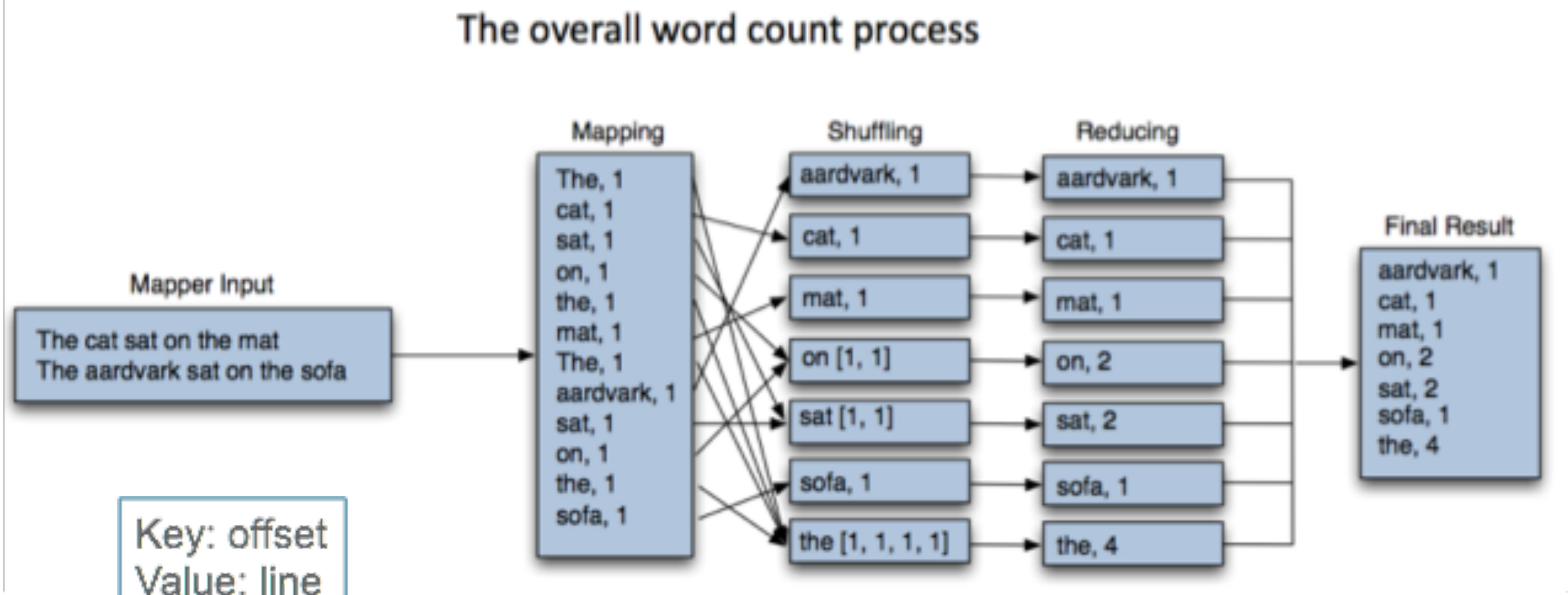
MapReduce example: WordCount (1/2)

Count the number of occurrences of each word in a large amount of input data

```
map(String input_key, String input_value)
    foreach word w in input_value:
        emit(w, 1)
```

```
reduce(String output_key,
                        Iterator<int> intermediate_vals)
    set count = 0
    foreach v in intermediate_vals:
        count += v
    emit(output_key, count)
```

MapReduce example: WordCount (2/2)



Key: offset
Value: line

Key: word
Value: count

Key: word
Value: sum of count

Outline

- Big Data
- Hadoop & Map-Reduce
- **HDFS vs. NFS**
- Evolving towards a Big Data architecture
- FPGA co-processing
- Conclusions

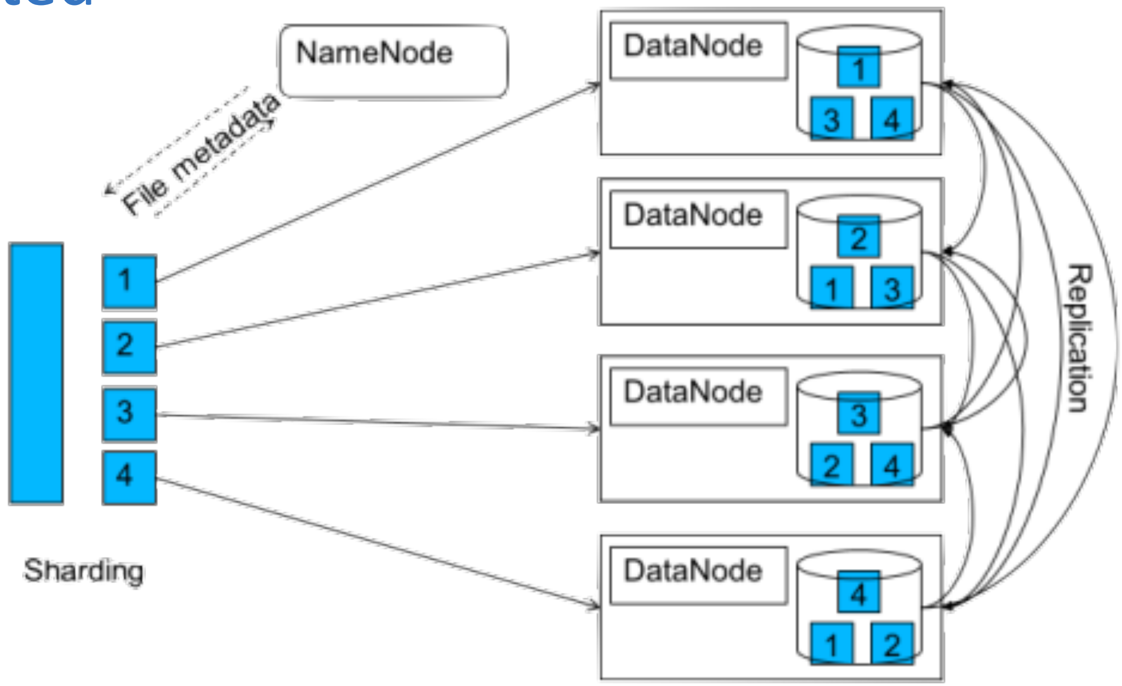
hadoop HDFS

HDFS Concepts

- Stores files by breaking it into smallest units called ***Blocks***
- Default block size: 64MB
- Large block size to help in maintaining high throughput
- To ensure both reliability and availability each block is replicated across multiple machine on the cluster

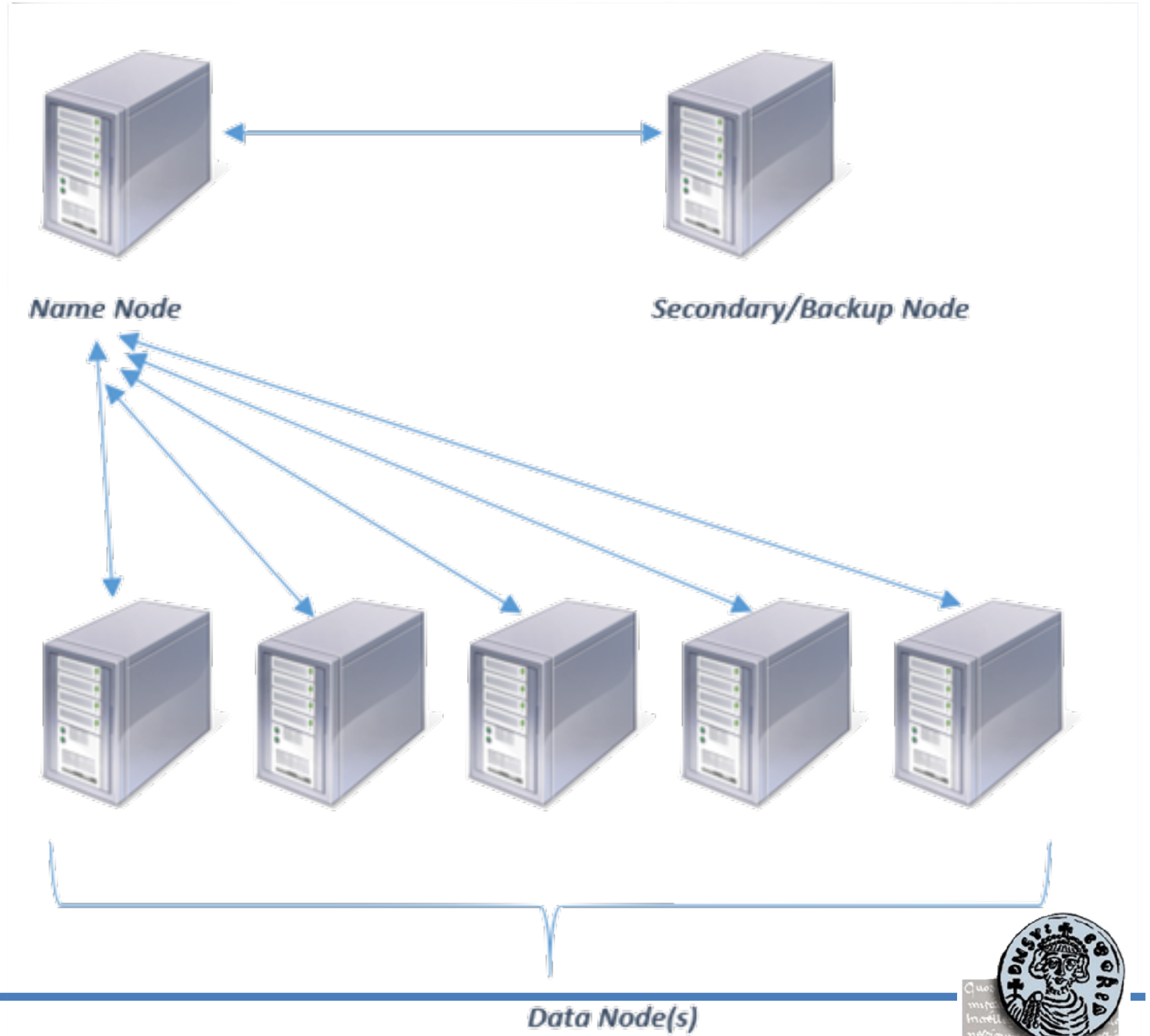
HDFS Architecture

- Master/Worker design
- HDFS is resilient (even in case of node failure)
- Data is replicated



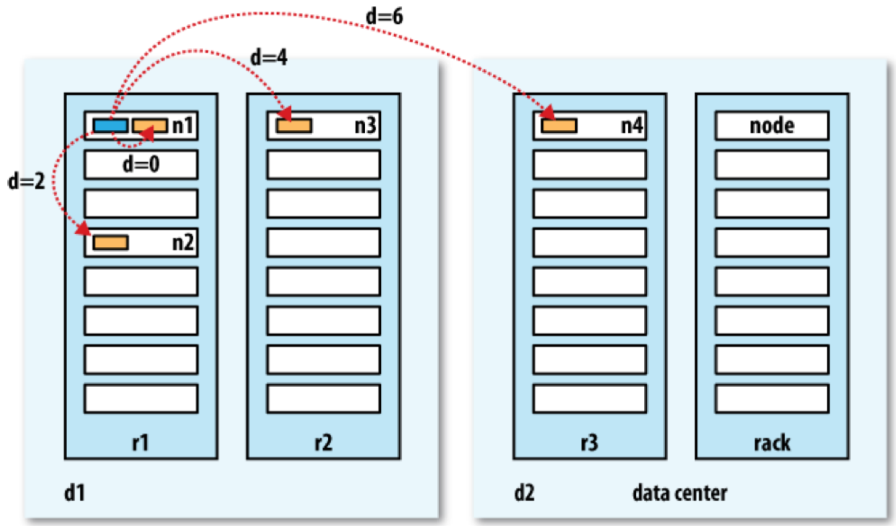
Design of HDFS - Daemons

- NameNode
- DataNode
- Secondary NameNode



Network Topology

- Network is presented as a tree
- Distance between 2 nodes is the sum of their distances to their closest common ancestor



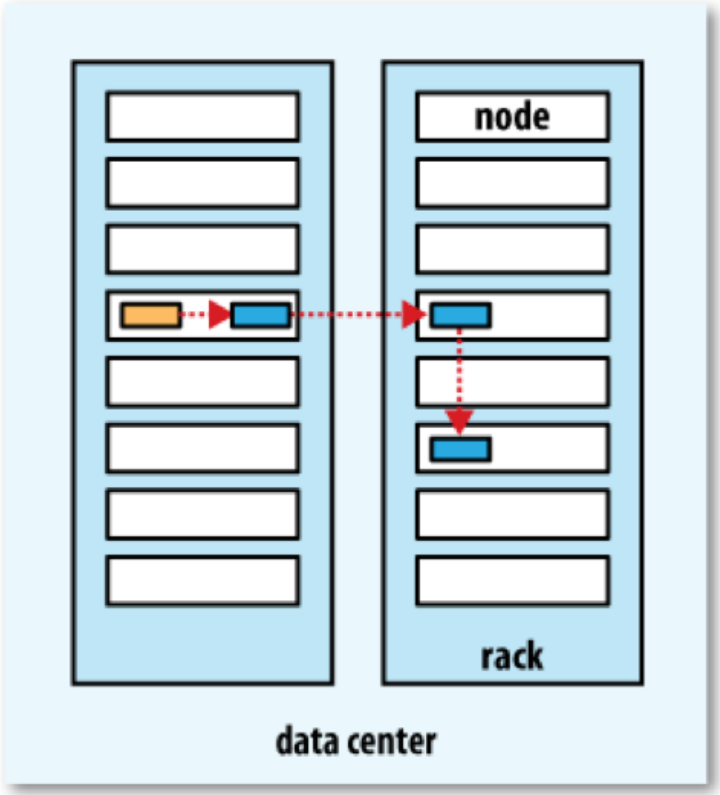
- Bandwidth available for each of the following scenarios becomes progressively less:
  - processes on the same node
  - different nodes on the same rack
  - nodes on different racks in the same data center

Replica Placement

- Tradeoff between reliability and write/read bandwidth

- Default strategy is to place
  - the *first replica* on the same nodes as the client
  - the *second replica* on a different rack from the first, chosen at random
  - the *third replica* on the same rack as the second, but on a different node chosen at random
  - *further replicas* on random nodes on the cluster

- Block replication is across distinct datanodes



HDFS

- "File System" or "Storage Layer" of Hadoop
- Designed for storing very large files (on a petabytes scale)
- Breaks incoming files into blocks and stores them redundantly across the cluster

Problem	***Hadoop solution***
Data is too big to store in one computer	*Data is stored on multiple computers*
Very high end machines are expensive	*Run on commodity hw*
Commodity hw will fail	*Sw is intelligent enough to deal with hw failure*
Hw failure may lead to data loss	*Replicate (duplicate) data*

About NFS

- Design straightforward but also very constrained
- Provides remote access to a single logical volume stored on a single machine
- An NFS server makes a portion of its local system visible to external clients
- Important advantage: transparency
- As a distributed file system it is limited in its power: files in an NFS volume all reside on a single machine.
  - *No reliability*
  - *Possible server overload*
  - *Clients must copy data to their local machines before they can operate on it*

About HDFS

- Designed to store a very large amount of information
  - Spreading data across a large number of machines
  - Support much larger file sizes than NFS
- Store data reliably
  - Data should be available if individual machines in the cluster malfunction
- Provide fast, scalable access to information
  - Serve a larger number of clients by adding more machines to the cluster

HDFS vs NFS (1/2)

HDFS

- ***Distributed File System***, in which different machines are grouped for storing the data in a distributed manner
- The same data is stored in a distributed way on set of *commodity hardware*
- Allows *parallel processing* of data through MapReduce
- *Replicates the data* and thus provide *fault tolerance*

NFS

- ***Network File System***, that provides a shared directory to a number of machines that can access the directory as good as a directory on local file system
- All data belonging to an entity (file or set of files) are stored on a single machine and thus require a *dedicated hardware*
- Since data is stored on a single location it has been *read sequentially*
- Since data in NFS is *stored on a single machine*, it is difficult to access/restore if the machine goes out of network

HDFS vs NFS (2/2)

NFS	Hadoop
If the volume is full, no more files can be stored	*Information is stored across many servers and larger file sizes than NFS are supported*
If a volume or a server fails, there is no built-in redundancy	*Data are stored reliably*
Block sizes: 4-8KB	*Bock sizes: 64MB*
Most NFS administration is command line or included with overall system management tools	*HDFS provides a web server to perform status monitoring and file browsing operations*

Hadoop Distributions

- Distributions provide easy to install mediums like RPMs
- Distros package multiple components that work well together
- Tested
- Support

The Hadoop Ecosystem
(partial)

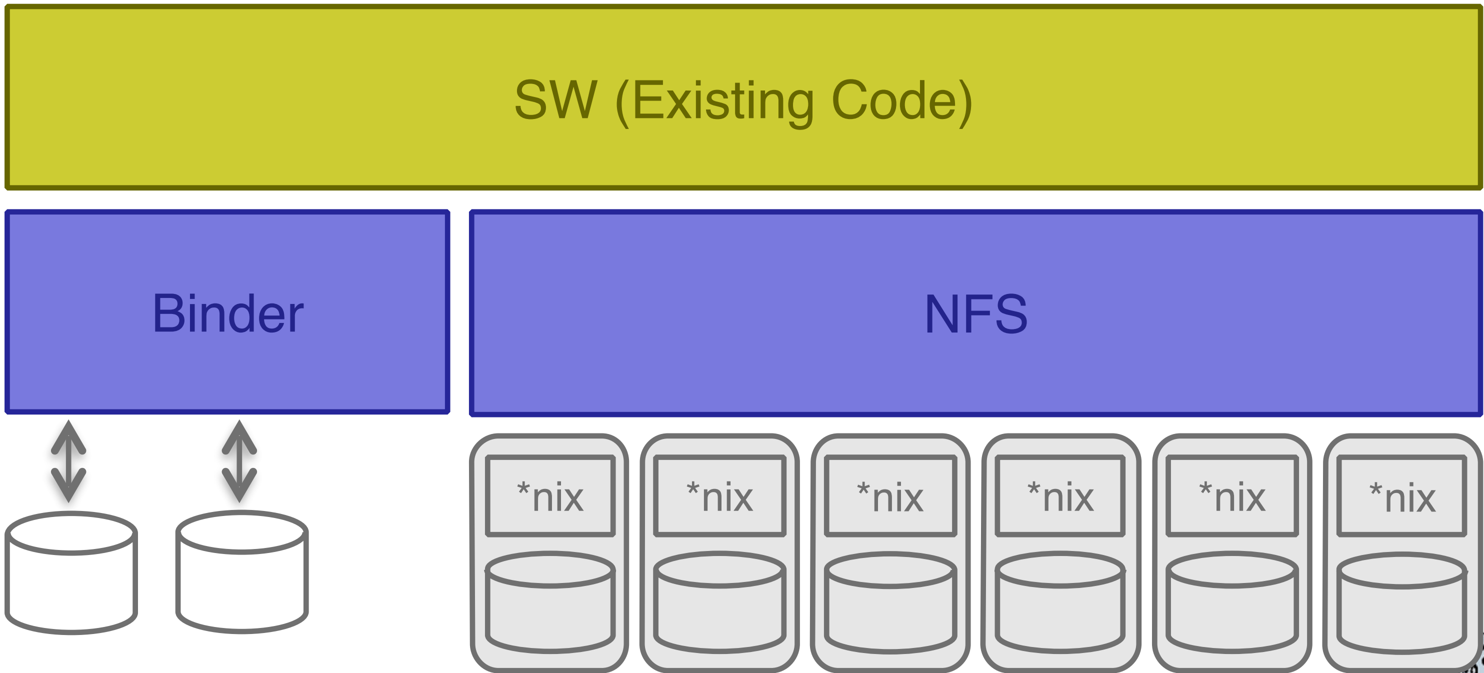
Outline

- Big Data
- Hadoop & Map-Reduce
- HDFS vs. NFS
- **Evolving towards a Big Data architecture**
- FPGA co-processing
- Conclusions

The Starting Point

A software architecture, relying on:

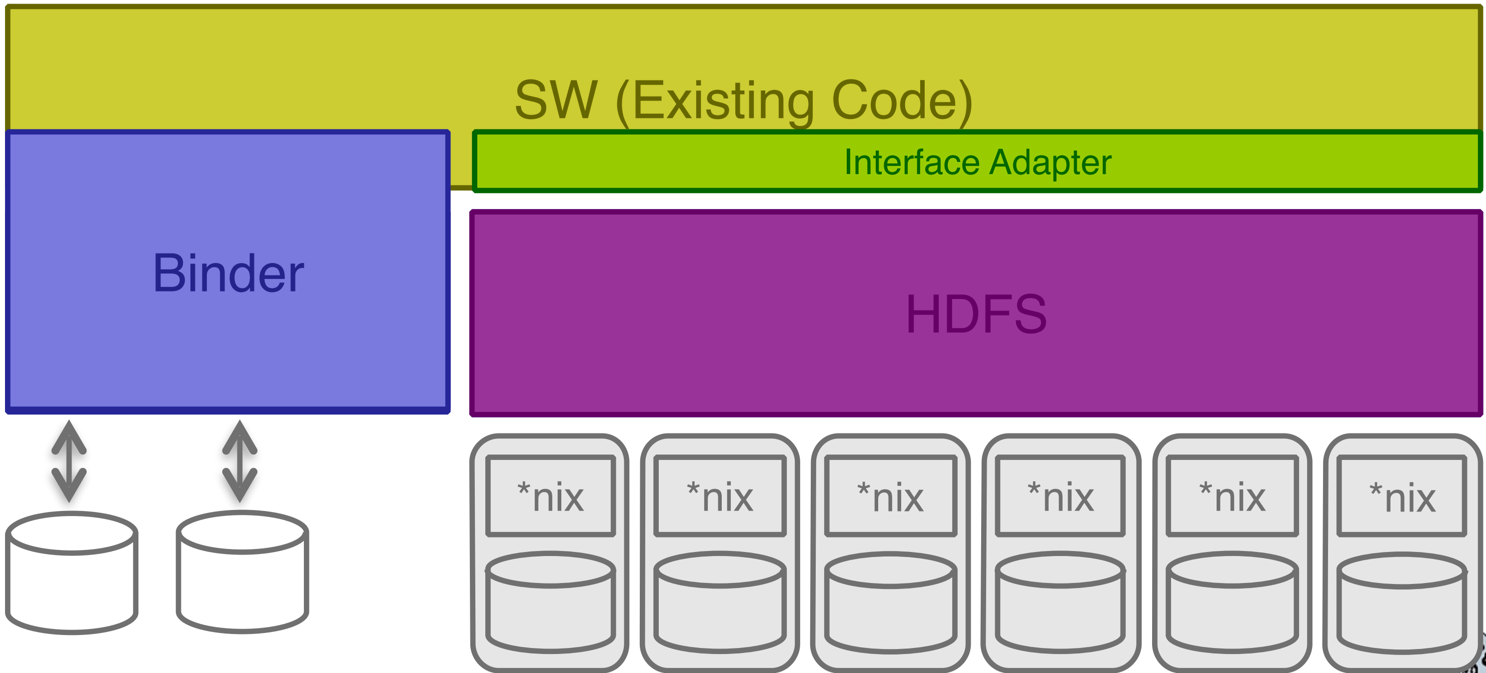
- A set of software procedures
- A *nix cluster glued by NFS
- A binder to external sources



