Spark

(How it works and a use case)

André Schaaff, François-Xavier Pineau
Centre de Données astronomiques de Strasbourg

Noémie Wali, Paul Trehiou
Université de technologie de Belfort-Montbéliard

First ASTERICS-OBELICS Workshop, Rome, 12-14/12/2016

CENTRE DE DONNÉES
ASTRONOMIQUES DE STRASBOURG

H2020-Astronomy ESFRI and Research Infrastructure Cluster (Grant Agreement number: 653477).

Outline

Context

Apache Spark

Use case

The data and the « cross-match » service

Test beds

First experiment and what we have learned

On-going work and perspectives

Quick live demo

Context

A continuous exploration of new technologies, especially in the “Big Data” field

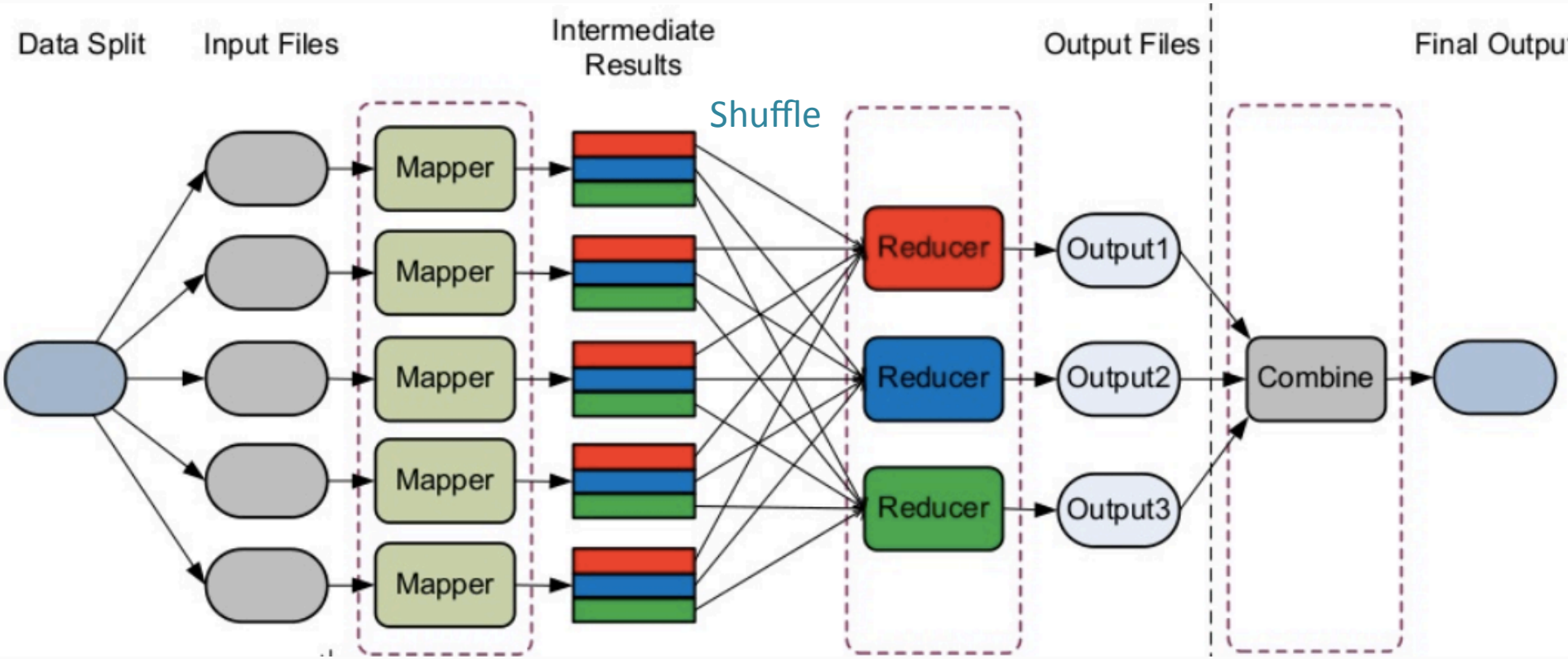
How to face the data revolution in frontend services which need to stay interactive ?

How to maintain and resize quickly the backend ?