Step 1

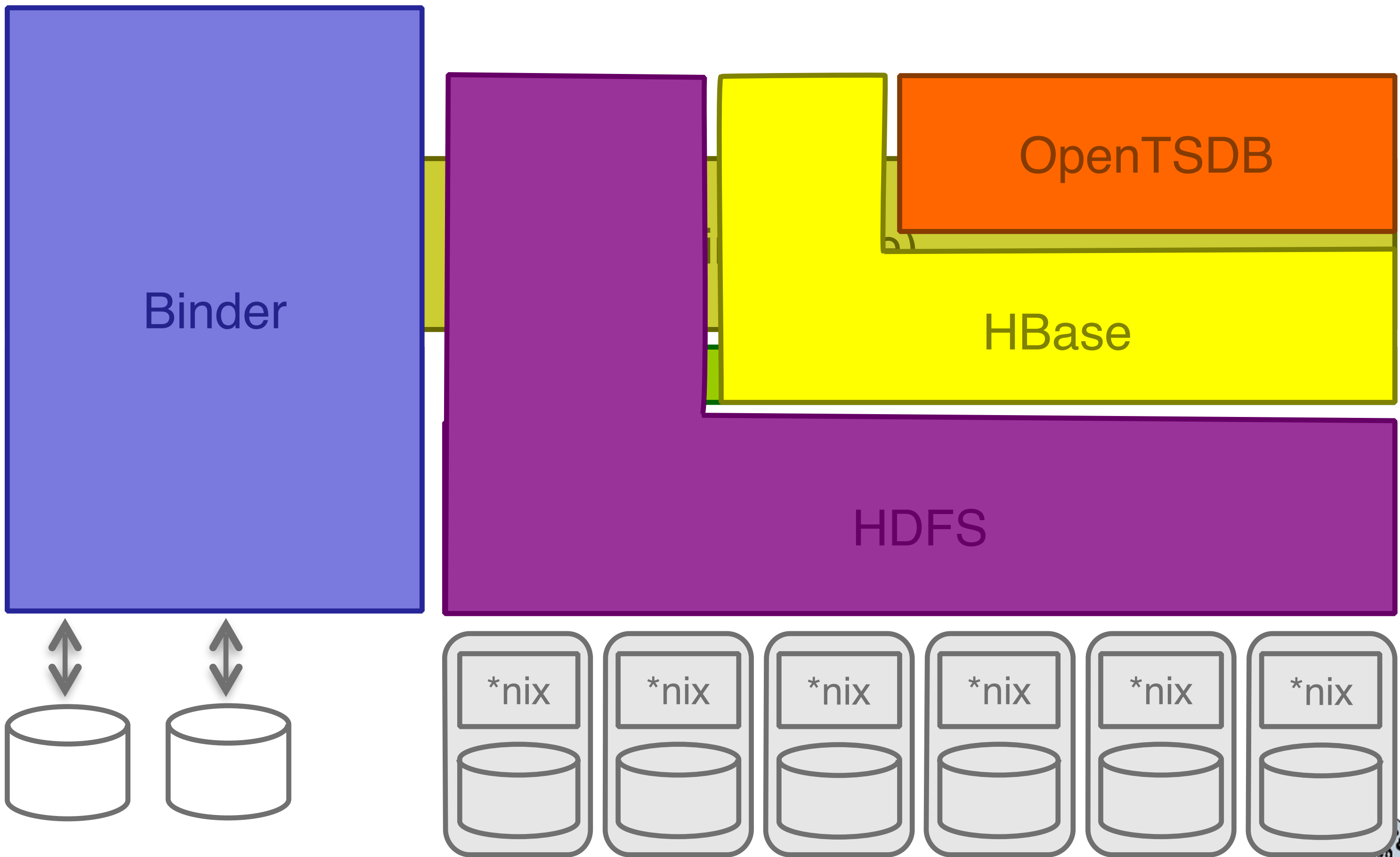
Move from NFS to HDFS:

- To achieve higher reliability
- To increase scalability



Step 2

- Introduce HBASE and OpenTSDB



HBASE

- Open source, distributed, versioned, column-oriented store
- Distributed Key/Value store
- Simple API (PUT, GET, DELETE, SCAN)
- Not a relational database
- Data changes through time
- Table rows are sorted by row key

Table

Row	Timestamp	Animal		Repair
		Type	Size	Cost
Enclosure1	12	Zebra	Medium	1000€
	11	Lion	Big	
Enclosure2	13	Monkey	Small	1500€

Region

Key

Column

Family

Cell

(Table, Row_Key, Family, Column, Timestamp) = Cell (Value)

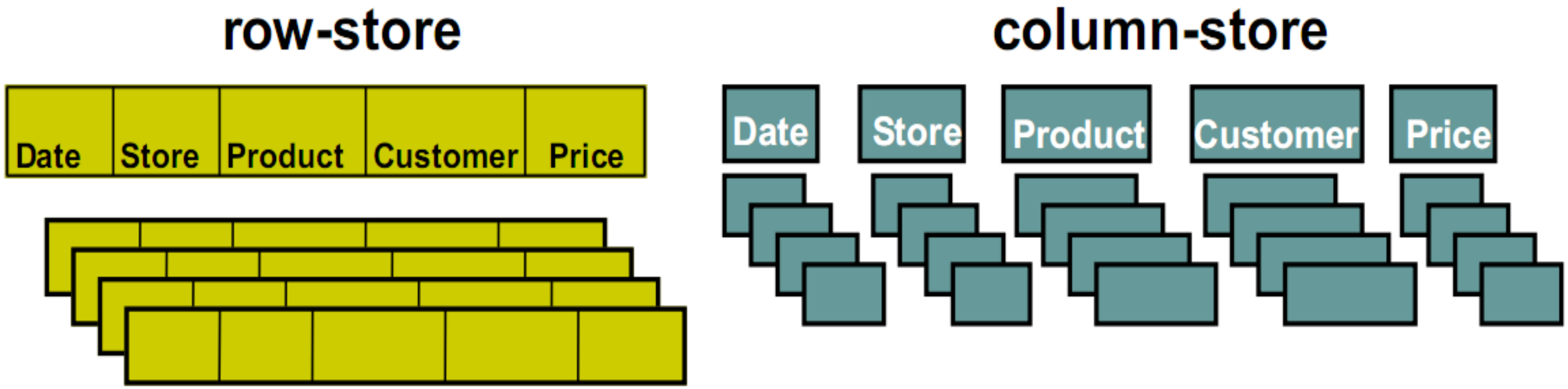
HBASE

- Row columns are grouped into ***column families***
- All column families have a common prefix
- ***Column-family-oriented store***: physically, all column family members are stored together on the filesystem (HFile)
- The basic unit of scalability and load balancing in HBase is called a ***region***
- Regions are contiguous ranges of rows stored together
- Each region is served by one ***region server***, and each of these region servers can serve many regions at any time

Column-oriented DB

Column-oriented (a.k.a. vertical) databases store data with a focus on columns, instead of rows, allowing for huge data compression and very fast query times.

The downside to these databases is that they will generally only allow batch updates, having a much slower update time than traditional models.



Columns vs. Rows

- Column-oriented databases are suitable for read-mostly, read-intensive, large data repositories
  - OLAP, On-Line Analytical Processing
  - Big Data Analytics
- Row-oriented (conventional) databases are more suitable for accessing/update single transactions
  - OLTP, On-Line Transaction Processing
  - CRUD, Create/Read/Delete/Update activities

Row Store	Column Store
(+) Easy to add/modify a record	(+) Only need to read in relevant data
(-) Might read in unnecessary data	(-) Tuple writes require multiple accesses

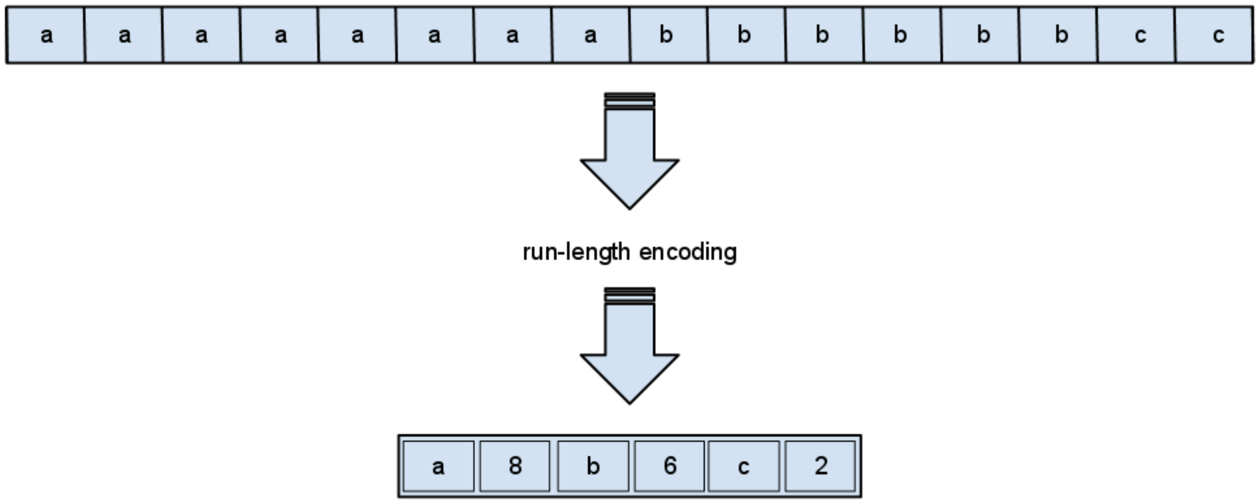
Optimizing C-DBs

- Column-oriented databases make large use of the following optimizations:
  - Compression
  - Late Materialization
  - Block Iteration
  - Invisible Join

Compression

- Low information entropy (high data value locality) leads to High compression ratio
- If data is sorted on one column that column will be super-compressible in row store

- eg. Run Length Encoding



Late Materialization

- As result of queries we expect records
- So at some point of time multiple column must be combined
- One simple approach is to join the columns relevant for a particular query
- But further performance can be improve using late-materialization

Late Materialization

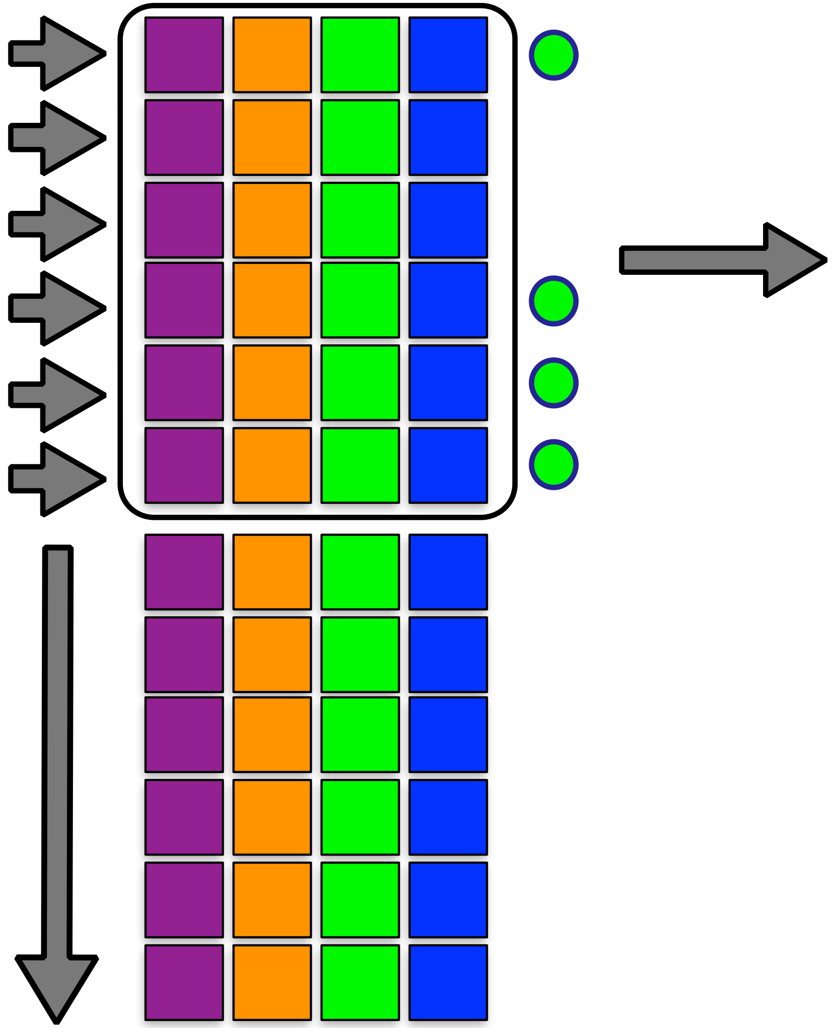
- Delay Tuple Construction
- Might avoid constructing it altogether
- Intermediate position lists might need to be constructed
- Eg: SELECT R.a FROM R WHERE R.c = 5 AND R.b = 10
  - Output of each predicate is a bit string
  - Perform Bitwise AND
  - Use final position list to extract R.a

Late Materialization

- Advantages
  - Unnecessary construction of tuple is avoided
  - Direct operation on compressed data
  - Cache performance is improved

Block Iteration

- Operators operate on blocks of tuples at once
- Iterate over blocks rather than tuples
- Like batch processing
- If column is fixed width, it can be operated as an array
- Minimizes per-tuple overhead
- Exploits potential for parallelism



Invisible Join

- Invisible join is a late materialized join but minimize the values that need to be extracted out of order
- Invisible join
  - Rewrite joins into predicates on the foreign key columns in the fact table
  - These predicates evaluated either by hash-lookup
  - Or by between-predicate rewriting

Invisible Join

```
SELECT c.nation, s.nation, d.year,
       sum(lo.revenue) as revenue
FROM customer AS c, lineorder AS lo,
     supplier AS s, dwdate AS d
WHERE lo.custkey = c.custkey
  AND lo.suppkey = s.suppkey
  AND lo.orderdate = d.datekey
  AND c.region =  ASIA
  AND s.region =  ASIA
  AND d.year >= 1992 and d.year <= 1997
GROUP BY c.nation, s.nation, d.year
ORDER BY d.year asc, revenue desc;
```

Find Total revenue from Asian customers who purchase a product supplied by an Asian supplier between 1992 and 1997 grouped by nation of the customer, supplier and year of transaction

Invisible Join

STEP 1

Apply `region = 'Asia'` on Customer table

custkey	region	nation	...
1	Asia	China	...
2	Europe	France	...
3	Asia	India	...

→ Hash table with keys 1 and 3

Apply `region = 'Asia'` on Supplier table

suppkey	region	nation	...
1	Asia	Russia	...
2	Europe	Spain	...

→ Hash table with key 1

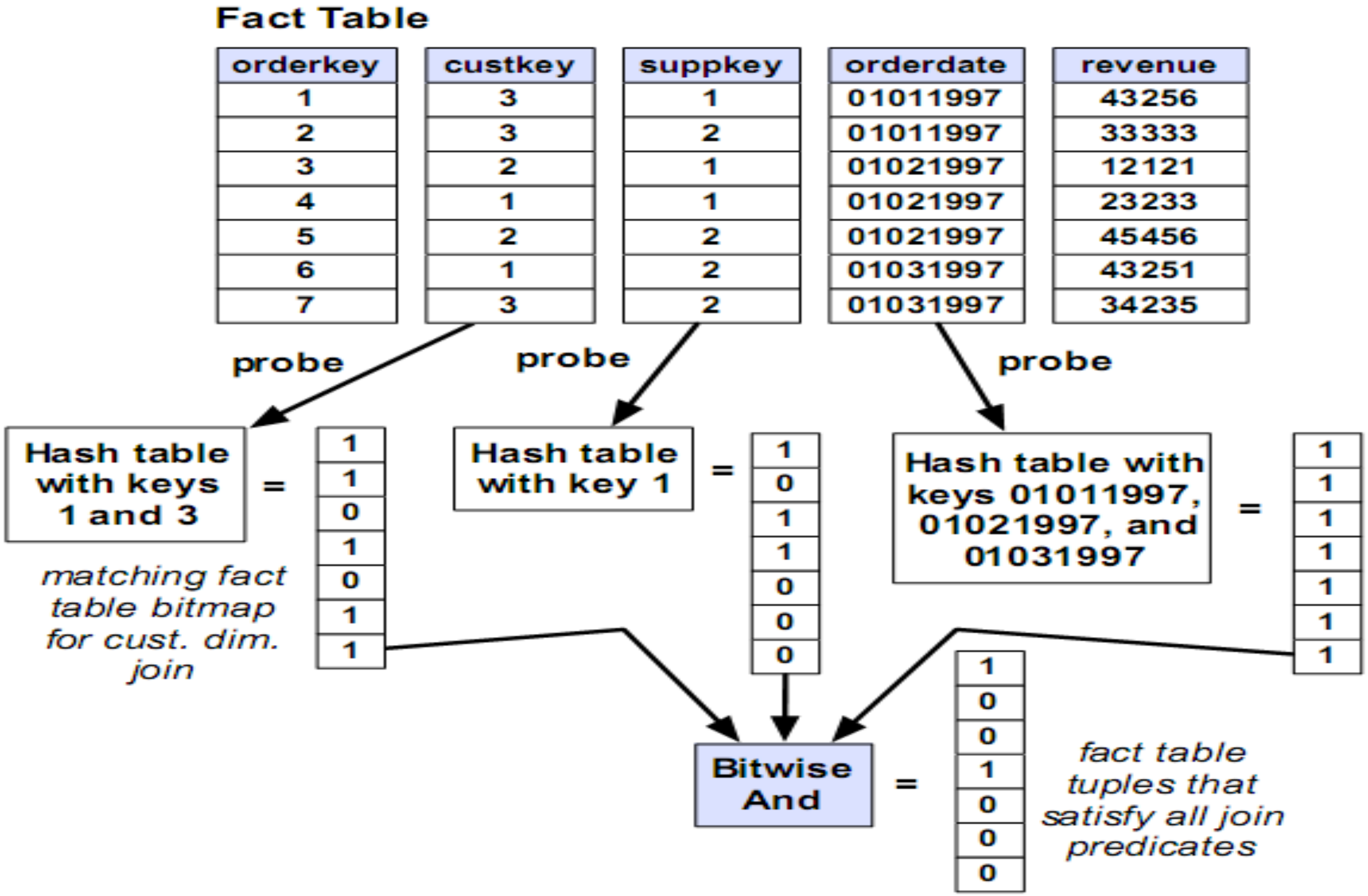
Apply `year in [1992,1997]` on Date table

dateid	year	...
01011997	1997	...
01021997	1997	...
01031997	1997	...

→ Hash table with keys 01011997, 01021997, and 01031997

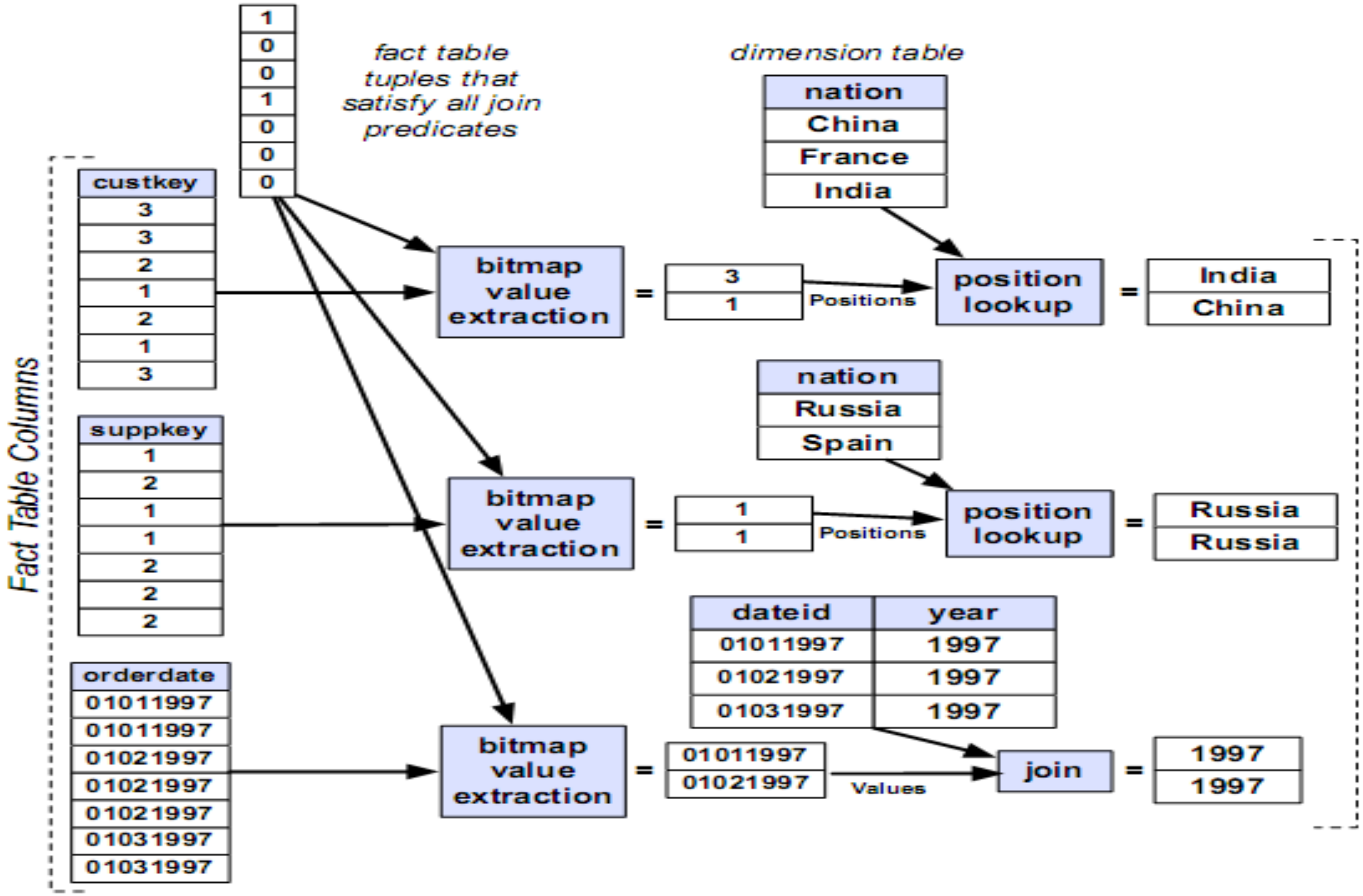
Invisible Join

STEP 2

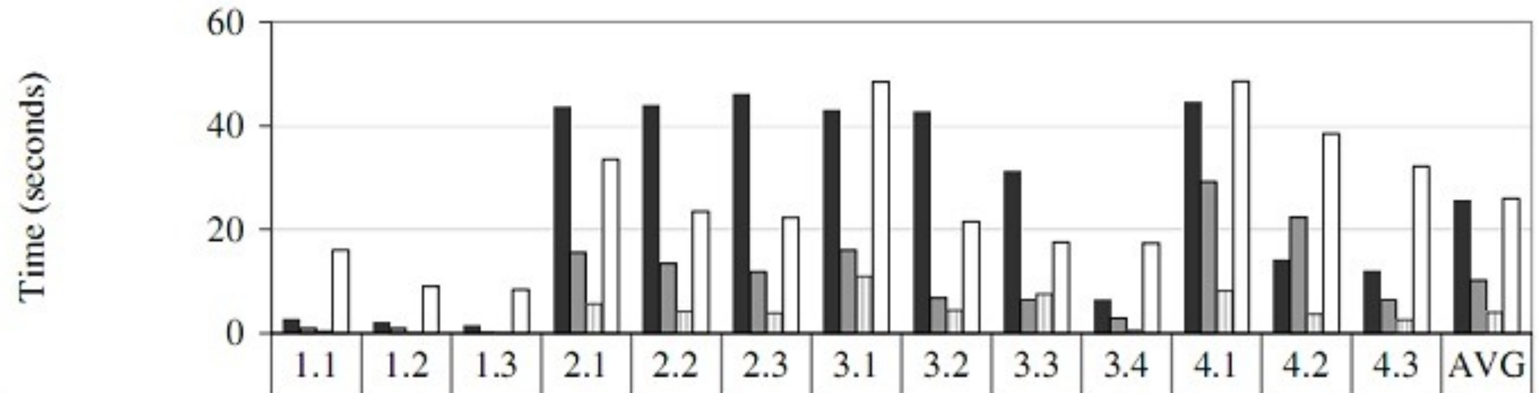


Invisible Join

STEP 3



Comparison



	1.1	1.2	1.3	2.1	2.2	2.3	3.1	3.2	3.3	3.4	4.1	4.2	4.3	AVG
RS	2.7	2.0	1.5	43.8	44.1	46.0	43.0	42.8	31.2	6.5	44.4	14.1	12.2	25.7
RS (MV)	1.0	1.0	0.2	15.5	13.5	11.8	16.1	6.9	6.4	3.0	29.2	22.4	6.4	10.2
CS	0.4	0.1	0.1	5.7	4.2	3.9	11.0	4.4	7.6	0.6	8.2	3.7	2.6	4.0
CS (Row-MV)	16.0	9.1	8.4	33.5	23.5	22.3	48.5	21.5	17.6	17.4	48.6	38.4	32.1	25.9

Baseline performance of C-Store "CS" and System X "RS", compared with materialized view cases on the same systems.

- RS: Conventional Data Base System (Not Mentioned)
- CS: Base C-Store case
- RS (MV): System X with optimal collection of MVs
- CS (Row-MV): Column store constructed from RS(MV)

OpenTSDB

OpenTSDB

*“OpenTSDB is a distributed, scalable Time Series Database written on top of HBase. OpenTSDB was written to address a common need: **store, index, and serve metrics collected from computer systems at a large scale**, and make this data easily accessible and graphable"*

- Internal data architecture supporting very high-performance data recording

Time series

- ***Time series*** data is defined as a sequence of data points measured typically at successive times spaced at uniform time intervals

- With time series data, not only it is possible to determine the sequence in which some events happened, but it is also possible to correlate different types of events or conditions that co-occur

TSDB

- Optimized for best performance for queries based on range of time
- NoSQL approaches
- Advantages in flexibility and performance
- There are different methods to store and access TS:
  - Flat files
  - RDBMS
  - NoSQL non-relational db

Flat Files

- ***Parchet*** is an effective and simple, modern format that can store the time and the number of optional values
- Problem: as the number of time series in a single file increases the fraction of usable data for any particular query decreases

```
message simpleSeries {
  repeated group sample {
    required float t;
    optional float tempIn;
    optional float pressureIn;
    optional float tempOut;
    optional float pressureOut;
  }
}
```

```
message fancySeries {
  repeated group block {
    repeated group tags {
      optional string name;
      optional string value;
    }
    repeated float time;
    repeated float value;
  }
}
```

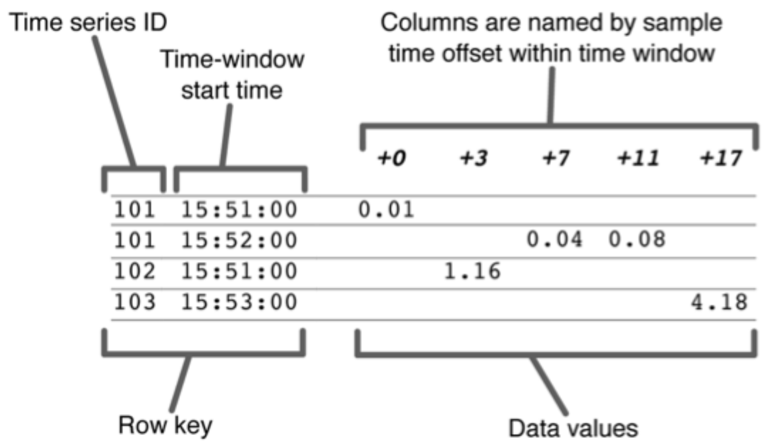
RDBMS

- Time, series ID and a value are stored
- Details of the series are stored in a dimension table
- Problem: use of one row per measurement

Time	Time series ID	Value
15:51:00	101	0.01
15:51:03	102	1.16
15:52:07	101	0.04
15:52:11	101	0.08
15:53:17	103	4.18

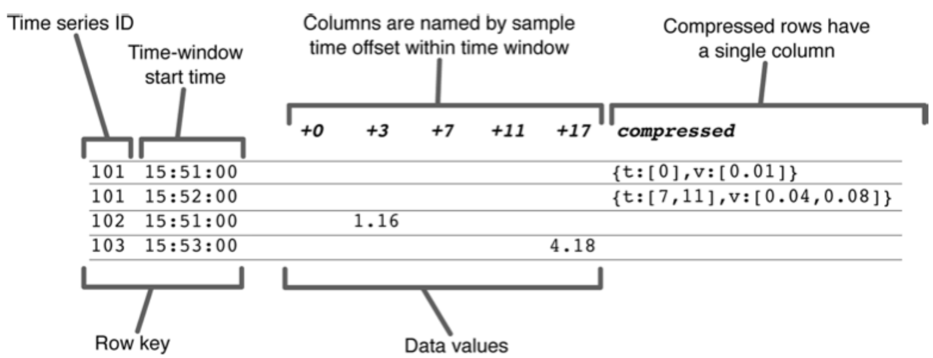
NoSQL DB with Wide Tables

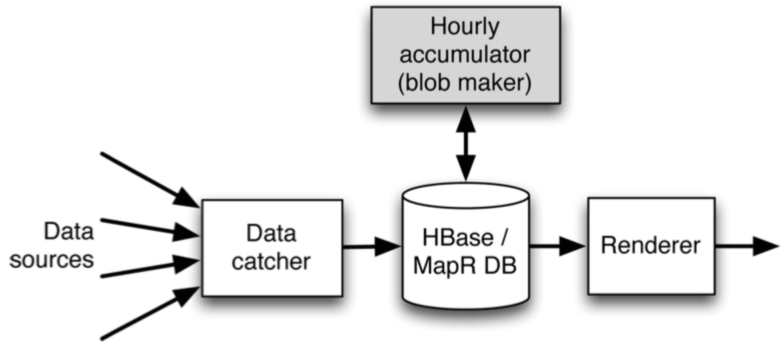
- Store many values in each row
- Allowing data point to be retrieved at a higher speed
- Rows containing data from a single time series to wind up near one another on disk



NoSQL DB with Hybrid Design

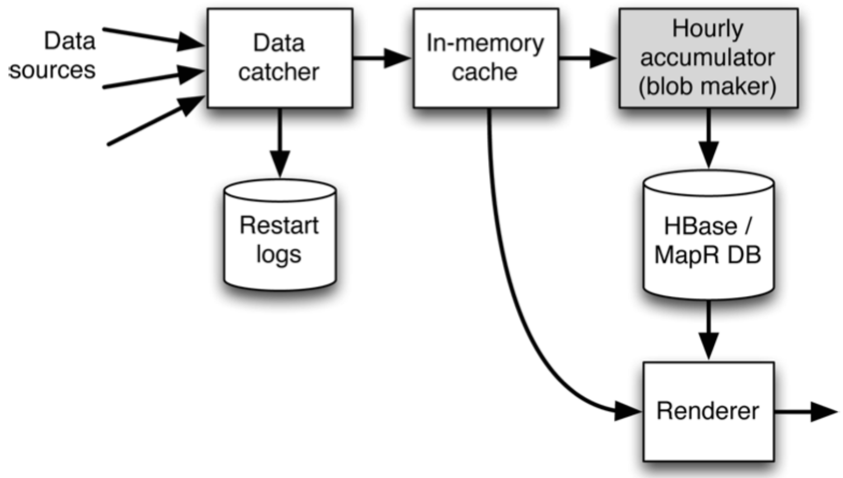
- Collapse all of data foe a row into a single data structure known as blob
- Blob can be highly compressed so that less data needs to be read from disk





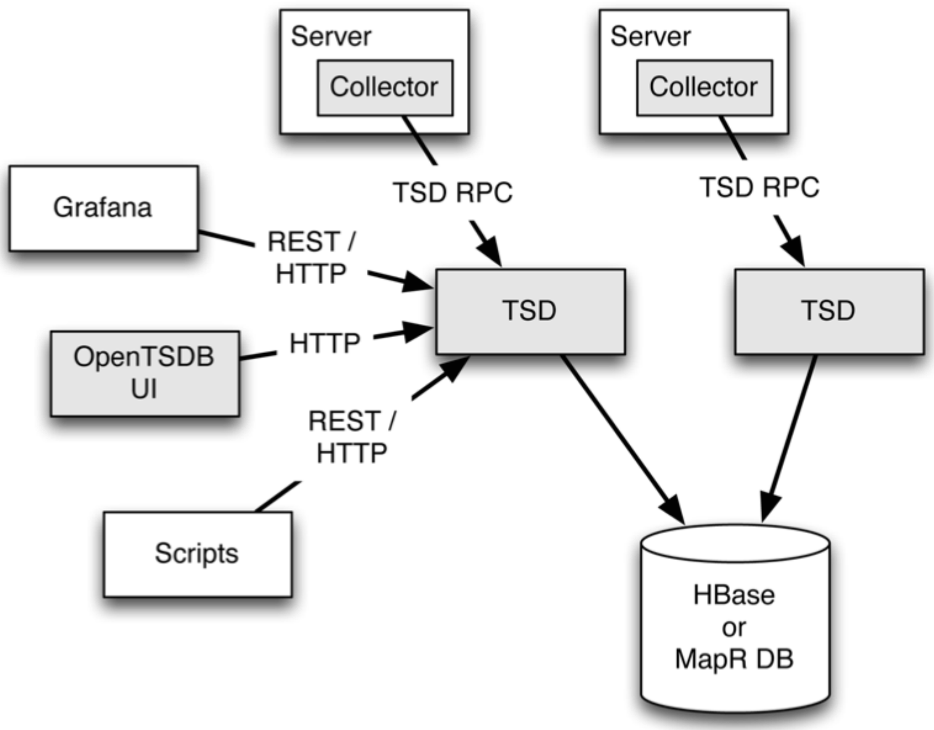
Direct Blob Insertion Design

- The blob maker uses incoming data from a memory cache
- The full data stream is only written to the memory cache
- Data is not written to the storage tier until it is compressed into blobs



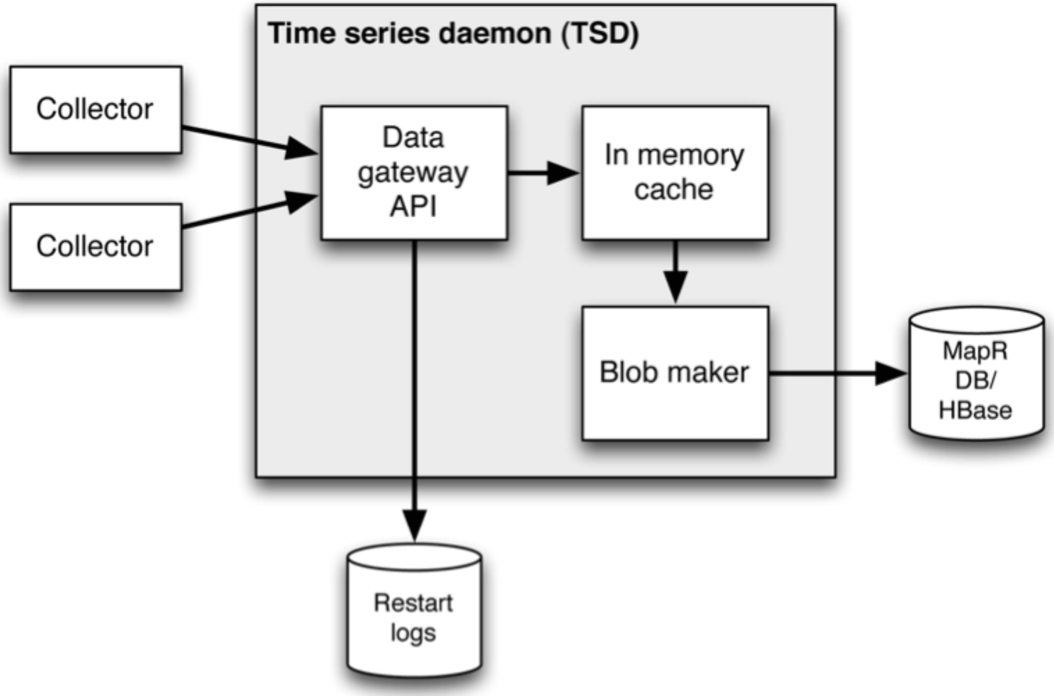
Architecture of OpenTSDB

- In OpenTSDB, a time series data point consists of:
  - A metric name.
  - A UNIX timestamp (seconds or millisecinds since Epoch).
  - A value (64 bit integer or single-precision floating point value).
  - A set of tags (key-value pairs) that describe the time series the point belongs to.
- Data collectors
- Time-series daemons (TSD)
- User interface functions
- Scripts/Tools



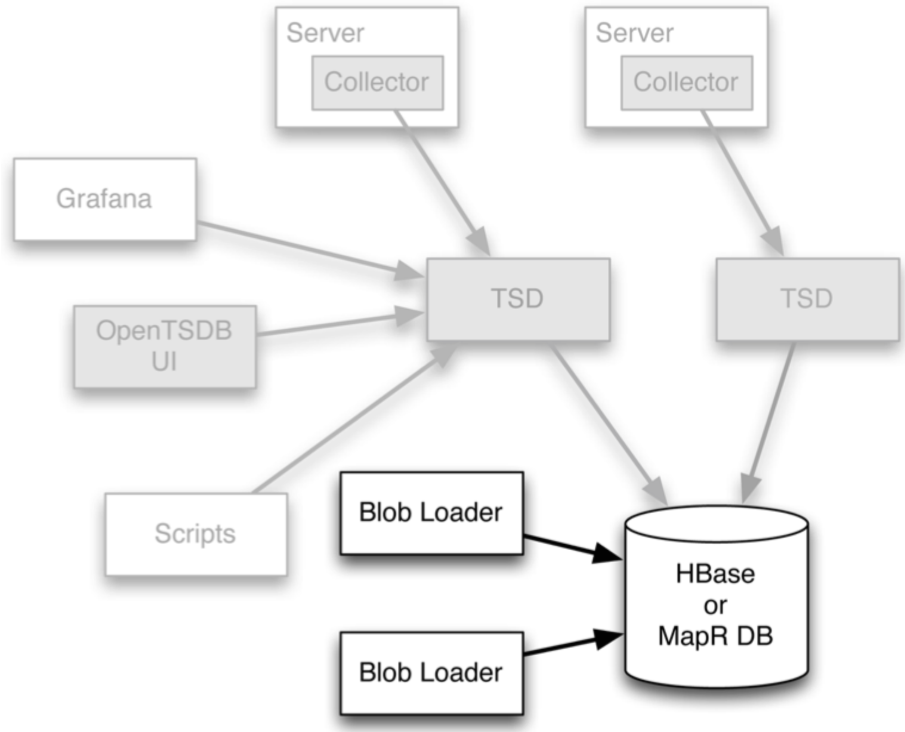
Direct Blob Loading for High Performance

- Data is ingested initially to the storage tier in the blob-oriented format that stores many data points per row



Rapid Loading of Historical Data

- Separate data flow that loads data in blob-style format directly to the storage tier independently of the TSD

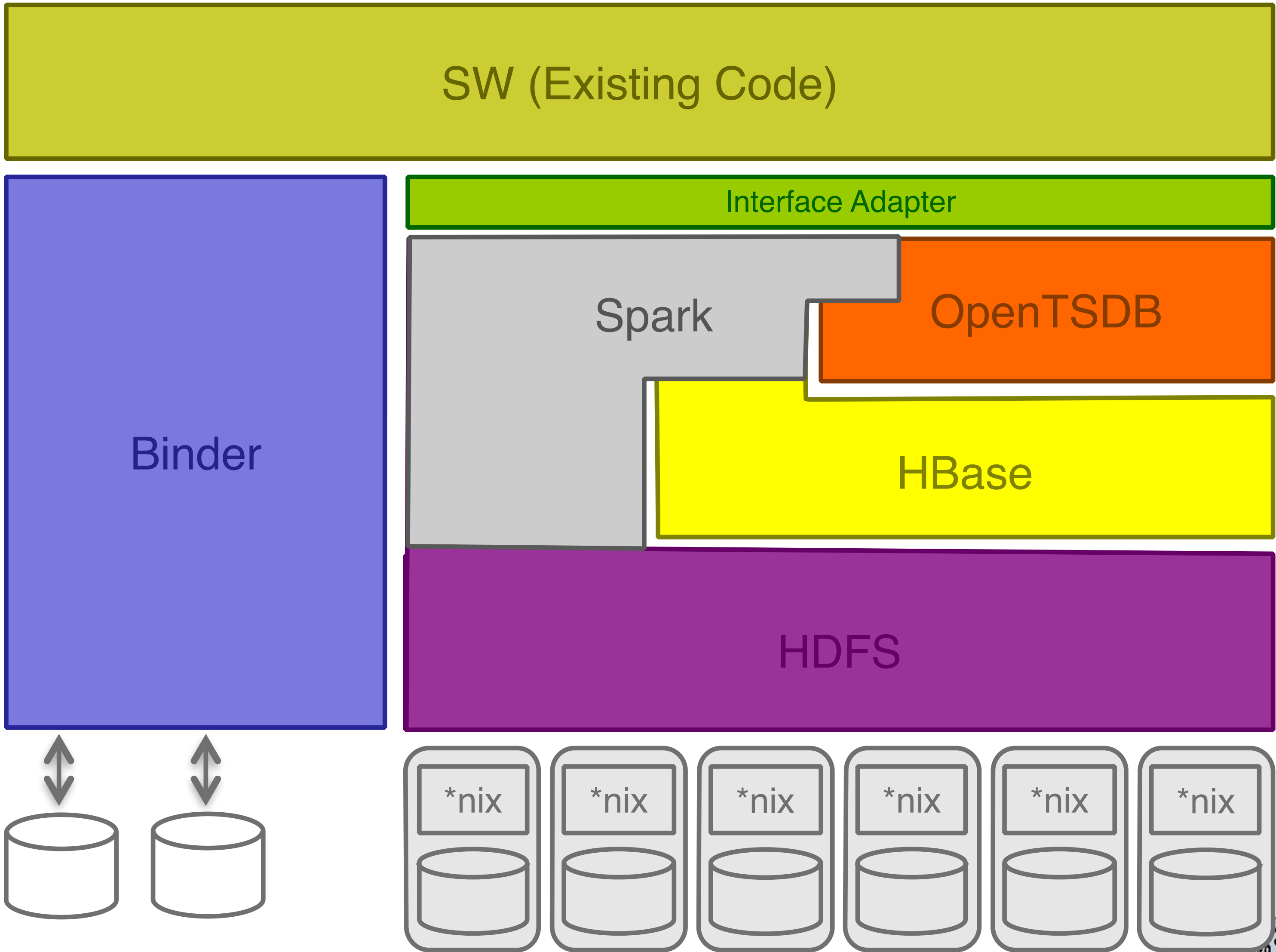


Accessing Data with OpenTSDB

- Steps to transform raw data into processed data:
  - selection
  - grouping
  - down-sampling
  - aggregation
  - interpolation
  - rate conversion

Step 3

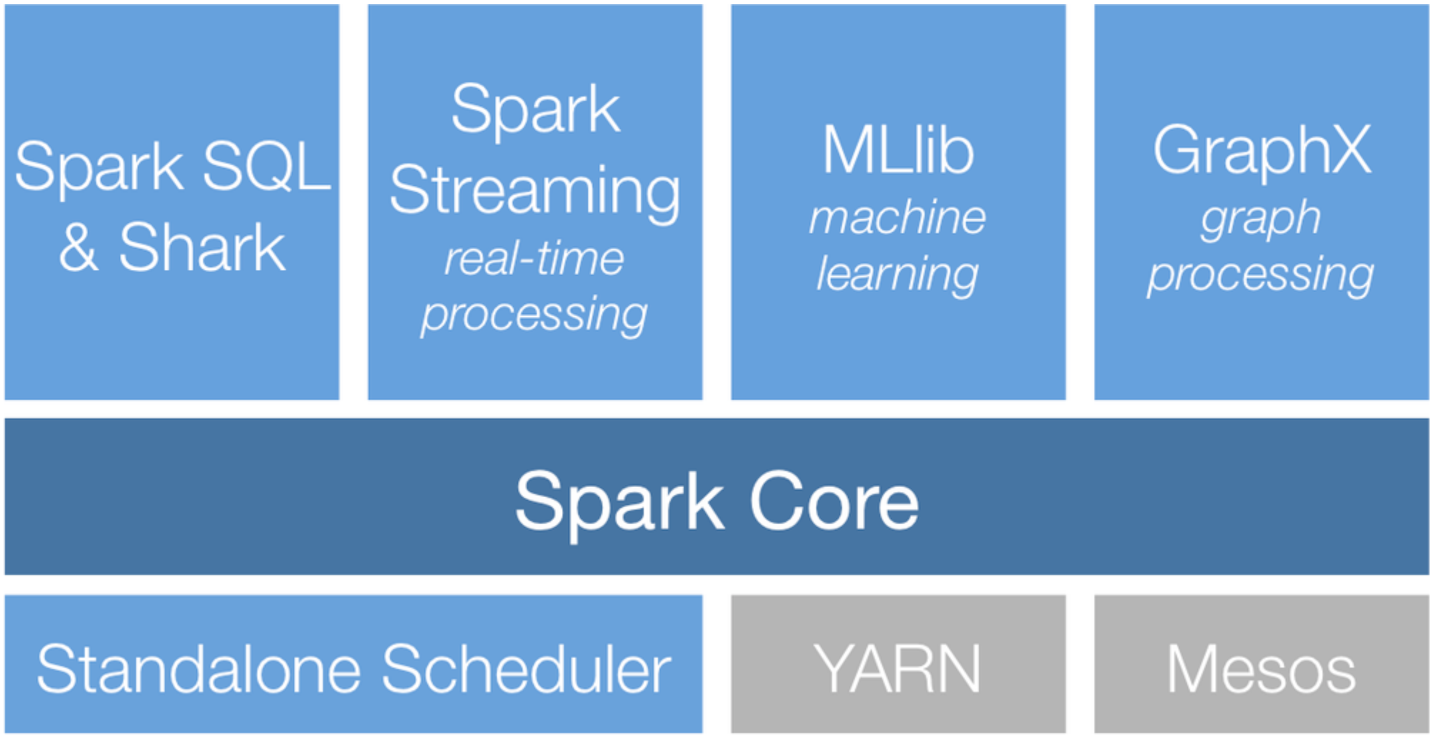
Place a unified mediator to distribute the job across nodes and different layers