Apache Spark

- "Apache Spark is a cluster computing platform designed to be fast and general purpose."
- It extends the MapReduce model to support more types of computations (interactive queries, stream processing, etc.) and it offers APIs for Scala, Java, Python, R,...
- N.B.: we use the release 2.0.2

MapReduce



Credit: G. Fedak, INRIA

Apache Spark (2)

- Computations in memory (as much as possible, otherwise pilling to the disks)
- Introduction of data models
  - RDD (Resilient Distributed Datasets)
    - Immutable distributed collection of elements
    - Operations: Transformations (map, filter, etc.), Actions (reduce, count, etc.)
  - Datasets to represent tabular data, queryable via SQL
- It uses mainly Hadoop Distributed File System (HDFS).

Example, Java API

Data preparation phase of our use case

```
private void parsePartitionSave(final JavaSparkContext jsc,
        final String hdfUrlIn, final int nPartitions, final String hdfUrlOut) {
    // Create the partitioner
    final HashPartitioner hp = new HashPartitioner(nPartitions);
    // Load HDFS CSV file into JavaRDD
    final JavaRDD<String> csvRDD = jsc.textFile(hdfUrlIn);
    // Parse, compute index and put result in PairRDD,
    // partitioning according to the key
    final JavaPairRDD<LongWritable, RowWritable> pairRDD =
            csvRDD.mapToPair(this.parseFunction).partitionBy(hp);
    // Save the PairRDD in HDFS
    pairRDD.saveAsHadoopFile(hdfUrlOut,
            LongWritable.class, RowWritable.class,
            SequenceFileOutputFormat.class);
}
```

Example, Java API (2)

Join phase of our use case

```
private void performXmatch(final JavaSparkContext jsc,
        final String rdd1URL, final String rdd2URL, final String txtResultURL) {
    // Load HDFS file 1 into JavaPairRDDs
    final JavaPairRDD<LongWritable, RowWritable> pairRDD1 =
            jsc.sequenceFile(rdd1URL, LongWritable.class, RowWritable.class)
            .mapToPair(READ_FUNCTION);
    // Load HDFS file 2 into JavaPairRDDs and duplicate
    JavaPairRDD<LongWritable, RowWritable> pairRDD2 =
            jsc.sequenceFile(rdd2URL, LongWritable.class, RowWritable.class)
            .mapToPair(READ_FUNCTION).flatMapToPair(this.duplicateFunction);
    // Perform the xmatch: join operation + filtering
    JavaPairRDD<LongWritable, Tuple2<RowWritable, RowWritable>> joinRes =
            pairRDD1.join(pairRDD2).filter(this.filterFunction);
    // Save the resul in HDFS
    joinRes.saveAsTextFile(txtResultURL);
}
```

How to install Spark ?

- Apache last release (2.0.2)
  - From http://spark.apache.org/downloads.html
  - Java is needed to launch Spark
  - Hadoop (2.7.3) but depending on what you want to do, http://hadoop.apache.org/releases.html
  - Do it for all of your nodes (or clone it)
- Cloudera, Hortonworks, …

How to use it ?

- With “spark-shell”, “spark-submit” or by trying the embeded examples
  - Examples:
    - ./bin/run-example SparkPi 12
    - ./bin/spark-submit ./examples/src/main/python/pi.py 10
- In a cluster mode: Standalone, Apache Mesos, Hadoop Yarn