SPARK

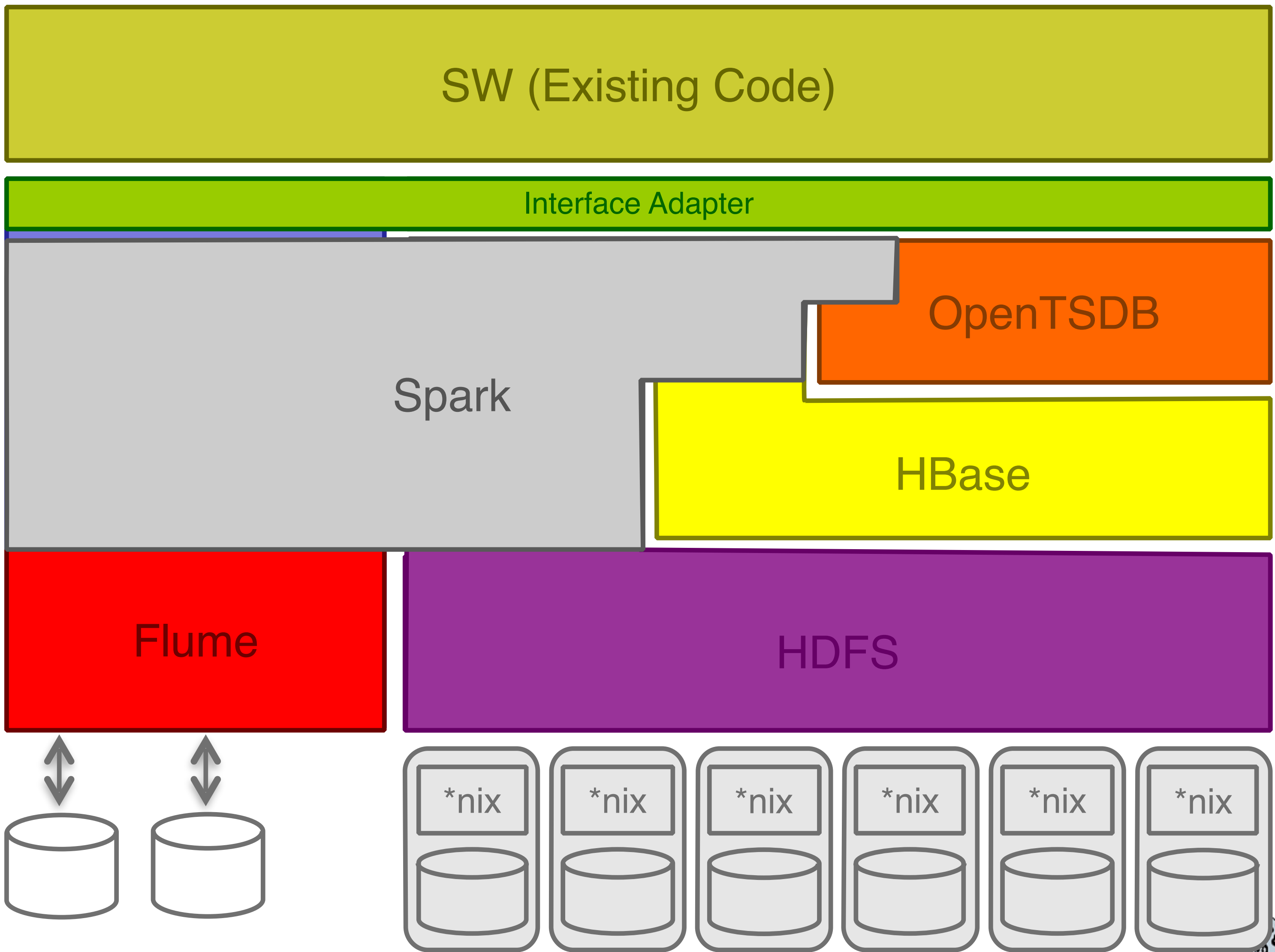
- Open source project at the U.C. Berkeley AMPLab
- Initially developed for two applications where keeping data in memory helps
  - iterative algorithms (common in machine learning)
  - interactive data mining
- Compared to MapReduce, Spark differs in two things
  - Spark holds intermediate results in memory (rather than writing them to disk)
  - Spark supports more than just map and reduce functions (greatly expanding the set of possible analyses that can be executed over HDFS data)

SPARK Components



Step 4

Make integration seamless

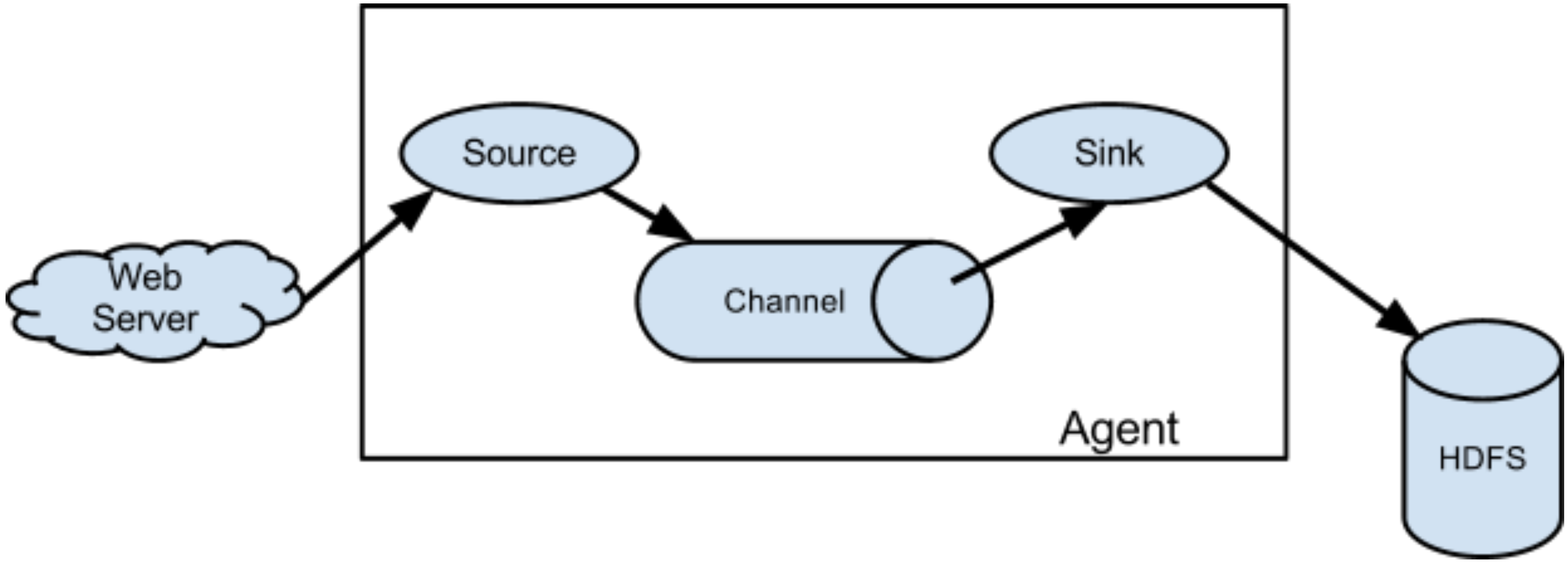


FLUME

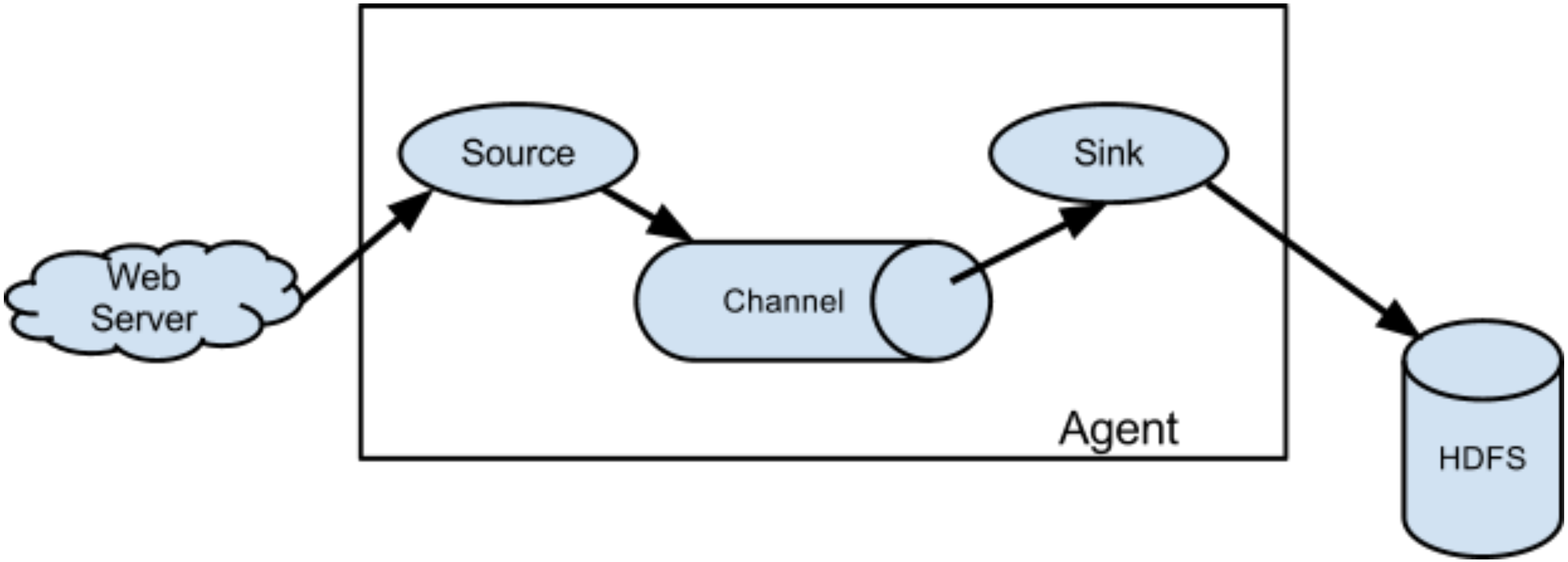
- Distributed, reliable, available service for efficiently collecting, aggregating and moving large amounts of data
- Simple and flexible architecture based on streaming data flows
- Components:
  - *Event* - data being collected
  - *Flume Agent* - source (where the data comes from), channel (repository for the data), sink (next destination for the data)

Università degli Studi del Sannio

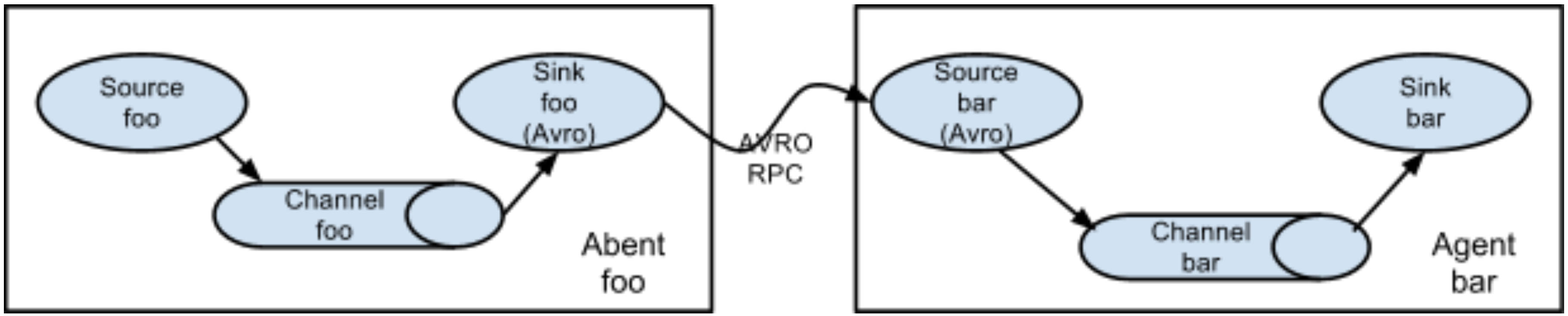
Flume Agents



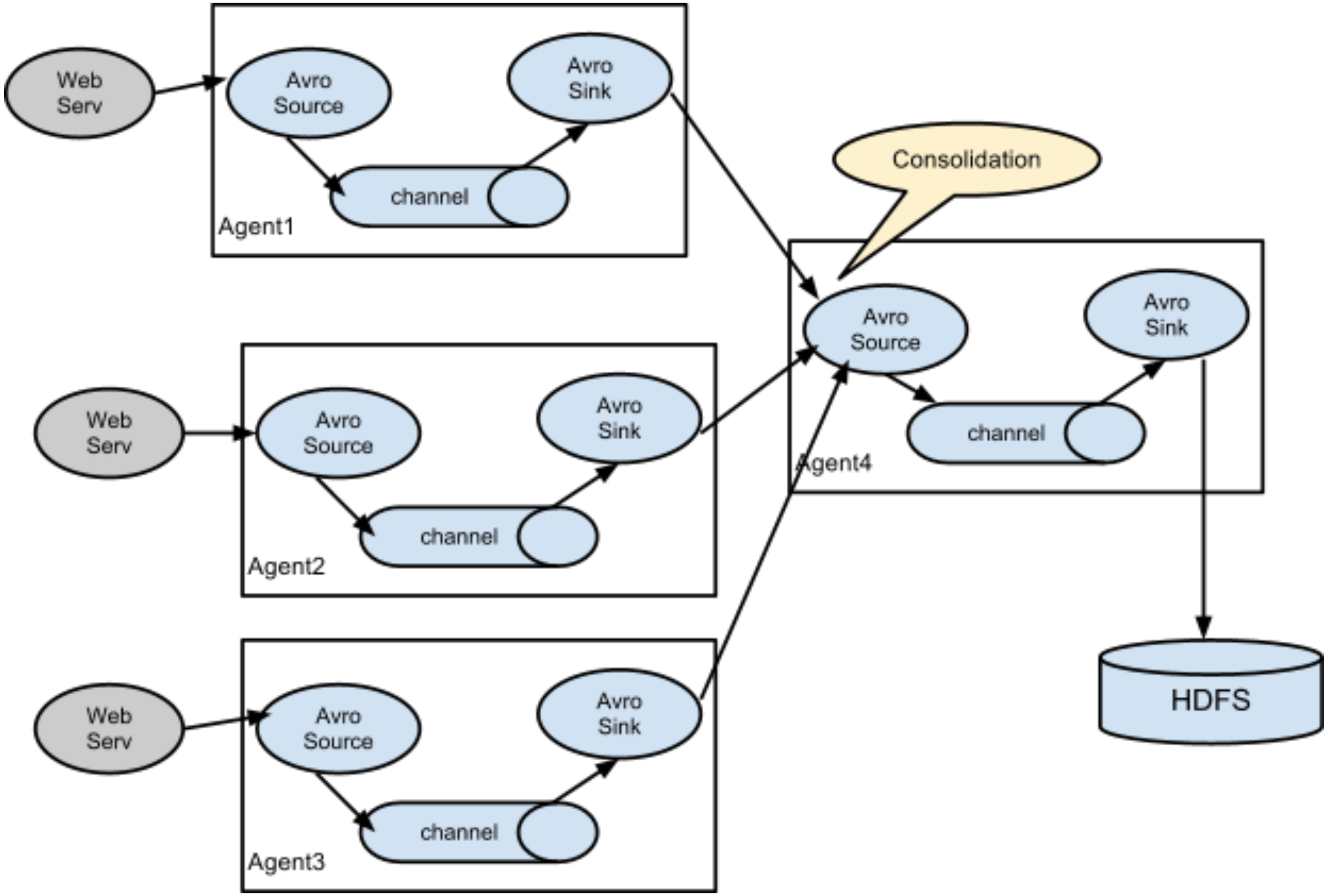
Flume Agents



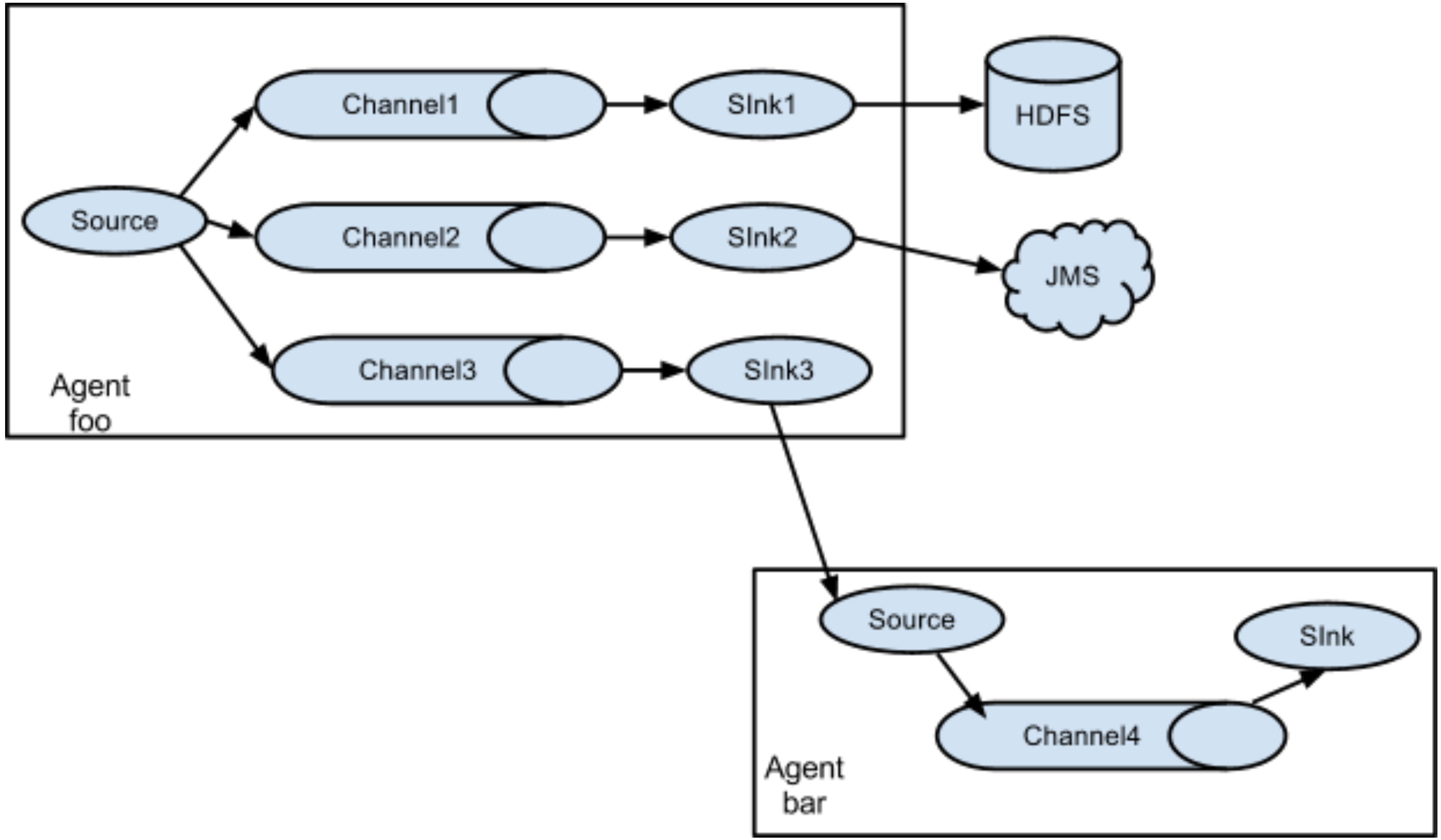
Agent Chaining



Consolidation



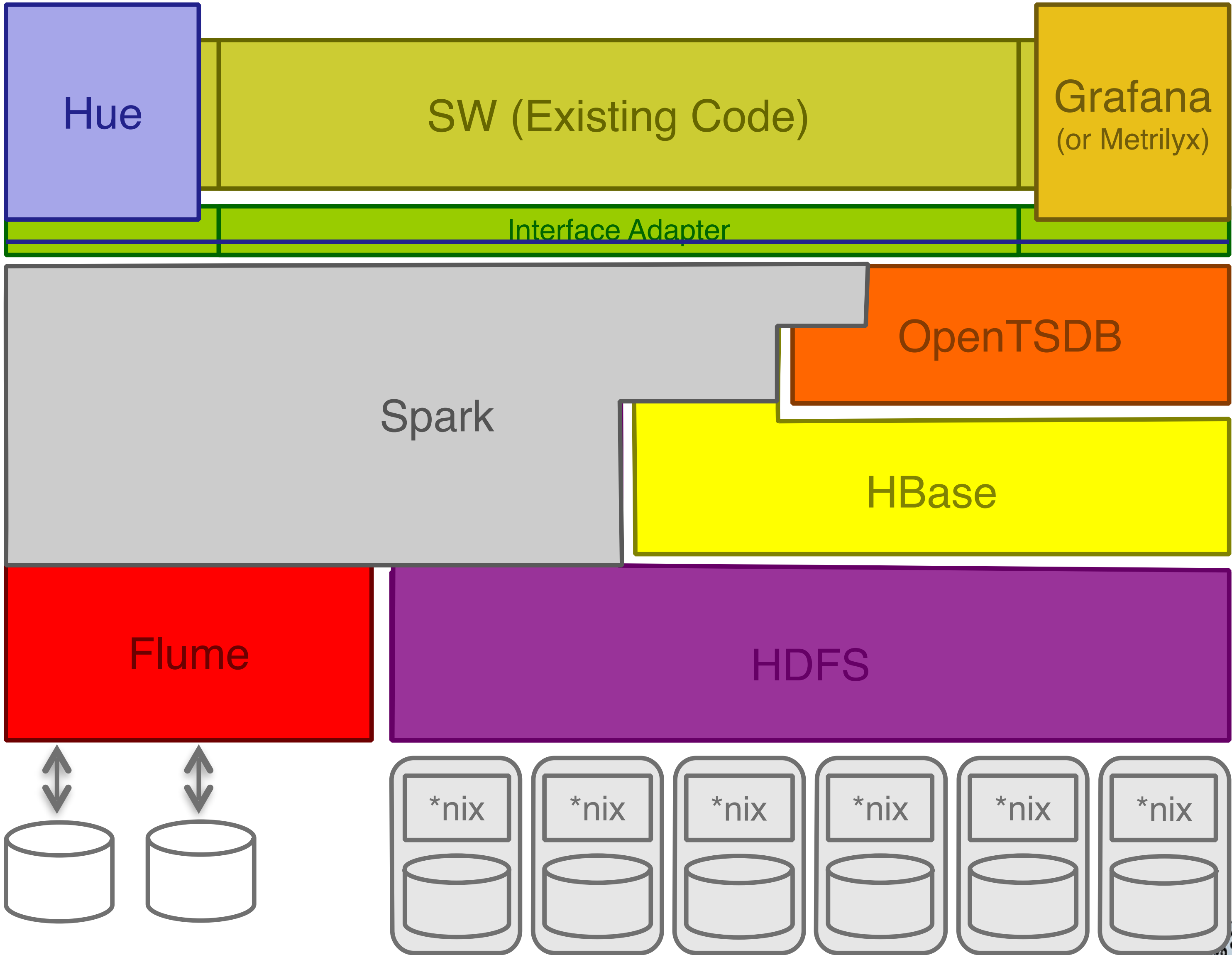
Multiplexing



Step 5

Adding top-level tools:

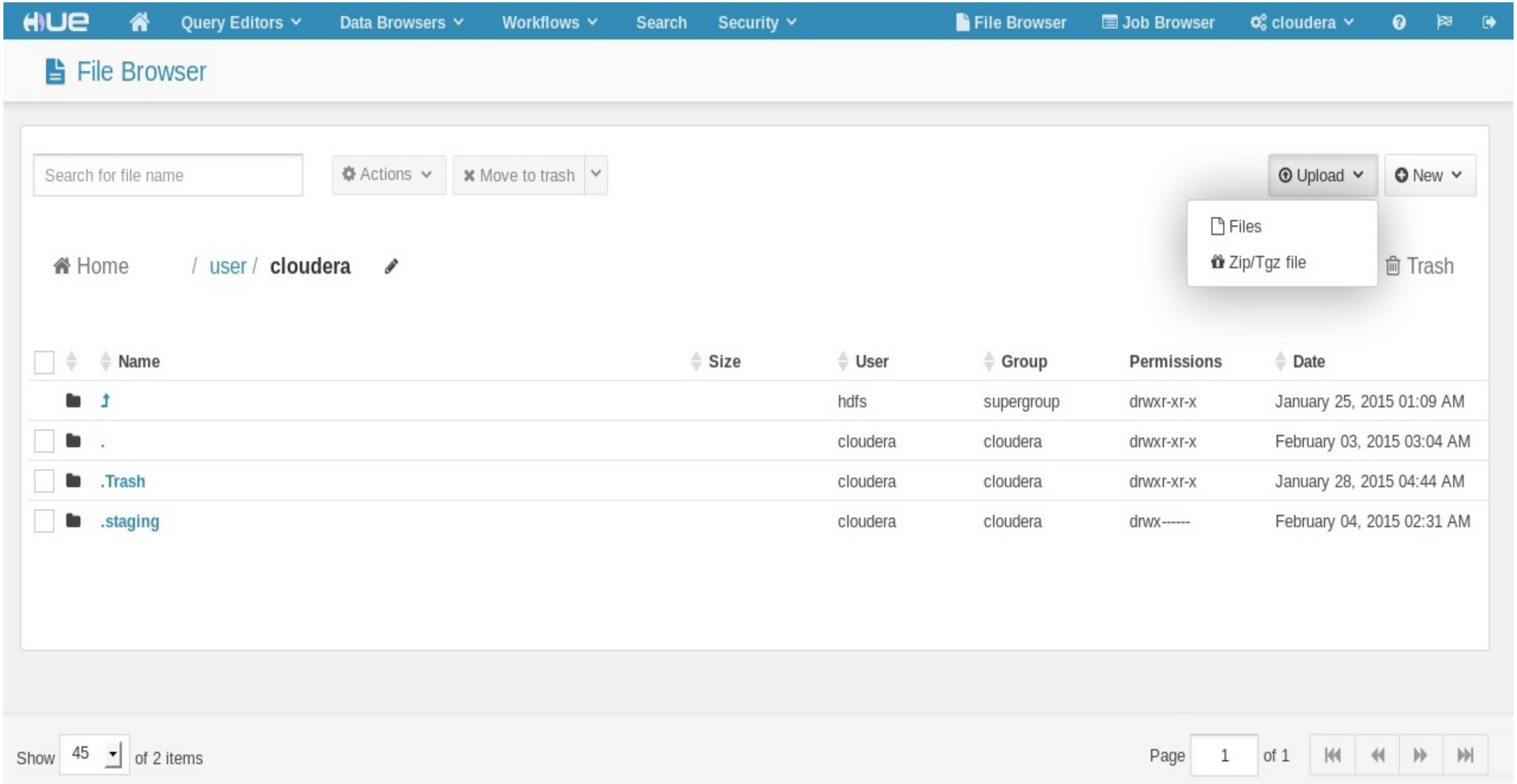
- Hue
- Grafana



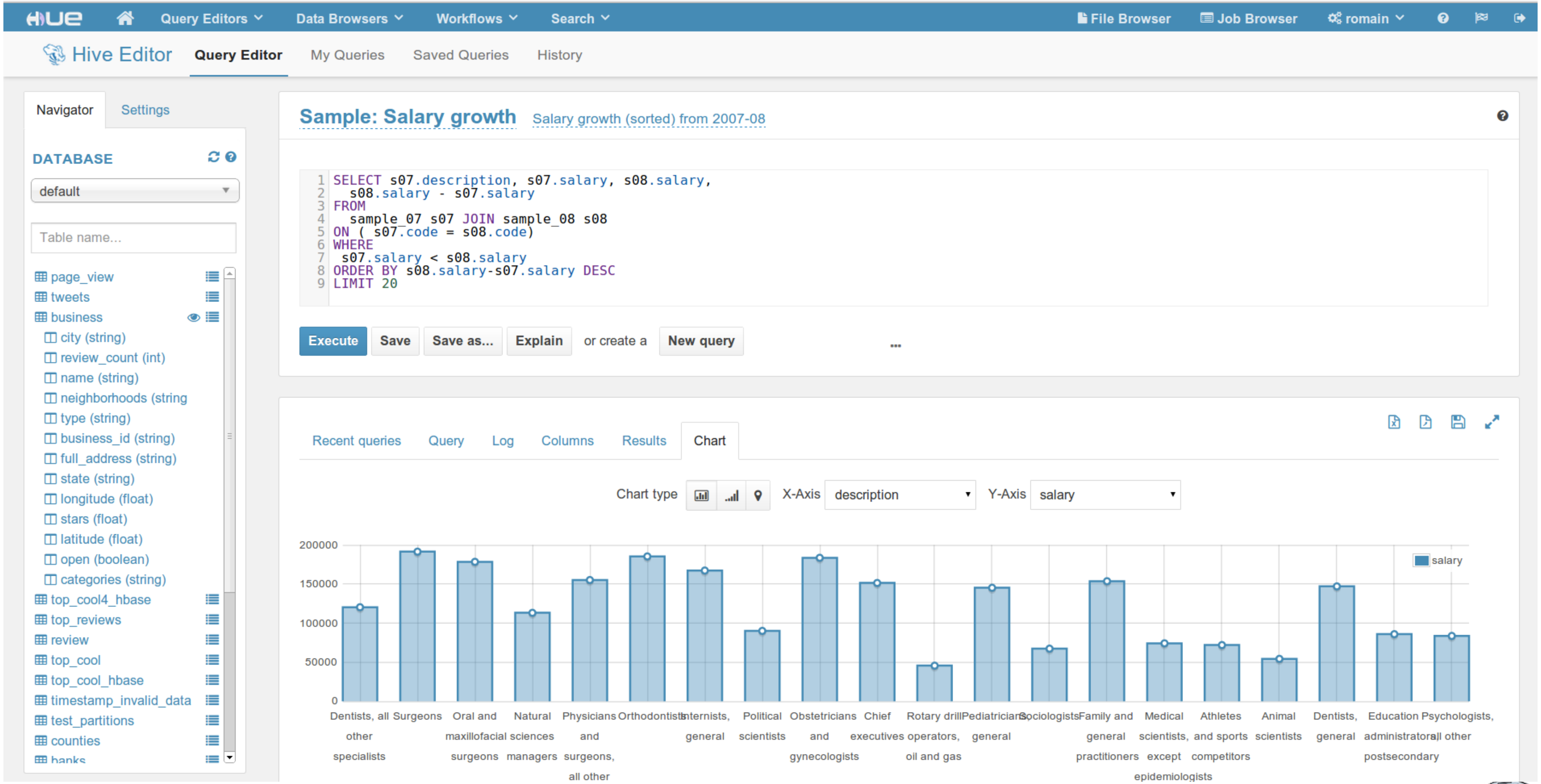
HUE

- <u>**H**</u>adoop <u>**U**</u>ser <u>**E**</u>xperience
- Graphical front-end to developer and administrator functionality
- Developed by Cloudera and released as Open Source
- Extensible (publically-available API)
- Web based
- Integrated with Hadoop tools
    - i.e. HDFS file browser, HBase browser, ZooKeeper browser, Spark editor

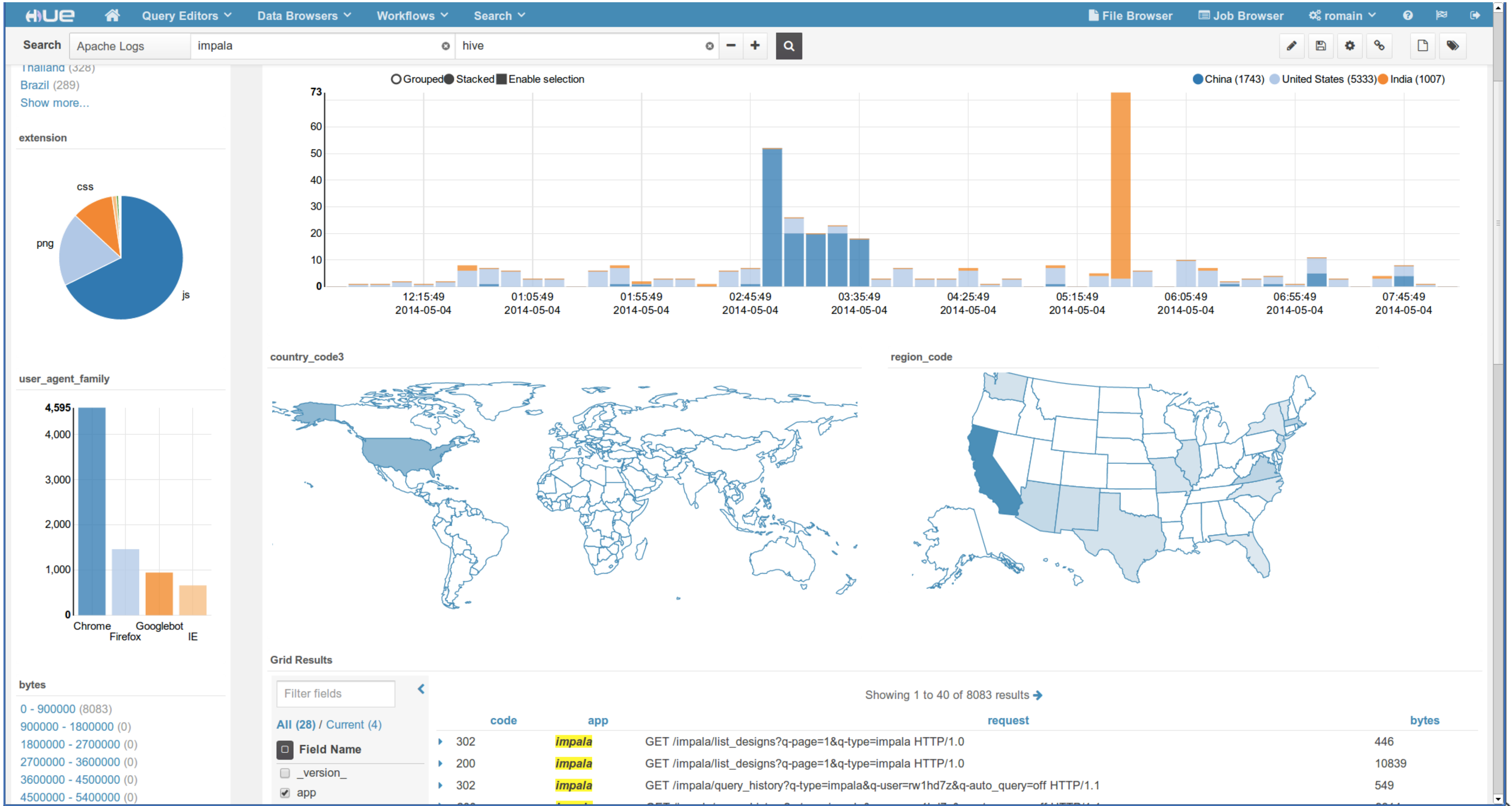
Screenshot 1



Screenshot 2



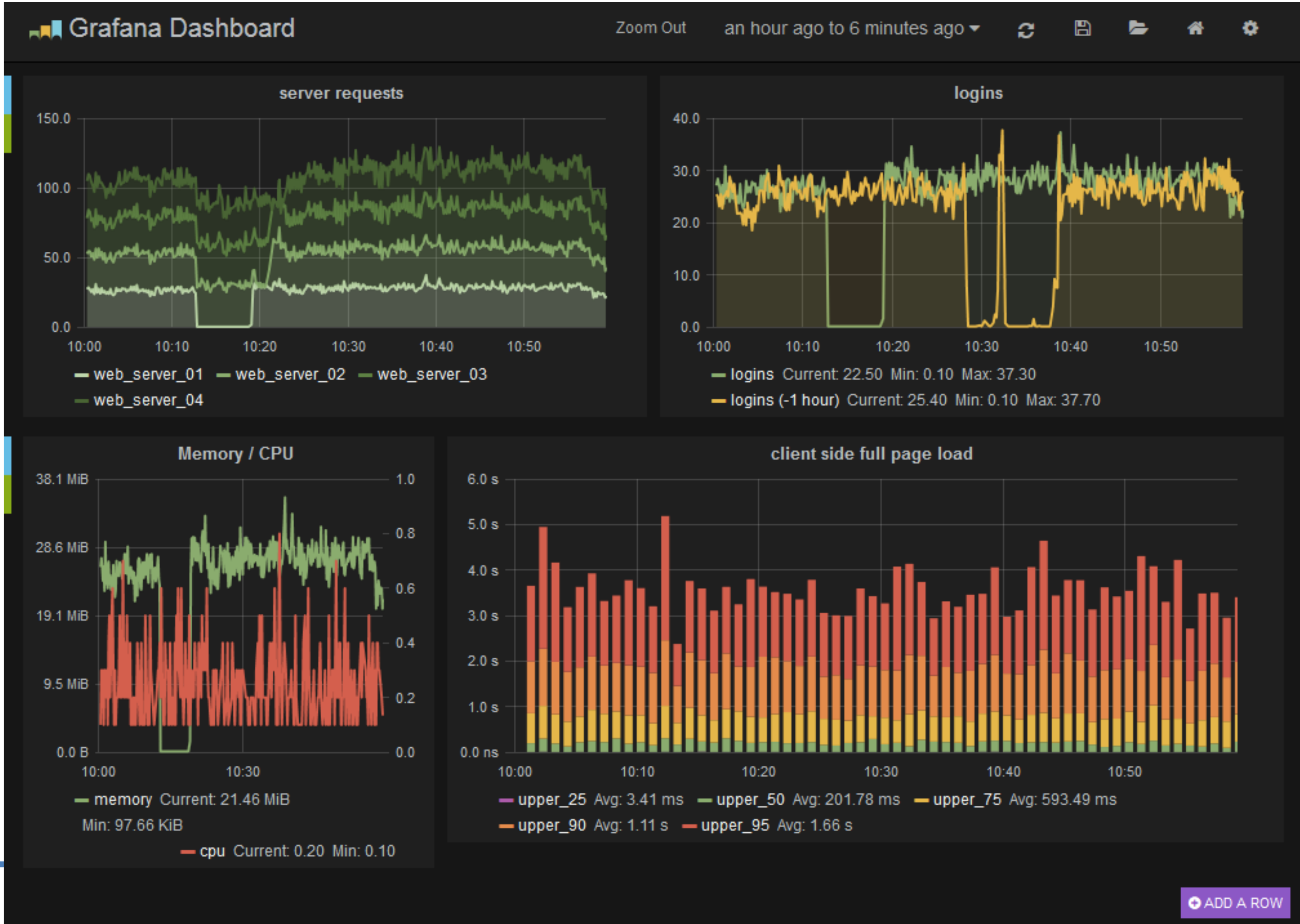
Screenshot 3



Grafana

- Open source
- Feature rich metrics dashboards and graph editor for OpenTSDB
  - Rich graphing: fast and flexible client side graphs with a multitude of options
  - Mixed styling
  - Dashboards: drag and drop panels, change row and panel width easily
  - Annotations

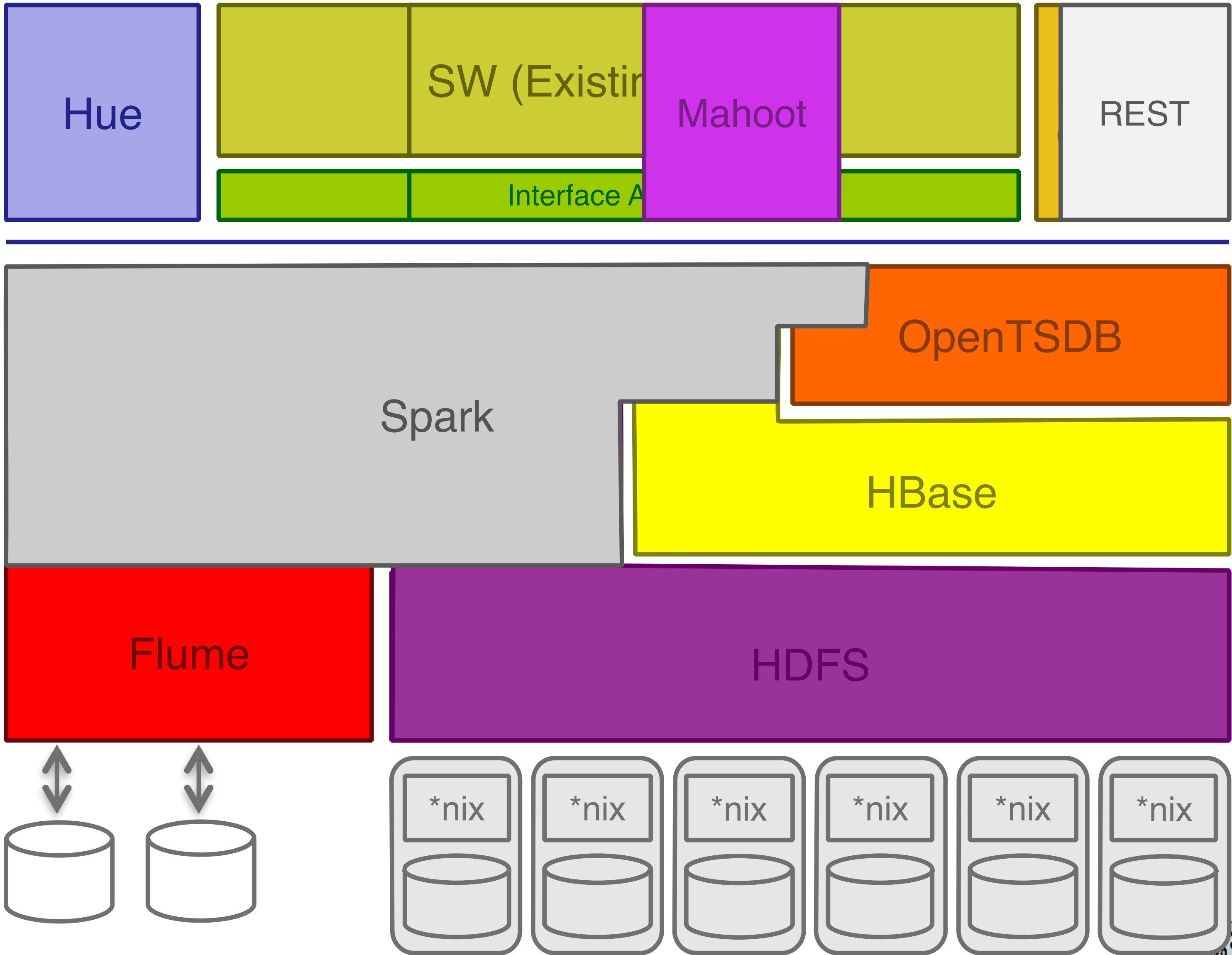
Grafana: A screenshot



Step 5

Empowering the architecture:

- Machine Learning
- Third-party Access
- … and more



MAHOUT

Distributed and scalable **machine learning algorithms on the Hadoop platform:**

- ~~Collaborative filtering~~
- **Clustering**
- **Classification**
- **Dimensionality Reduction**

Use cases:

- Yahoo —> spam detection
- Foursquare —> recommendations
- Adobe —> user targeting
- Amazon —> personalization platform

Classification:

- Logistic Regression - trained via SGD
- Naive Bayes / Complementary Naive Bayes
- Random Forest
- Hidden Markov Models
- Multilayer Perceptron

Clustering:

- k-Means Clustering
- Fuzzy k-Means
- Streaming k-Means
- Spectral Clustering

Dimensionality Reduction:

- Singular Value Decomposition
- Lanczos Algorithm
- Stochastic SVD
- PCA (via Stochastic SVD)
- QR Decomposition