Use case

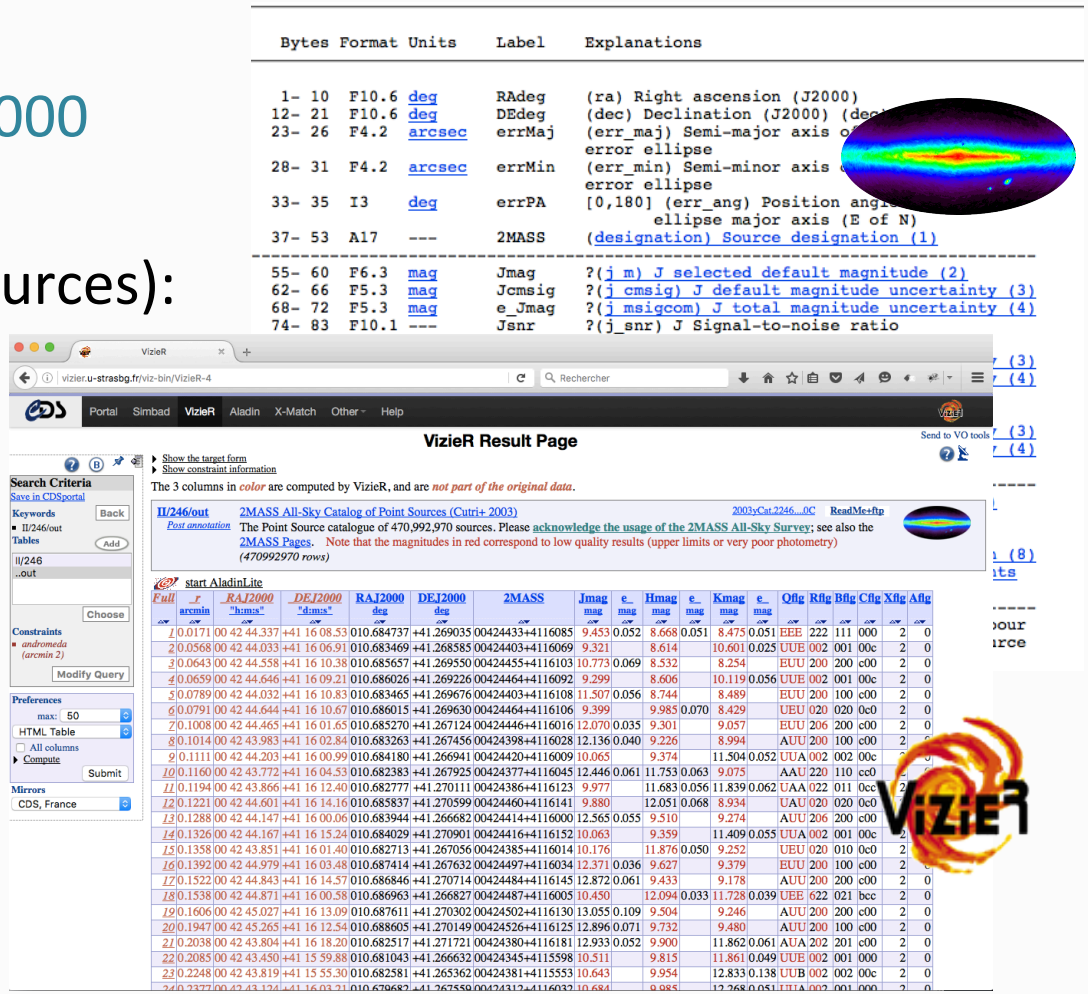
- Evaluation of Spark in the frame of a use case, the "cross-match" of source catalogues:
  - Improvement of the existing service ?

    1 server, 2x6 cores, 32GB, 12TB (15k tours) when we done the main test

    Now: 2x10 cores, 64GB with the same disks
  - Up to scale capability (data volumes, hardware, deployments (Docker ?), etc.) ?
  - Which cost (€, manpower, performances) ?
- Back thought: "bring the code to the data"

The data

- Source catalogues (>10,000 available)
- Examples (number of sources):
  - 2MASS[1], 470,992,970
  - SDSS[2] DR9, 469,053,874



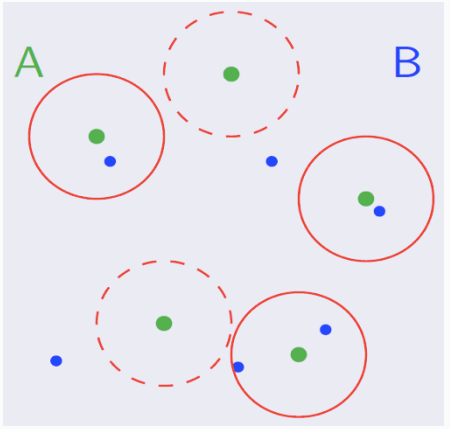
Example of a ReadMe file associated to 2MASS source catalogues available through the VizieR service

[1]2MASS, Two Micron All Sky Survey,
[2]SDSS, Sloan Digital Sky Survey

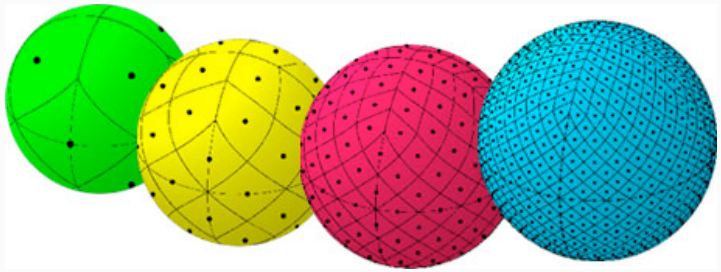
…and the CDS "cross-match" service

- The "cross-match" service does a cross correlation of sources between (very) large catalogues (current size: 10^9).