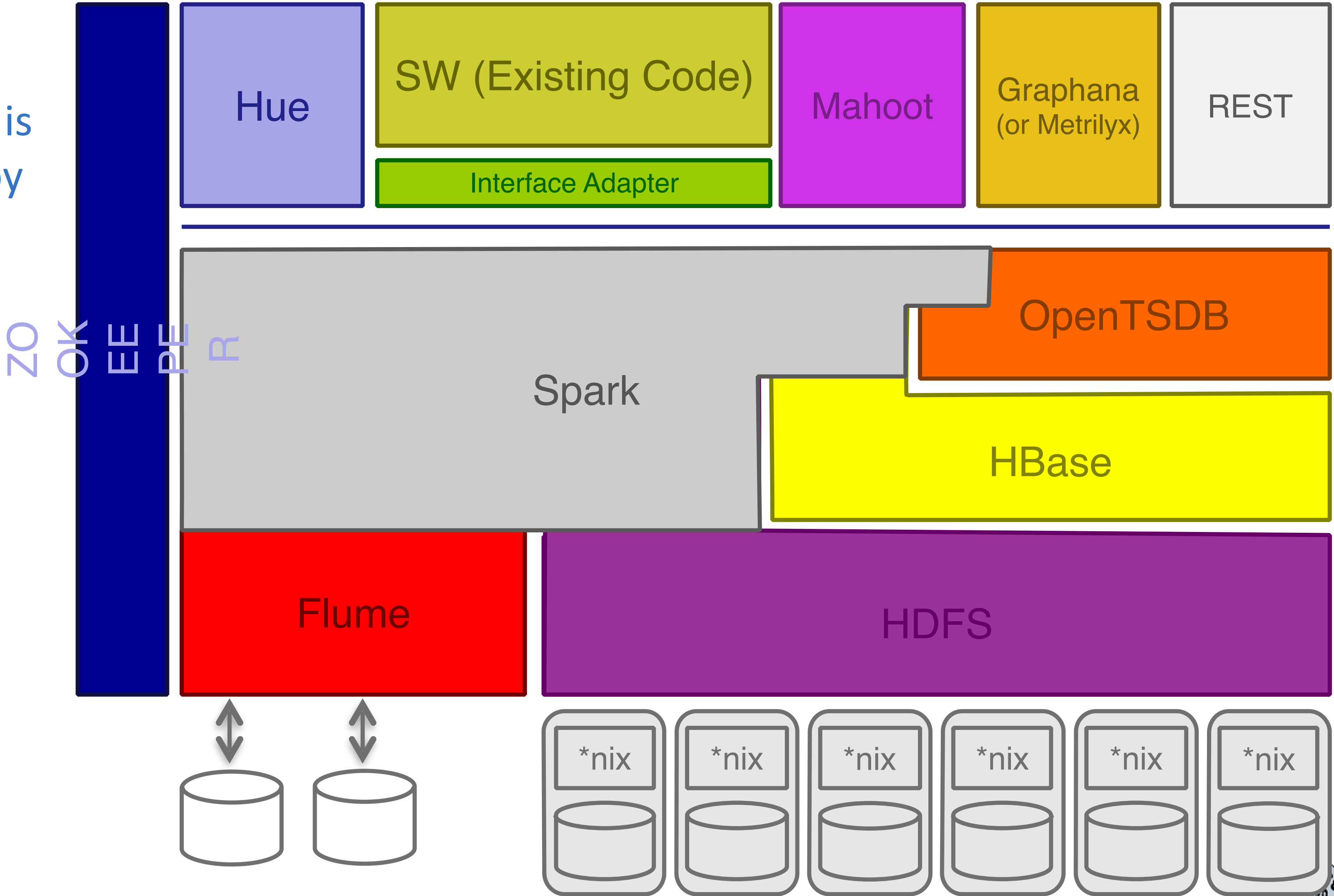
REST APIs

Make easier the integration with third-parties, including:

- **External software**
- **Scientific Partners**
- **Data Analysis Tools**
- **…**

Final Result

The overall architecture is mantained by ZOOKEPER



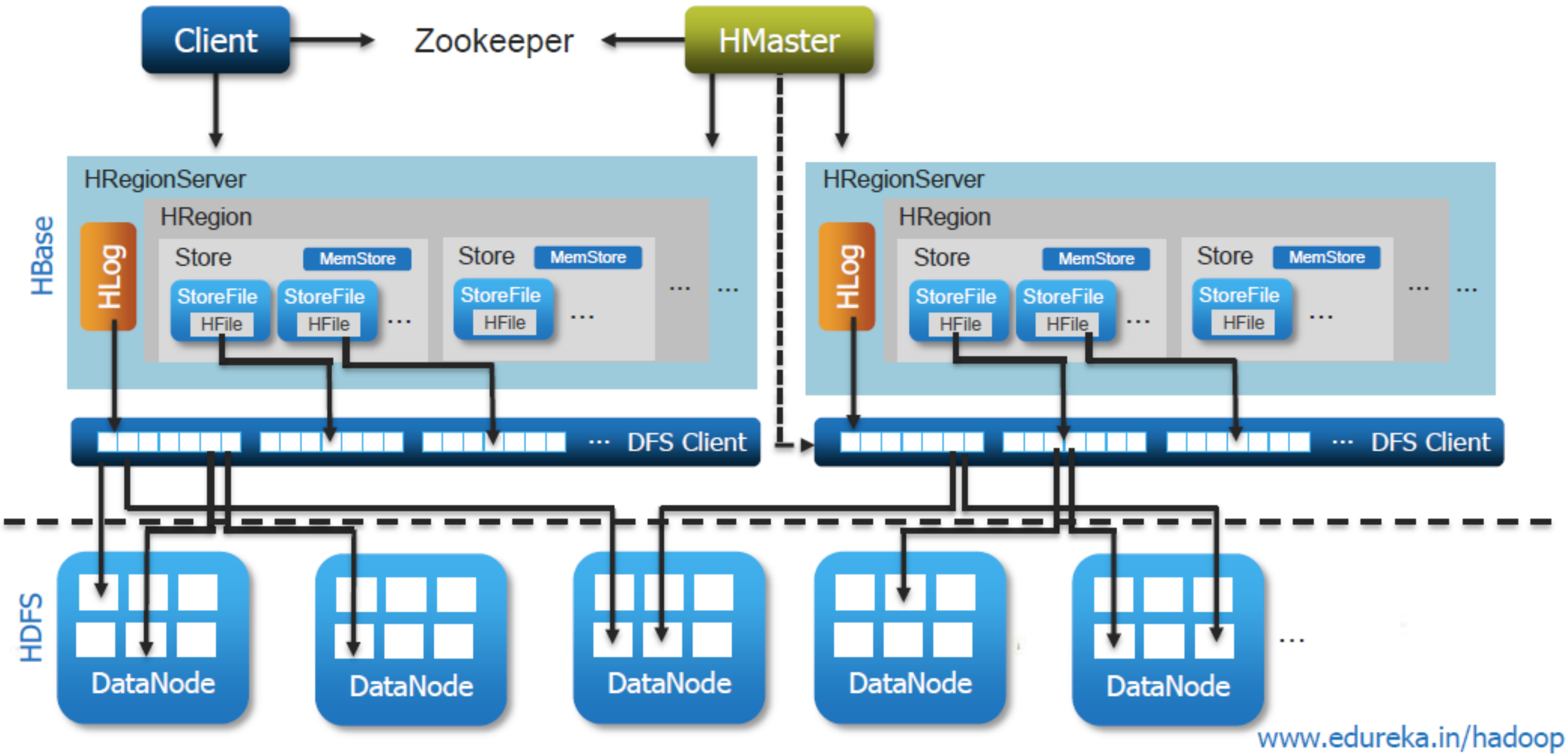
ZOOKEEPER

- *“ZooKeeper allows distributed processes to coordinate with each other through a shared hierarchical name space of data register”* (ZooKeeper Wiki)

- An open source, high-performance coordination service for distributed applications

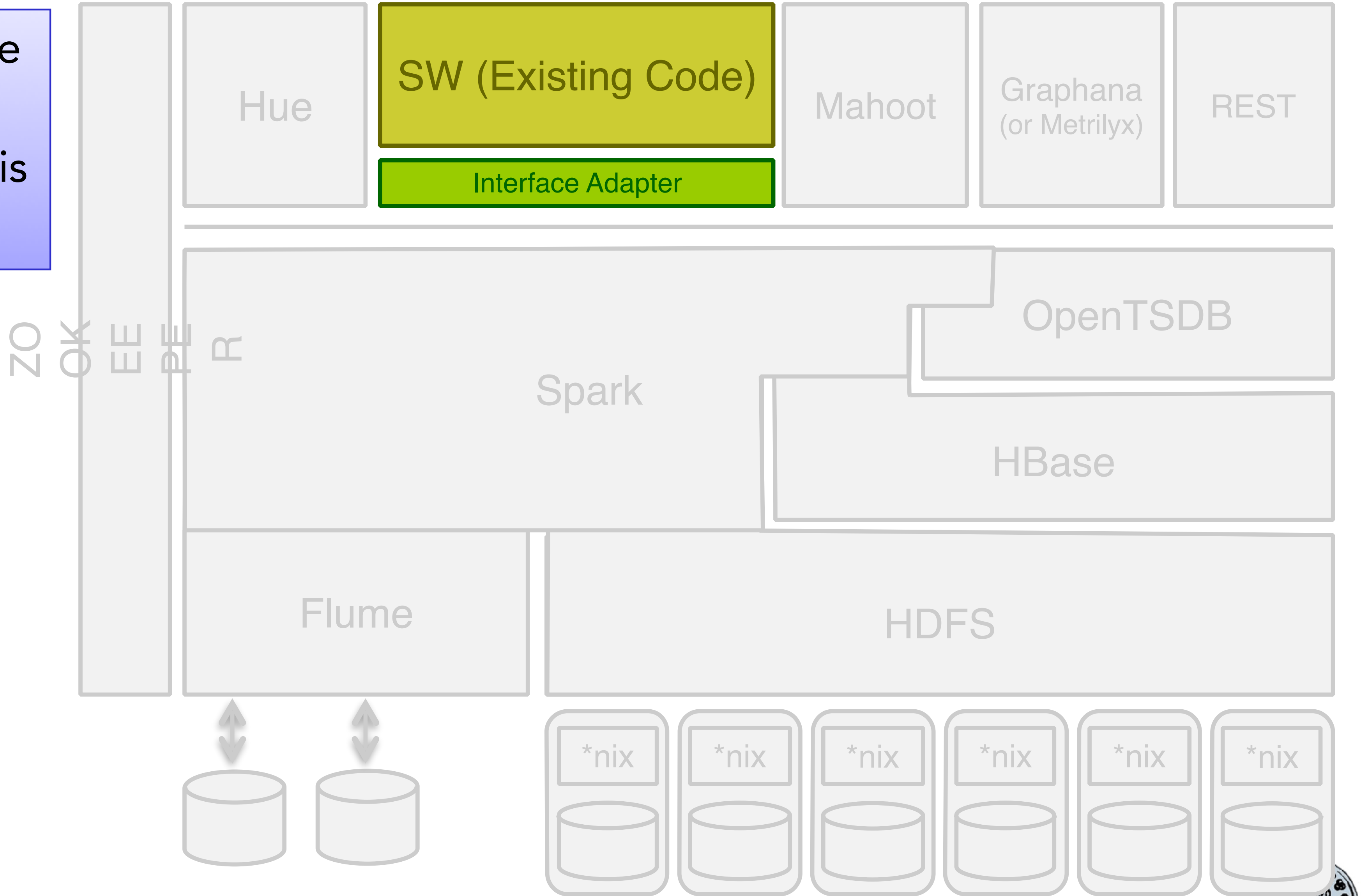
- Service for maintaining configuration information, naming, providing distributed synchronization and providing group services

ZOOKEPER EXAMPLE

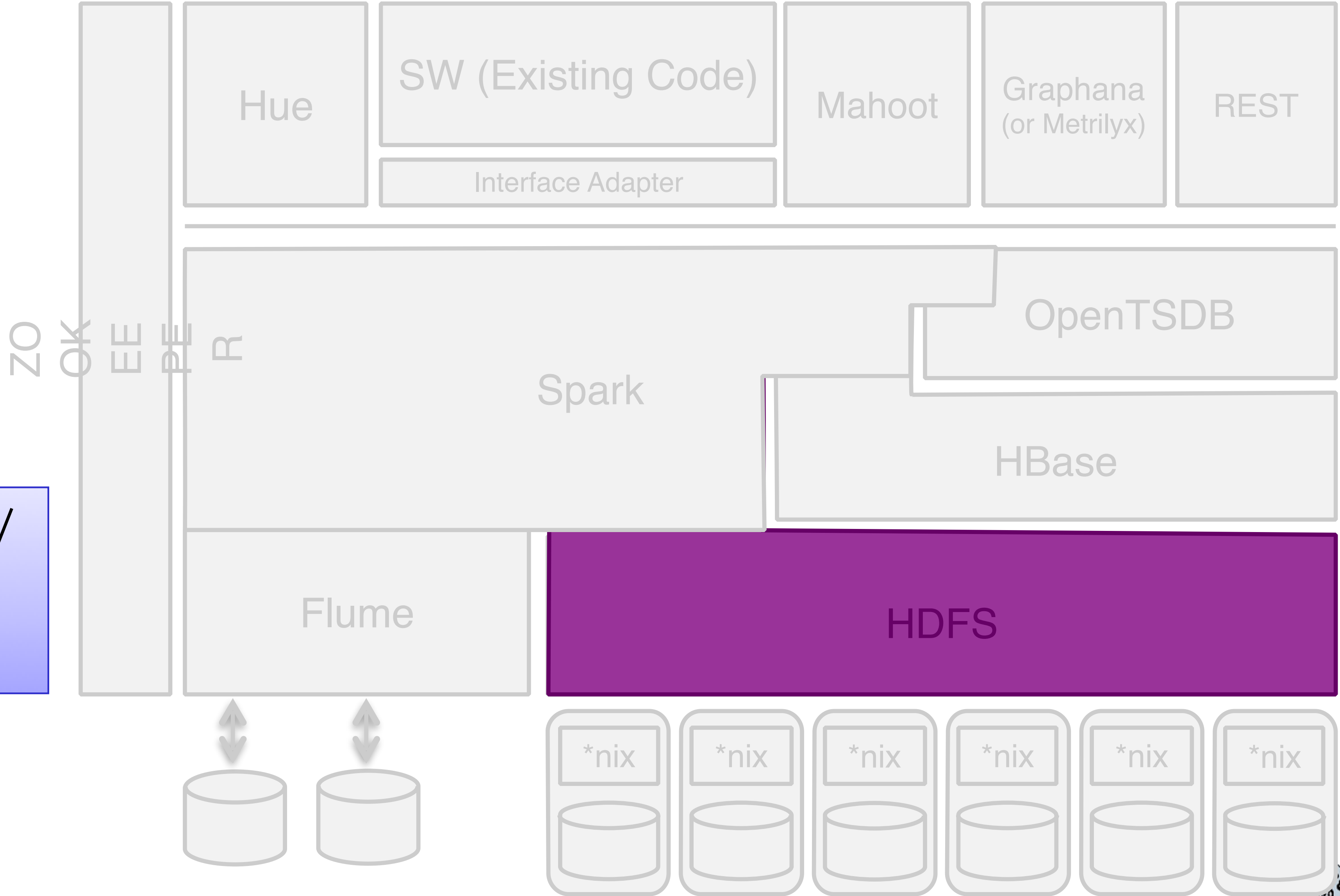


Overview: Legacy

This is where the domain knowledge is preserved.



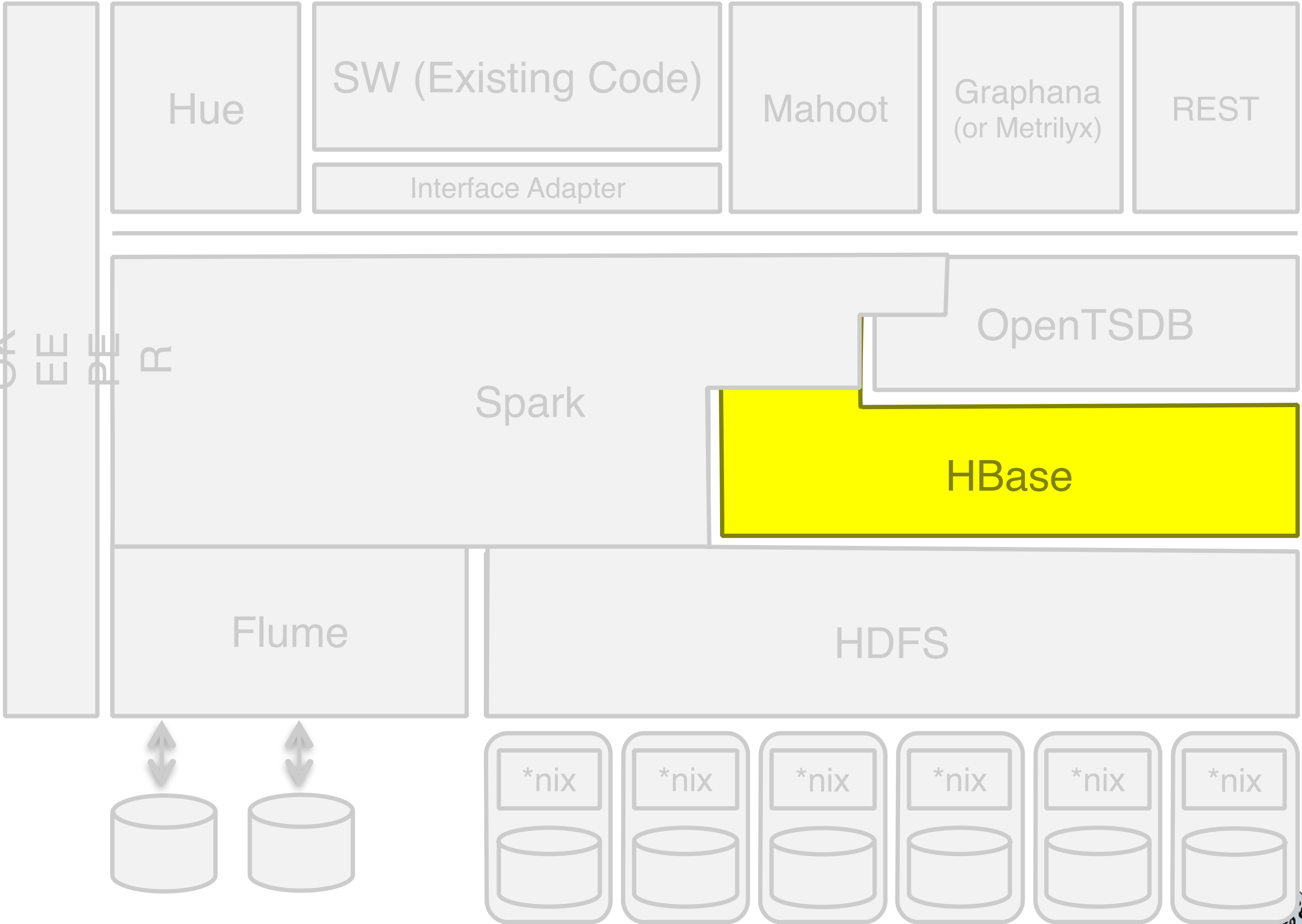
Overview: HDFS



Distributed/
Redundant
File System

Overview: HBase

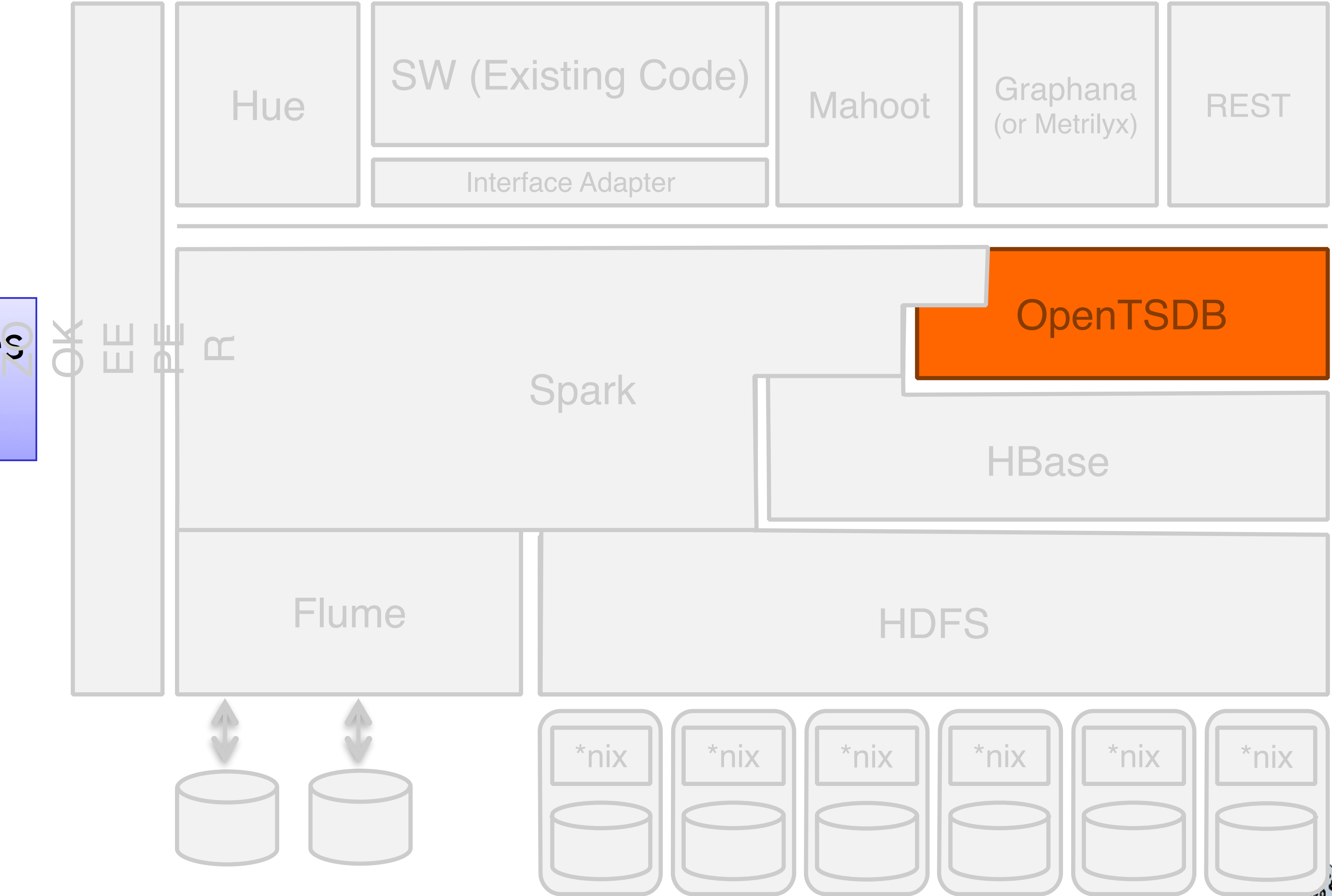
Vertical NoSQL Database



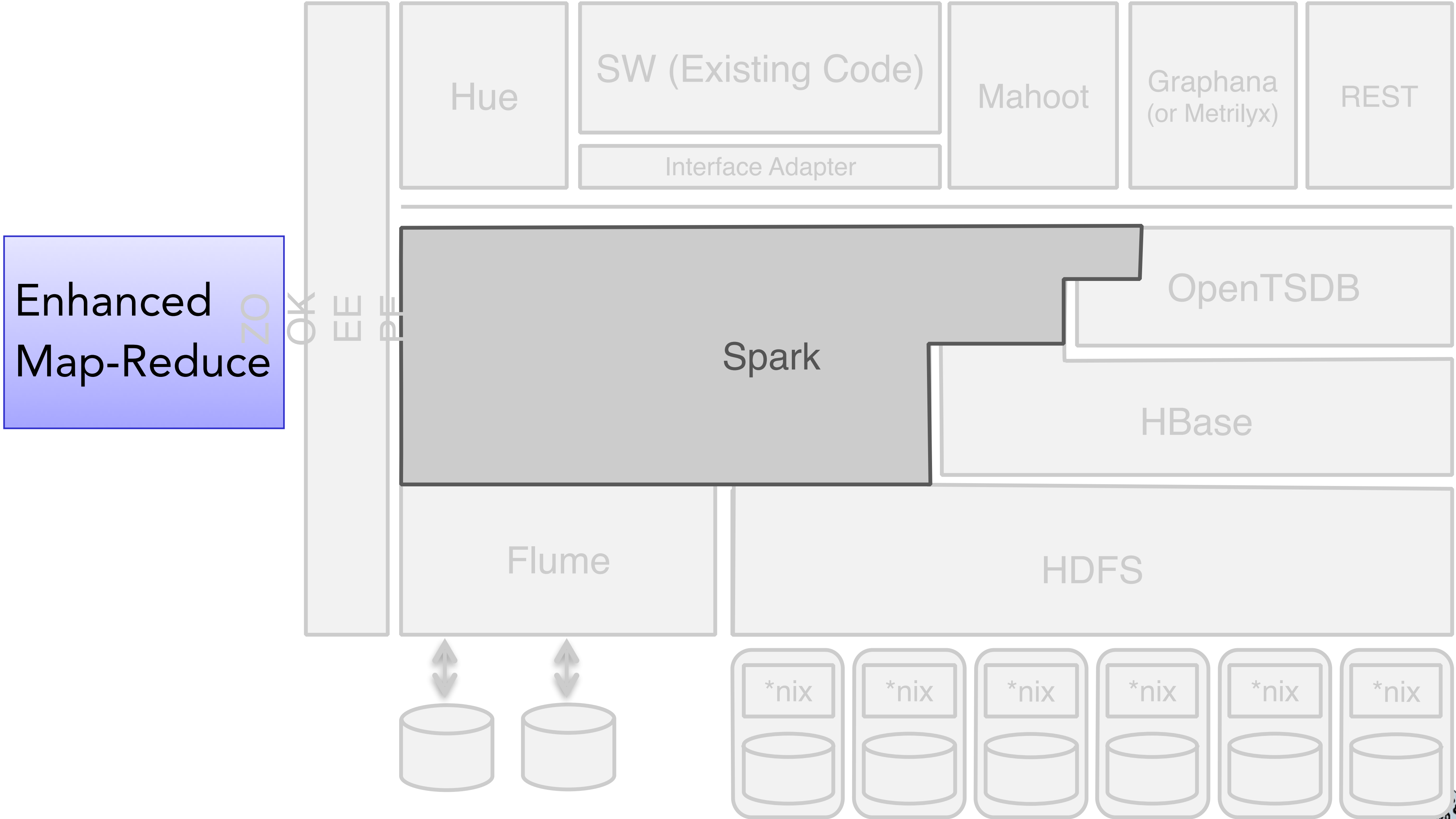
Università degli Studi del Sannio

Overview: OpenTSDB

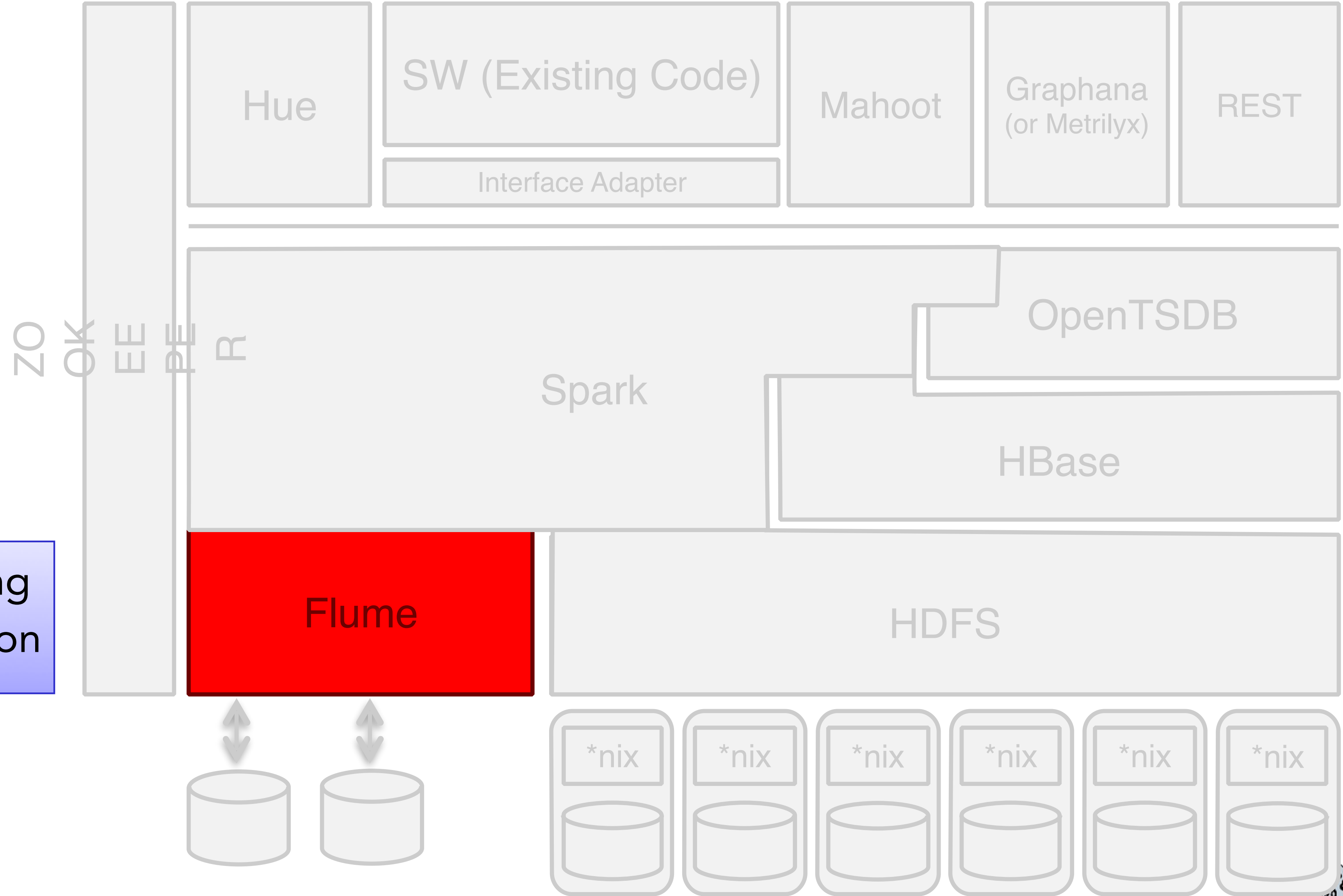
Time Series DBMS



Overview: Spark



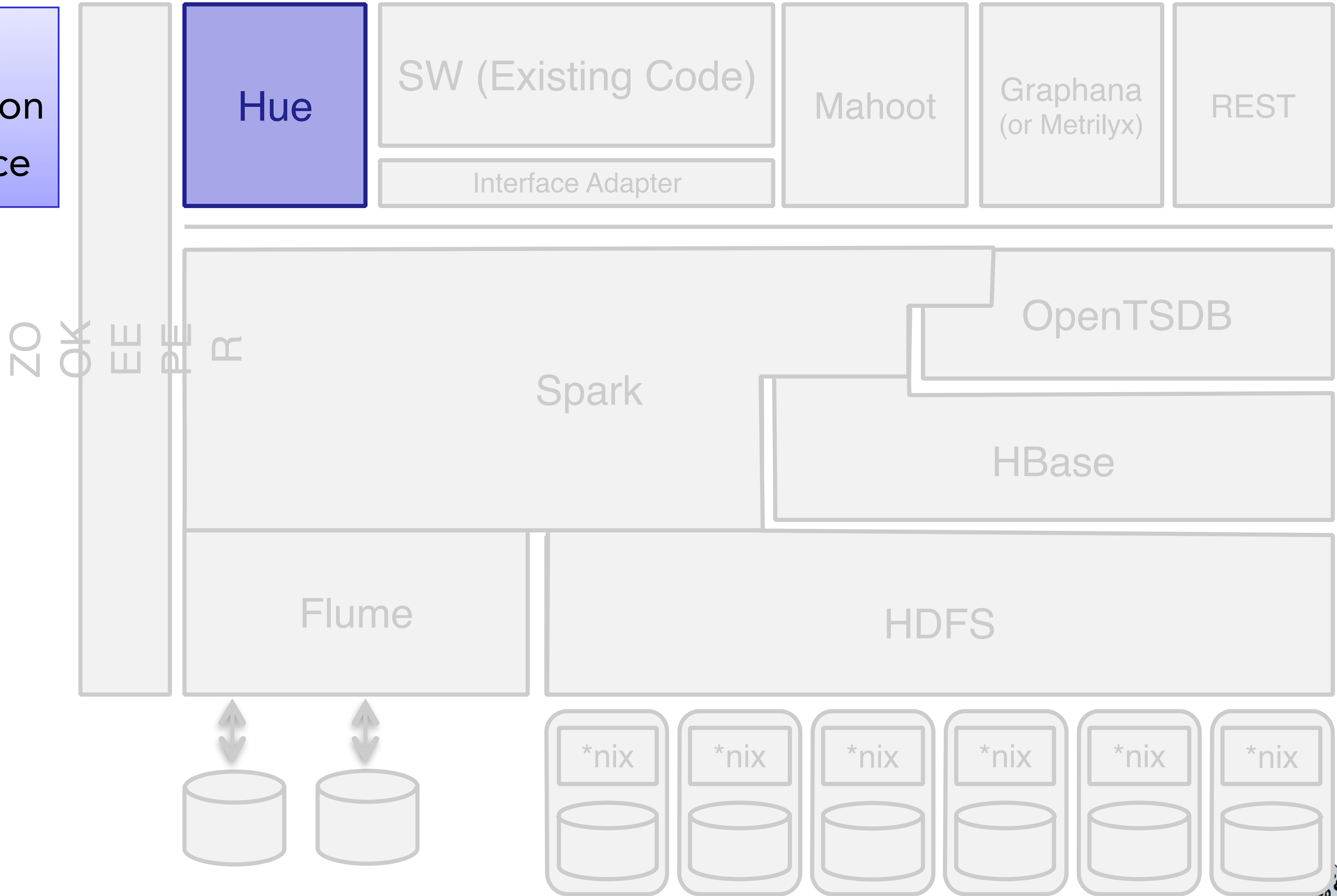
Overview: Flume



Data Gathering and Integration

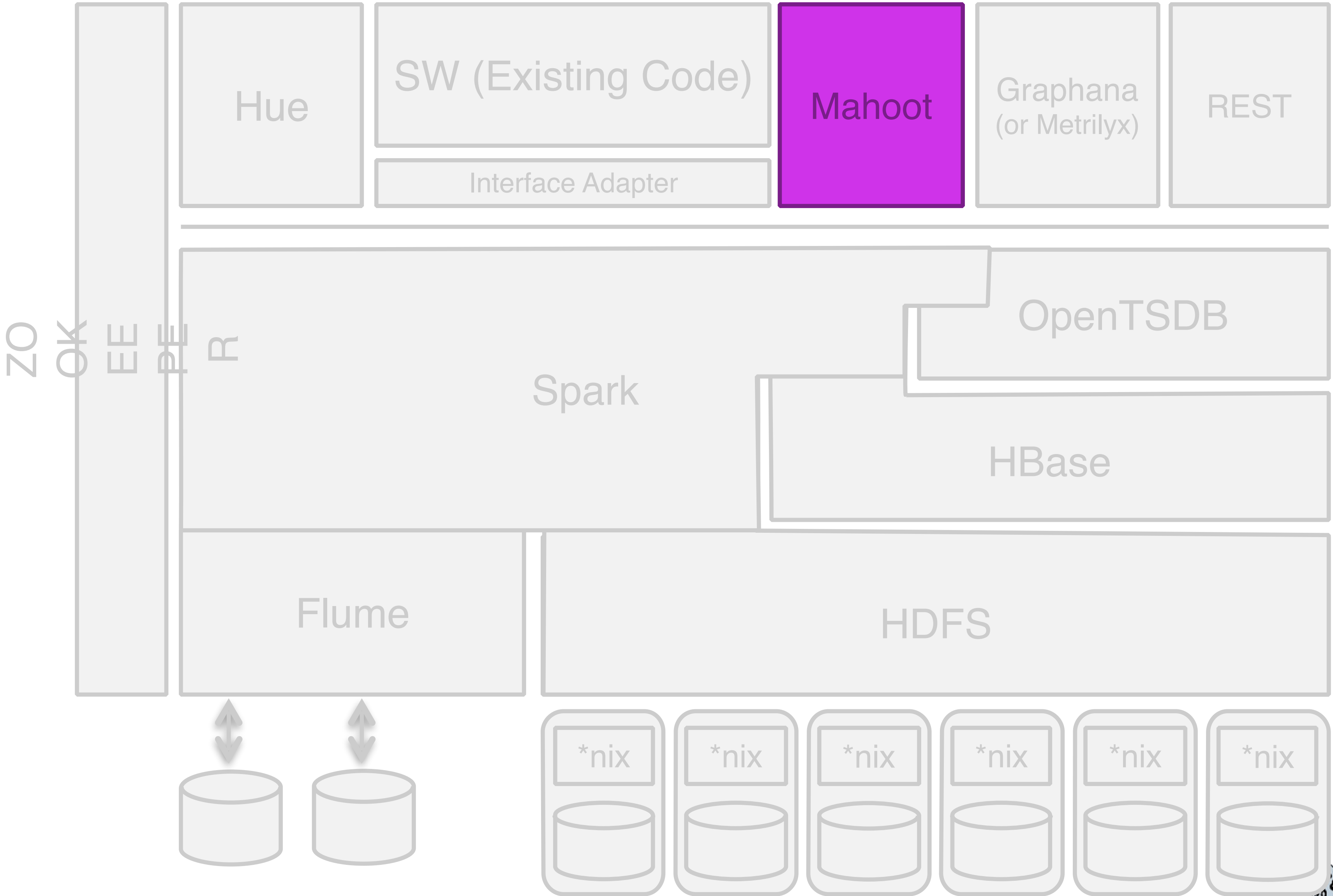
Overview: Hue

Web
Administration
User Interface



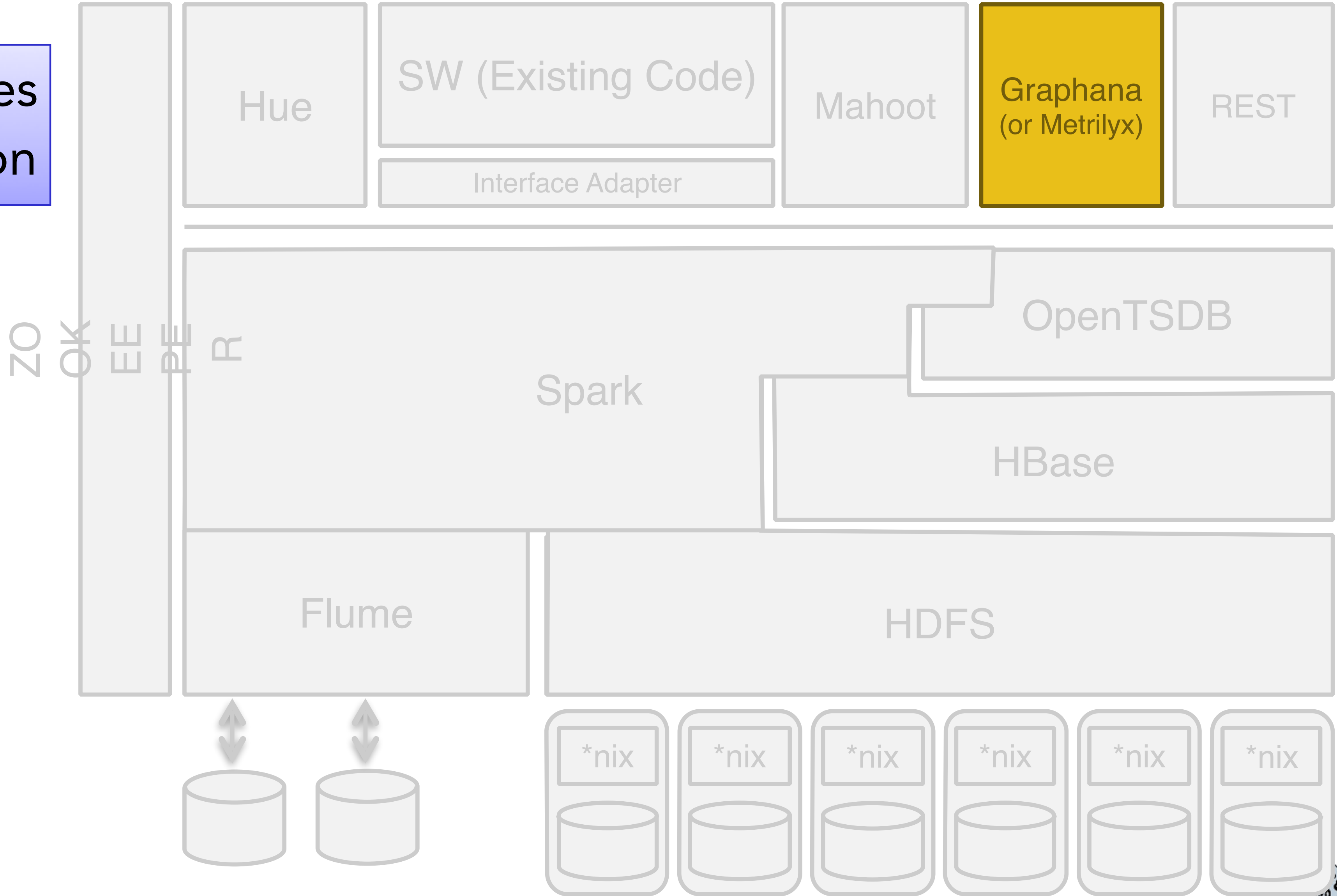
Overview: Mahoot

Machine Learning



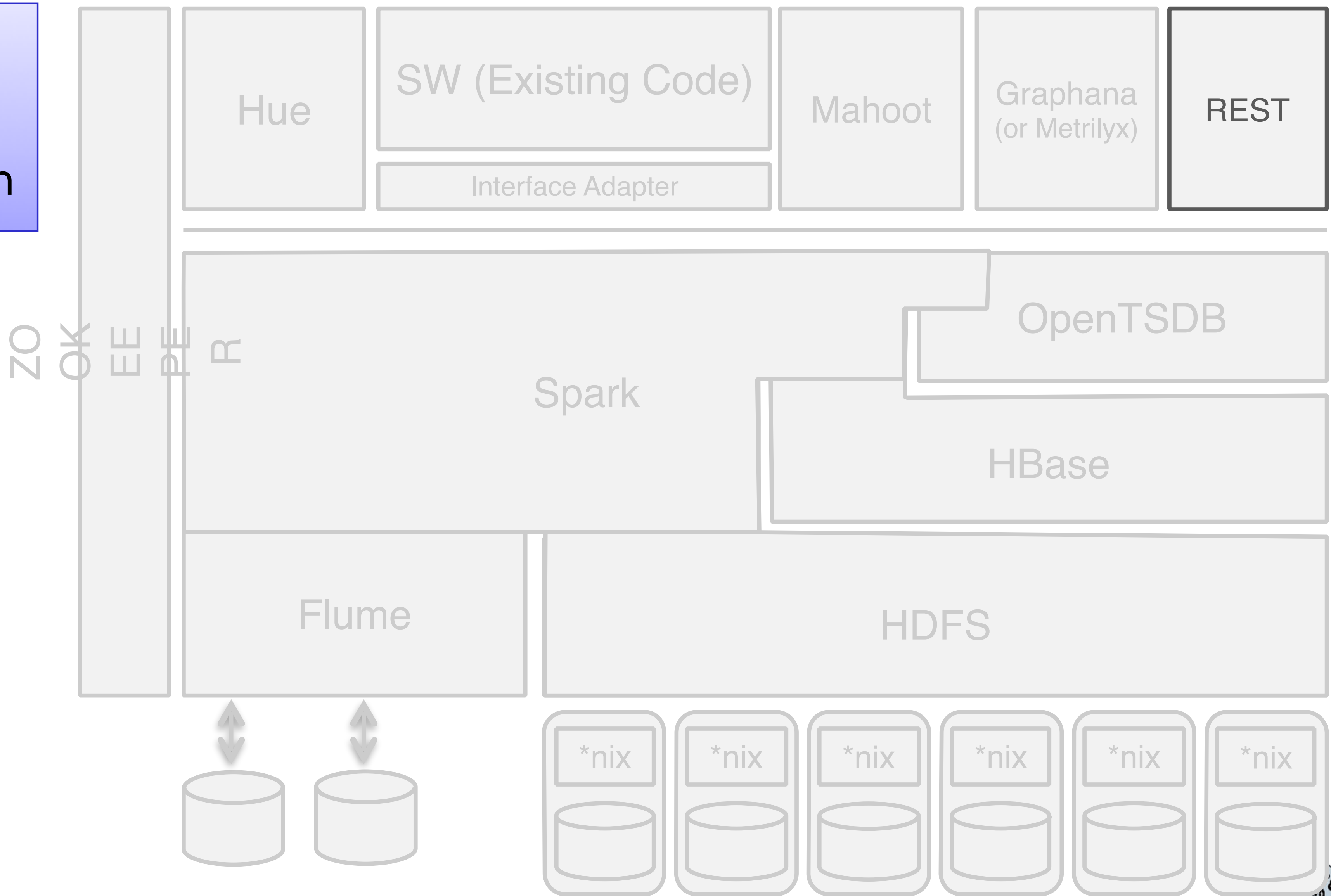
Overview: Graphana

Time Series Visualizaion

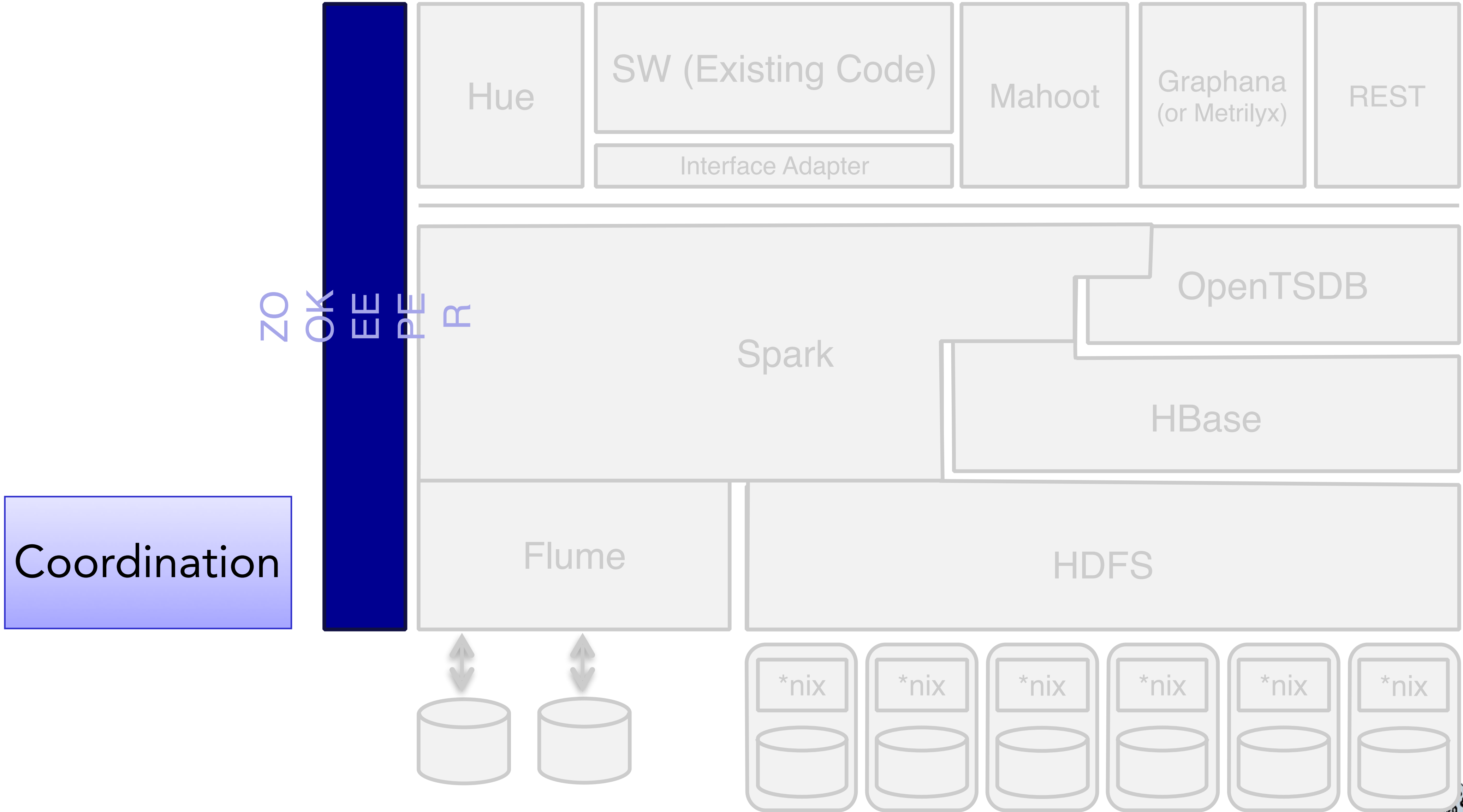


Overview: REST APIs

Third-Parties Integration



Overview: Zookeeper

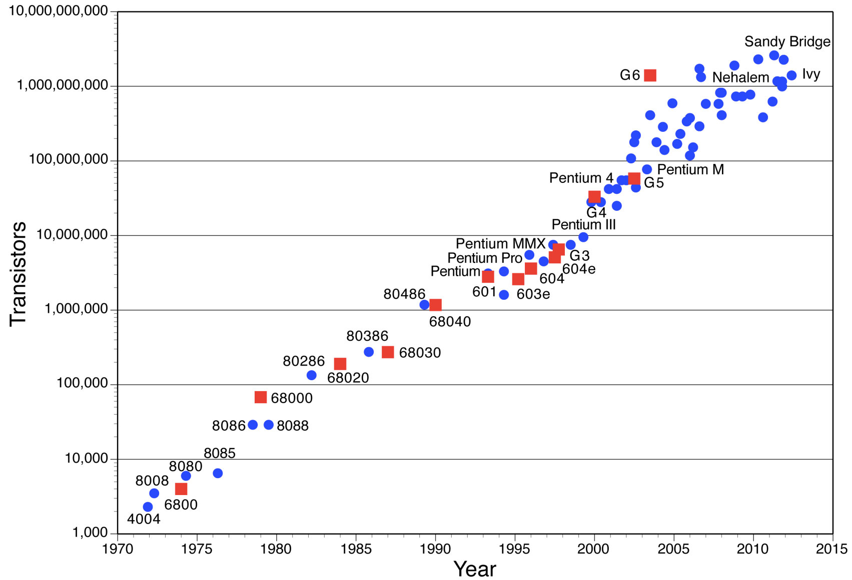


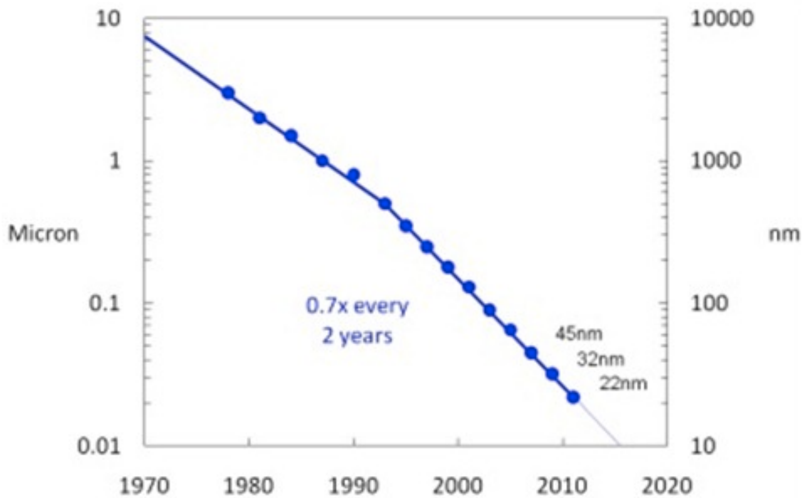
Outline

- Big Data
- Hadoop & Map-Reduce
- HDFS vs. NFS
- Evolving towards a Big Data architecture
- **FPGA co-processing**
- Conclusions

How to deal with complexity

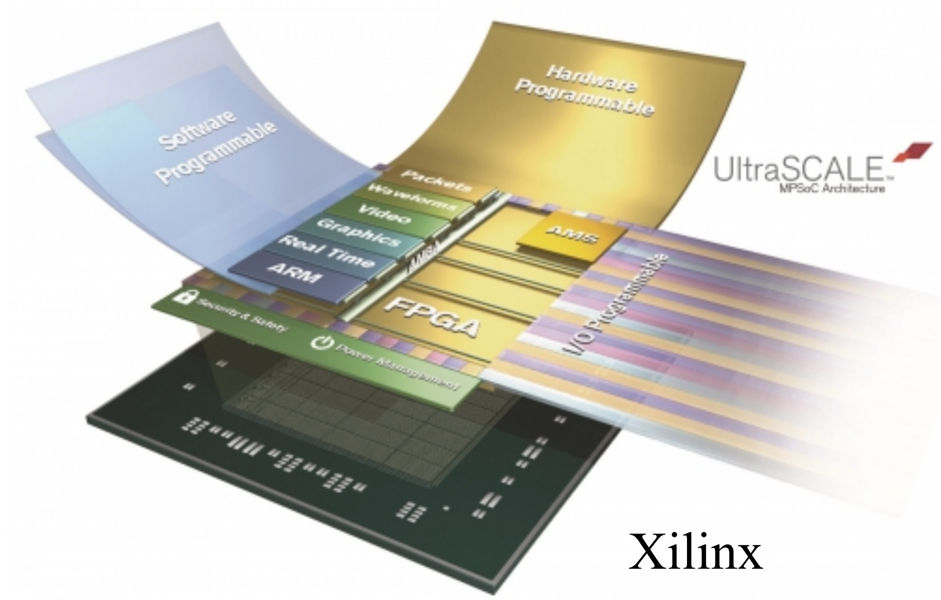
- We will need more and more computational power.
  - More Moore:
    - + integration scale
    - + clock frequency
  - More than Moore:
    - System on Chip (SoC)
    - FPGA



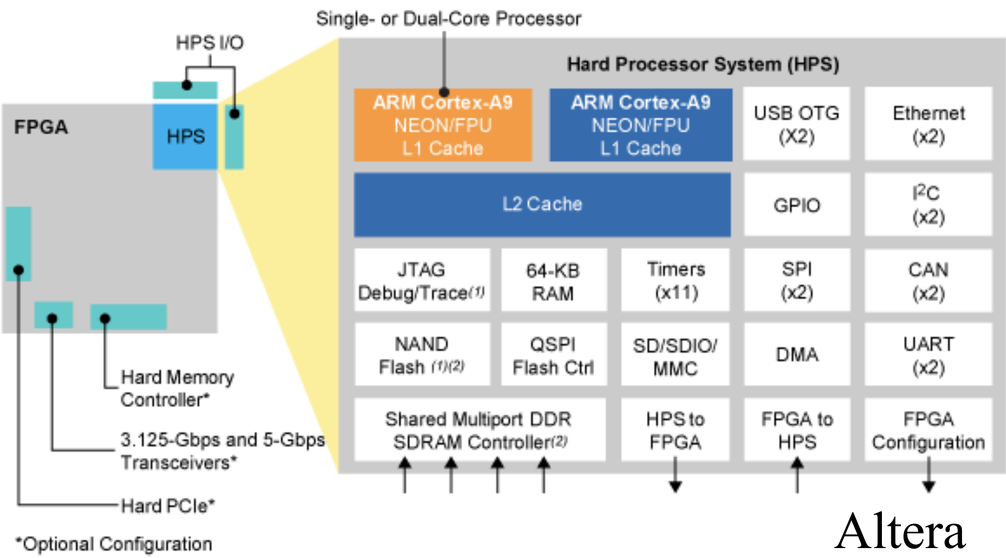


FPGA in SoC

- Systems-on-Chips
  - Faster
  - Less power
- FPGA
  - Programmable Hardware



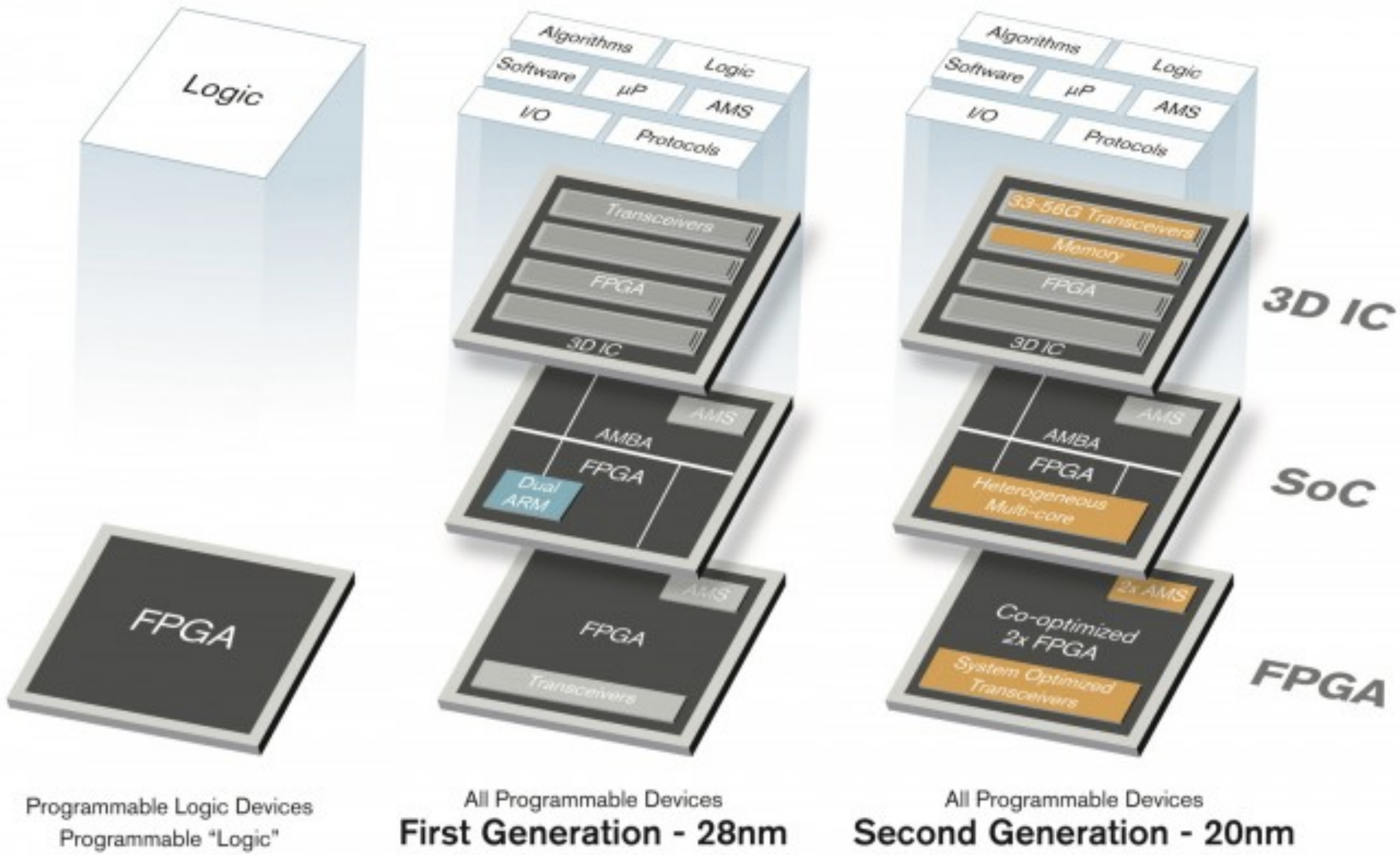
Xilinx



Altera

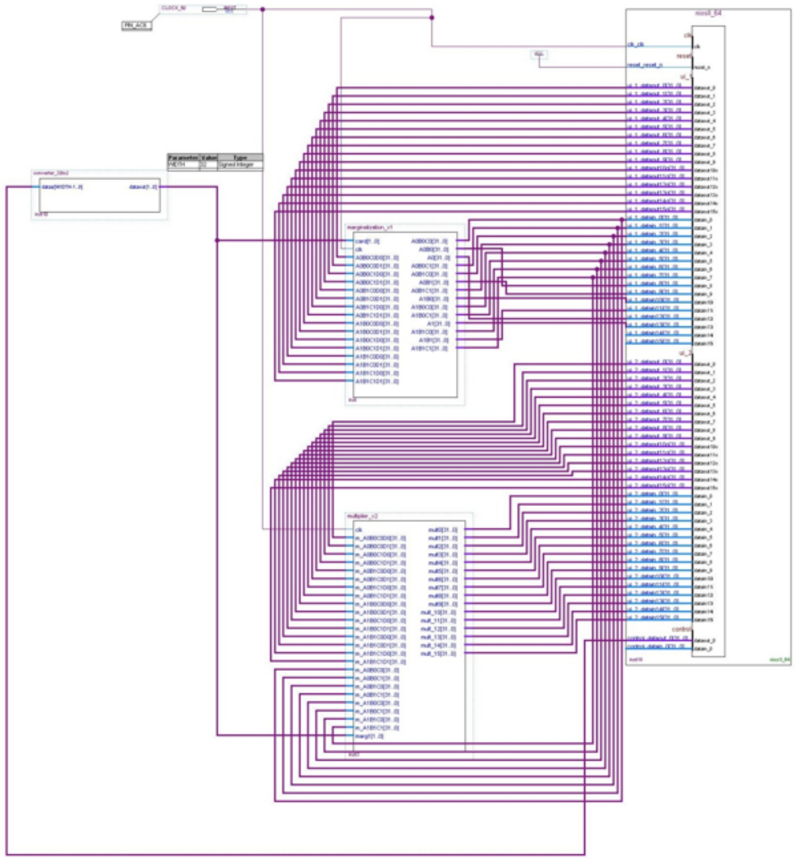
- Silicon Convergence
- Reconfigurable Computing
  - Make the device you need, when you need.

3D IC 20nm (Xilinx)