Fuzzy join between 2 tables (A and B)
of several hundred millions of data



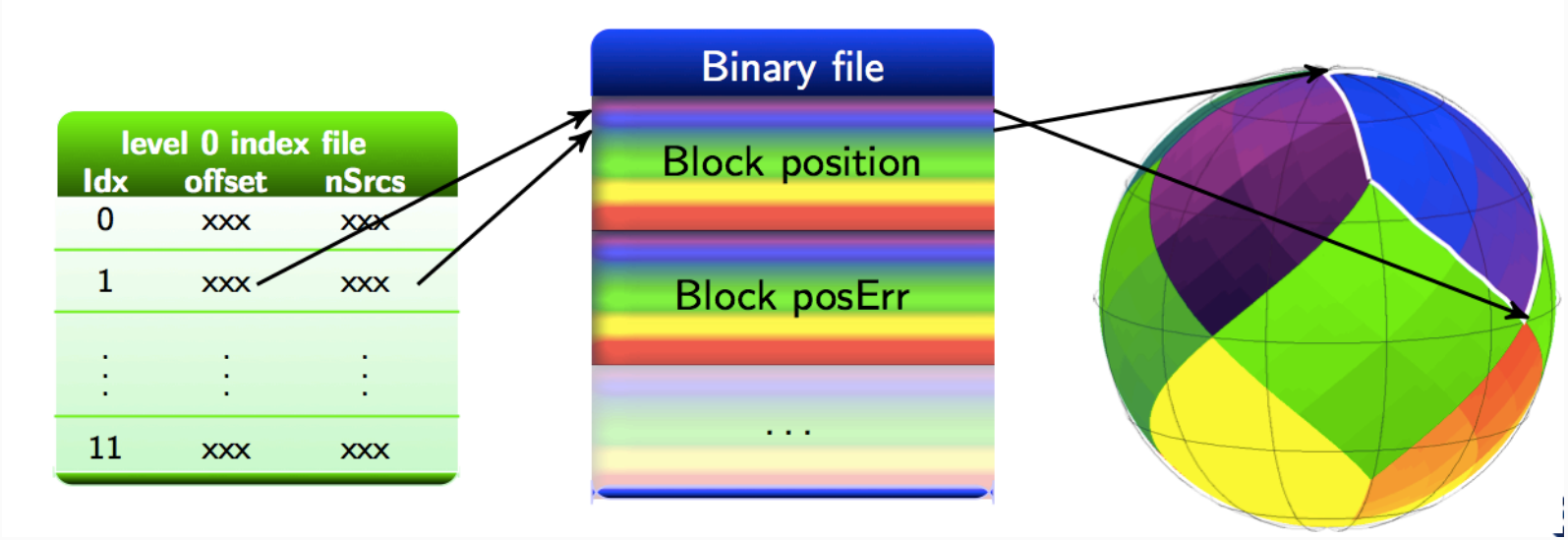
- Which area ?
  - Full sky: all the sources
  - A cone: only the sources which are at a certain angular distance from a given position
  - A HEALPix cell



Credits: http://healpix.jpl.nasa.gov/

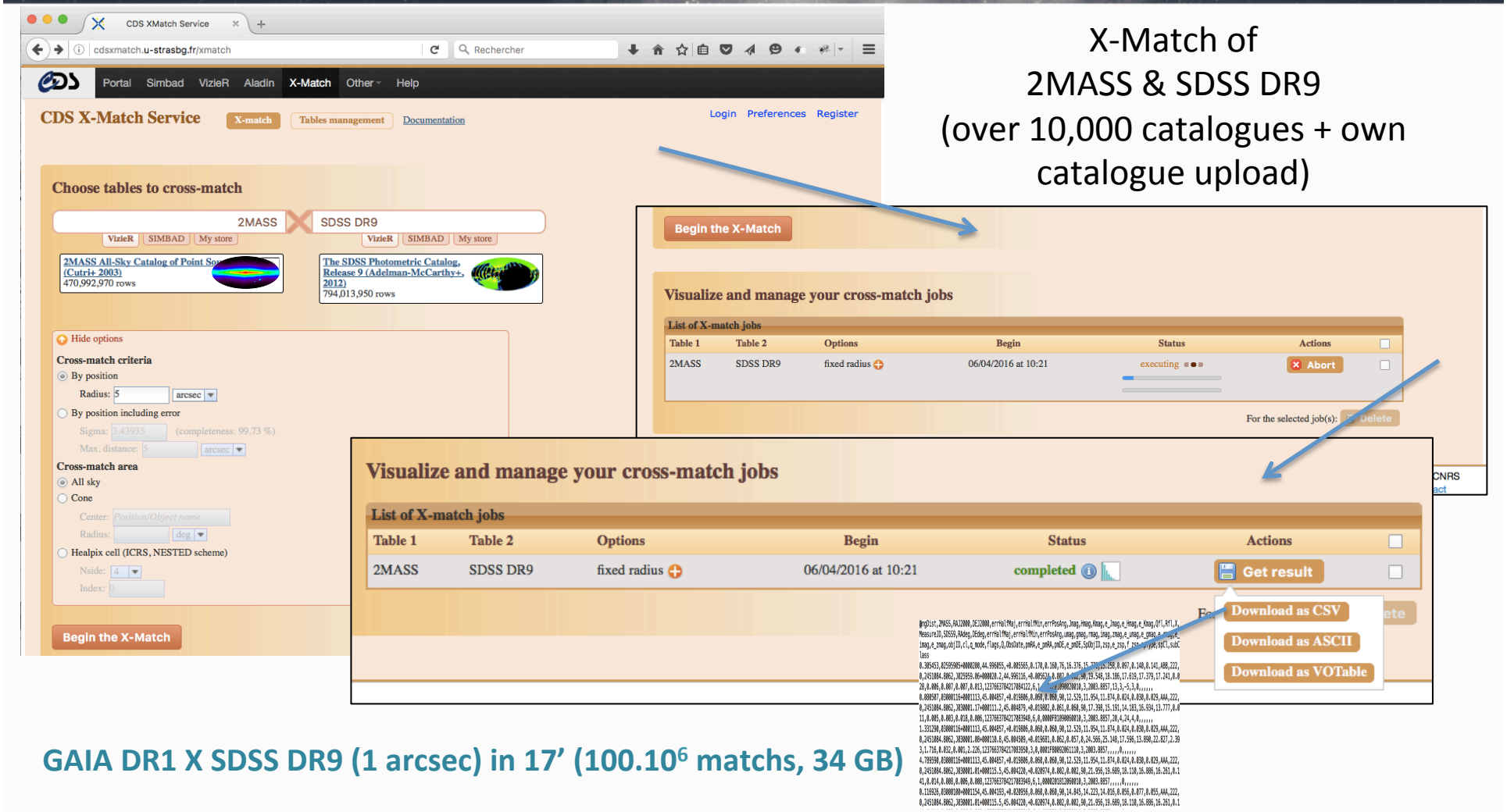
…and the CDS “cross-match” service (2)

- Data is not distributed but organised and stored on one server



The sky is cut into diamonds of the same size, pixels, each source or sky object is a numbered pixel.

…and the CDS "cross-match" service (3)



X-Match of
2MASS & SDSS DR9
(over 10,000 catalogues + own catalogue upload)

GAIA DR1 X SDSS DR9 (1 arcsec) in 17' (100.10^6 matchs, 34 GB)

Test beds: hardware & software

- Internal resources to test
  - 6 physical nodes (4 cores, 16GB, 1 TB), Ubuntu 16.04LTS
- Renting of external resources
  - 12 physical nodes, 4 cores, 32GB, Raid 2*2TB, Ubuntu 14.04LTS (8000€ / year)
  - Configuration was defined “ad hoc” and low cost
- Software side:
  - Apache distributions of Spark (1.5.0 to 2.0.2) and Hadoop (2.6 to 2.7.3)
  - Java, Scala

First experiment (SDSS DR7 X 2MASS)

Data preparation phase

Input files to HDFS and loading into 2 RDDs (information about an object in the Sky)

Each RDD is transformed in a pairRDD (key = "source pixel number", value = "all the information whose the source (ra, dec) ")

PairRDD distribution over the nodes (hashpartionning => grouping elements having the same key (same pixel number) in the same partition)

Elements with the same key are on the same node, distribution is essential to the join phase

PairRDDs are stored into HDFS as binary files preserving the structure (Key, Value)

Example, Java API (bis)