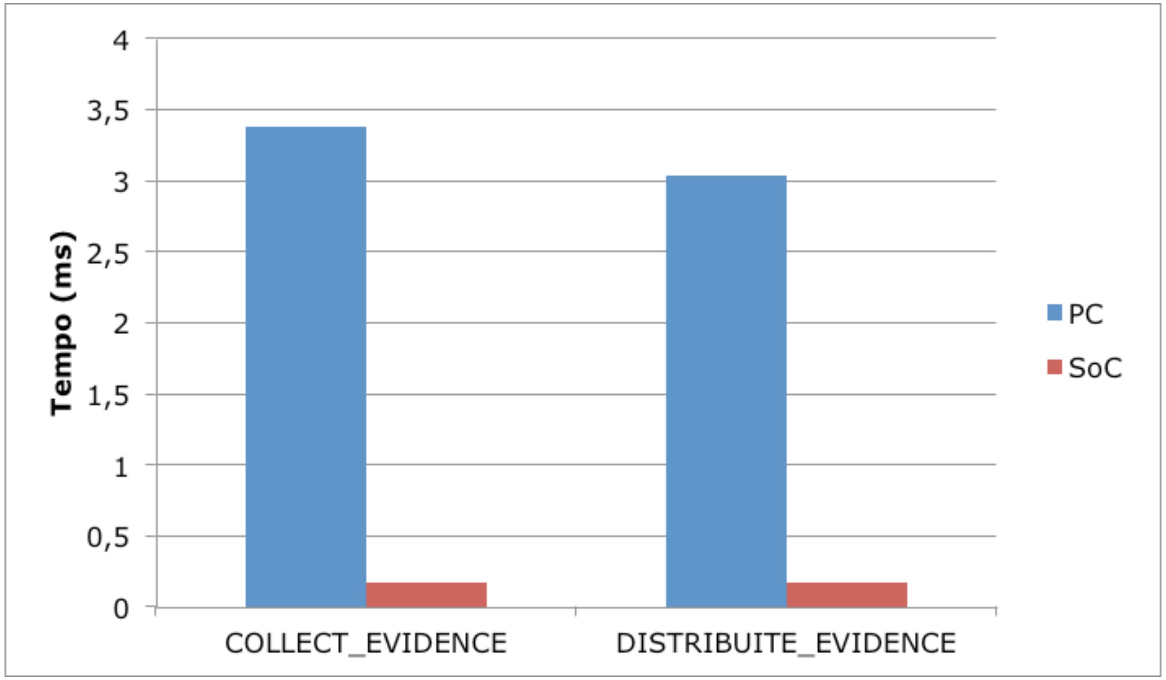


A prototype at our Lab

- A Bayesian coprocessor based on SoC in FPGA
- Based on Altera Stratix IV chipset and Nios Architecture
  - Memory on chip
  - Bayesian device (Memory Mapped)
- Two levels
  - Evidence propagation is controlled via software by Nios processor
  - Clique computation is perfumed by hardware



Experimental Results

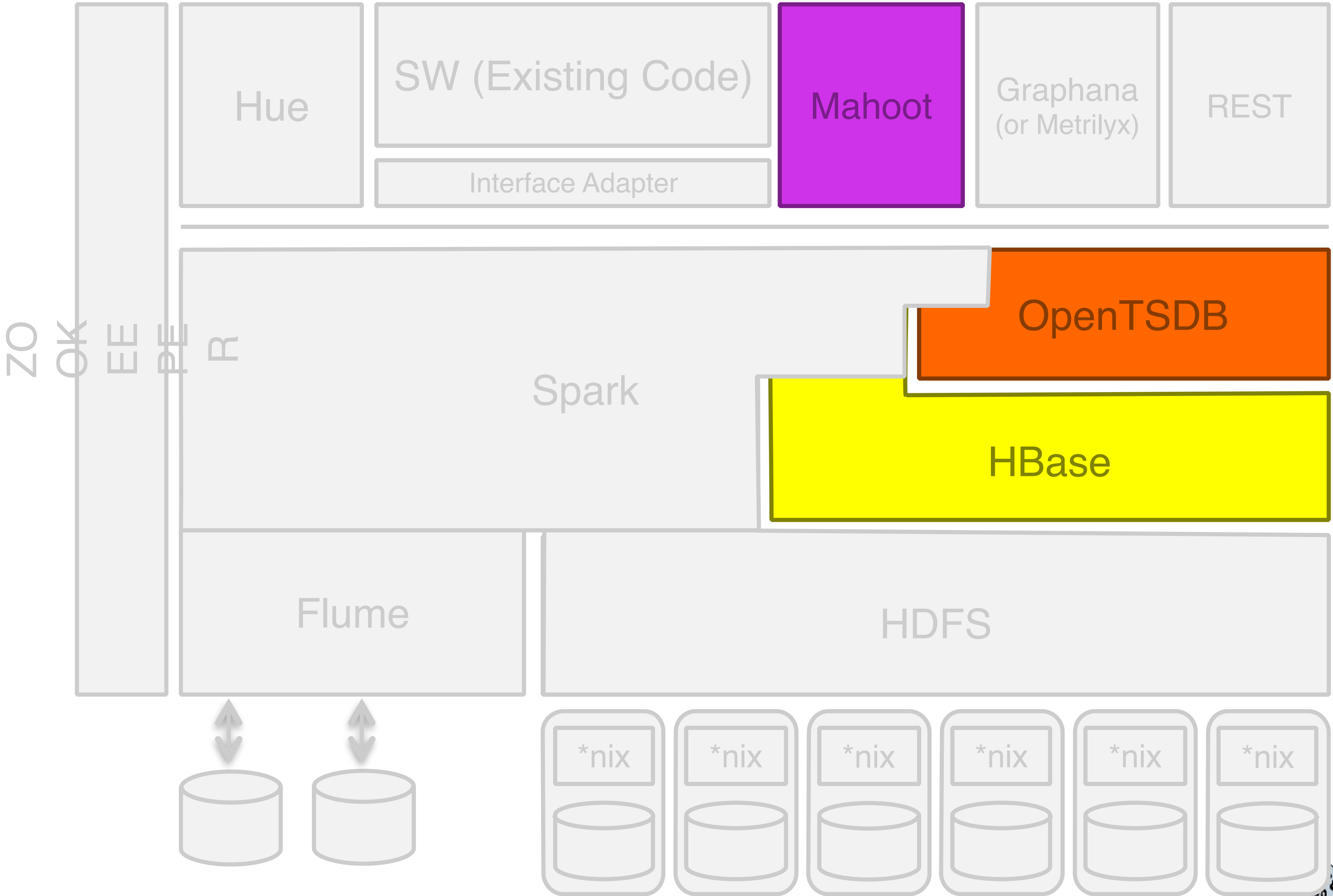


	PC	**SoC**
COLLECT_EVIDENCE	3.3765 ms	0.1731 ms
DISTRIBUITE_EVIDENCE	3.0313 ms	0.1742 ms
FULL STEP	**6.4078 ms**	**0.3473 ms**

Overview: FPGA

HW Native Support for:

- Analysis
- Time Series Retrieval
- Data Base Queries



Collaboration

- We eager to establish a collaboration on this topic

- Expected Benefits:

  - To build a cutting edge solution
  - To train KAGRA people on Big Data IT skills
  - To face a challenging project in Big Data storing & retrieval

Outline

- Big Data
- Hadoop & Map-Reduce
- HDFS vs. NFS
- Evolving towards a Big Data architecture
- FPGA co-processing
- **Conclusions**

Conclusion

- Modern technologies for Big Data handling might empower iKAGRA platform
- Evolution towards an advanced architecture can be seamless, in order to:
  - Make the most value of current software
  - Gain experience and training
  - Tailor the solution to emerging needs
  - Mitigate the risk of failure
- Research ideas: can we boost the architecture by FPGA co-processing

Enabling Big Data Architectures for the KAGRA Project

Luigi Troiano and Maria C. Vitelli
University of Sannio
{troiano,vitelli}@unisannio.it

ありがとう
ございます。

Graduate School of Science, University of Osaka — February 17th, 2015