Data preparation phase of our use case

```
private void parsePartitionSave(final JavaSparkContext jsc,
        final String hdfUrlIn, final int nPartitions, final String hdfUrlOut) {
    // Create the partitioner
    final HashPartitioner hp = new HashPartitioner(nPartitions);
    // Load HDFS CSV file into JavaRDD
    final JavaRDD<String> csvRDD = jsc.textFile(hdfUrlIn);
    // Parse, compute index and put result in PairRDD,
    // partitioning according to the key
    final JavaPairRDD<LongWritable, RowWritable> pairRDD =
            csvRDD.mapToPair(this.parseFunction).partitionBy(hp);
    // Save the PairRDD in HDFS
    pairRDD.saveAsHadoopFile(hdfUrlOut,
            LongWritable.class, RowWritable.class,
            SequenceFileOutputFormat.class);
}
```

First experiment (2)

Join phase

Loading in two PairRDDs + duplication* of some sources in the neighbour pixels in one of it

PairRDDs joined following the Key into a new PairRDD where the elements are (Key, Value1, Value2) triples

Join done following the Key (cell number), 2 near sources can be in the different cells and are not joined (=> duplication* of sources in the neighbour cells to avoid the side effects)
*a circle with a fixed radius is drawn around the source, If neighbour pixels are partially in this circle, the source is then duplicated in the neighbour cells

The joined elements are then filtered (distance between the 2 sources < a given threshold)

Final result stored in HDFS (in a text format for a later visualization and use)

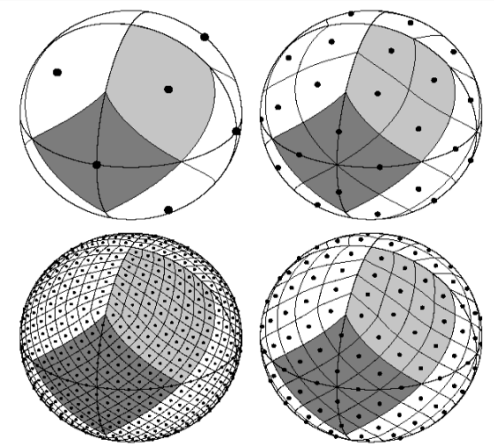
Example, Java API (2) (bis)

Join phase of our use case

```
private void performXmatch(final JavaSparkContext jsc,
        final String rdd1URL, final String rdd2URL, final String txtResultURL) {
    // Load HDFS file 1 into JavaPairRDDs
    final JavaPairRDD<LongWritable, RowWritable> pairRDD1 =
            jsc.sequenceFile(rdd1URL, LongWritable.class, RowWritable.class)
            .mapToPair(READ_FUNCTION);
    // Load HDFS file 2 into JavaPairRDDs and duplicate
    JavaPairRDD<LongWritable, RowWritable> pairRDD2 =
            jsc.sequenceFile(rdd2URL, LongWritable.class, RowWritable.class)
            .mapToPair(READ_FUNCTION).flatMapToPair(this.duplicateFunction);
    // Perform the xmatch: join operation + filtering
    JavaPairRDD<LongWritable, Tuple2<RowWritable, RowWritable>> joinRes =
            pairRDD1.join(pairRDD2).filter(this.filterFunction);
    // Save the resul in HDFS
    joinRes.saveAsTextFile(txtResultURL);
}
```

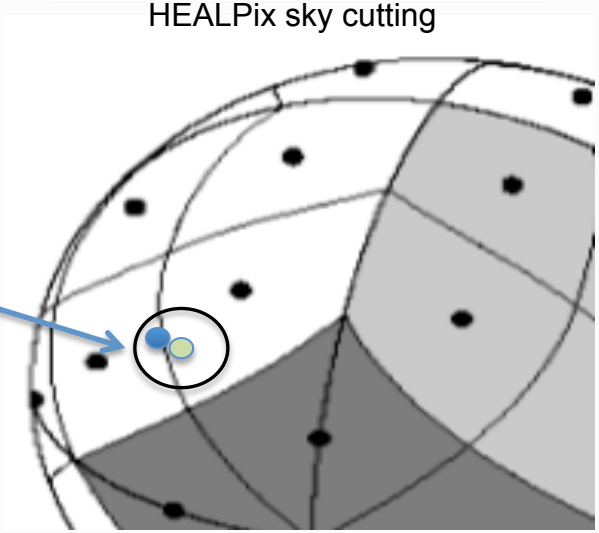
Illustration

- A X-Match implementation in MapReduce, couples (Key = pixel number, Value)



HEALPix sky cutting

- Side effects
  - Fuzzy join
  - Source duplication in the neighbour cells if needed



Credits: HEALPix – arXiv:astro-ph/0409513

First experiment result (12 nodes)

- Input data (SDSS DR7 (primary sources) and 2MASS): 54GB and 58GB file size; 357 175 411 and 470 992 970 elements
- Output data: 49 208 820 elements

X-Match service reference time was: 10 minutes

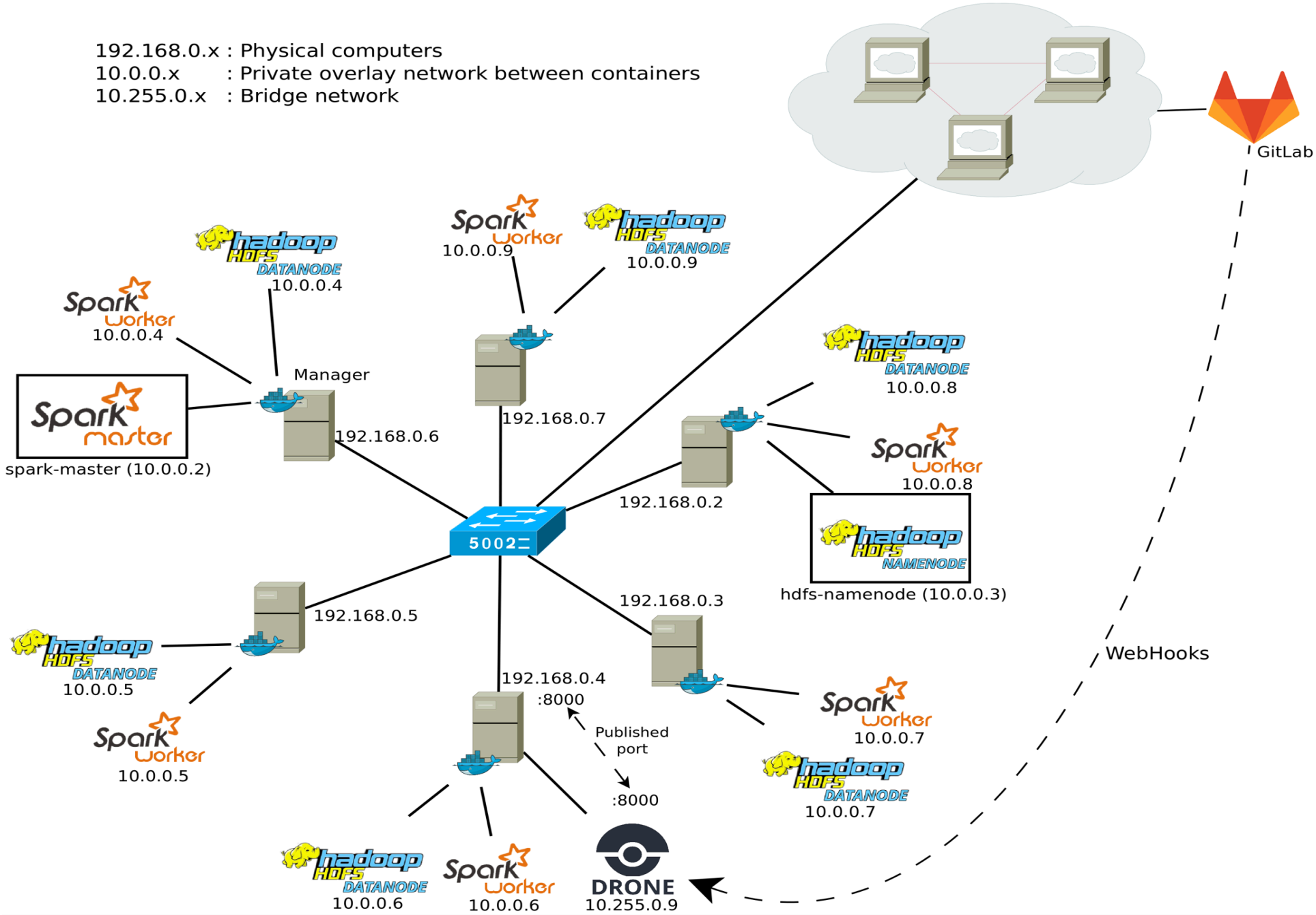
Cross-Match (source duplication done in phase 2 with all the data as output)					
HDFS block size= 128MB for the input files ; sdss7.csv and t 2mass.csv replicated 2 times					
HashPartitioner	**60 partitions**				
HDFS output files size	**32MB**				
Number of nodes Spark/HDFS	5	7	9	10	11
Phase 1: prepare	**23,0**	**16,0**	**14,0**	**14,0**	**13,0**
mapToPair (sdss7.csv)	5,1	4,9	4,9	4,8	4,7
saveAsHadoopFile (sdss7.bin)	5,7	2,7	2,0	2,3	1,5
mapToPair (2mass.csv)	5,7	5,2	5,2	5,1	5,0
saveAsHadoopFile (2mass.bin)	6,5	3,6	1,9	1,6	[illegible]
Phase 2: join	**31,0**	**21,0**	**13,0**	**11,0**	**9,9**
mapToPair (sdss7.bin)	7,2	4,7	3,5	3,0	[illegible]
flatMapToPair (2mass.bin)	11,8	8,3	5,5	4,9	4,3
saveAsTextFile (crossMatch_D.txt)	12,0	7,6	3,4	2,4	2,3
TOTAL	**54,0**	**37,0**	**27,0**	**25,0**	**22,9**

What we have learned

- Time was similar to the X-Match service from 11 nodes but
  - Keys common to 2 RDDs are not necessarily on the same node
    - It implies a transfer overhead between the nodes during the join => impact on the performances
    - We had clearly a bottleneck in the join phase ("shuffle")
    - "block affinity groups" is an on-going work at Apache.
  - We spent time on the "data co-location"
  - We found a solution to do it "manually" via scripts.

On-going work

- Introduction of Docker (components) and Drone (continuous integration) to "automate" the process and to focus mainly on the development side. It is becoming easy to migrate to external resources when needed.
- Use of Scala which is native in Spark (a part of the Java API is "experimental").
- Sharing of our experiments (in/outside the community).


192.168.0.x : Physical computers
10.0.0.x : Private overlay network between containers
10.255.0.x : Bridge network
GitLab
WebHooks
Spark Worker 10.0.0.9
hadoop HDFS DATANODE 10.0.0.9
hadoop HDFS DATANODE 10.0.0.4
Spark Worker 10.0.0.4
Manager
Spark master
spark-master (10.0.0.2)
192.168.0.6
192.168.0.7
hadoop HDFS DATANODE 10.0.0.8
Spark Worker 10.0.0.8
192.168.0.2
hadoop HDFS NAMENODE
hdfs-namenode (10.0.0.3)
5002
192.168.0.5
192.168.0.3
hadoop HDFS DATANODE 10.0.0.5
Spark Worker 10.0.0.5
192.168.0.4
:8000
Published port
:8000
Spark Worker 10.0.0.7
hadoop HDFS DATANODE 10.0.0.7
hadoop HDFS DATANODE 10.0.0.6
Spark Worker 10.0.0.6
DRONE 10.255.0.9

Perspectives

- What we expect:
  - Significant improving of the performances, with a reasonable hardware cost.
  - Evaluation (and comparison) of other technologies like Spark and Docker, minimize as much as possible the dependency to a specific one.
- Implement a prototype allowing a user "to move his code to the data".

Links

- Apache Spark, http://spark.apache.org/
- Apache Hadoop, http://hadoop.apache.org/
- Spark : Cluster Computing with Working Sets, Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica, University of California, Berkeley, http://static.usenix.org/legacy/events/hotcloud10/tech/full_papers/Zaharia.pdf
- Optimizing Shuffle Performance in Spark, Aaron Davidson, Andrew Or, UC Berkeley, http://www.cs.berkeley.edu/~kubitron/courses/cs262a-F13/projects/reports/project16_report.pdf
- Resilient Distributed Datasets : A Fault-Tolerant Abstraction for In-Memory Cluster Computing, Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica, University of California, Berkeley, https://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf
- JavaSpark Api, http://spark.apache.org/docs/latest/api/java/
- HEALPix, http://healpix.jpl.nasa.gov/

Quick live demo

- Just a demo to show that Spark is really easy to install and to use quickly on a simple laptop:
  - What we need
  - A simple use
  - A cluster mode
  - Etc.

Acknowledgement

- H2020-Astronomy ESFRI and Research Infrastructure Cluster (Grant Agreement number: 653477).