

Spark

(How it works and a use case)

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

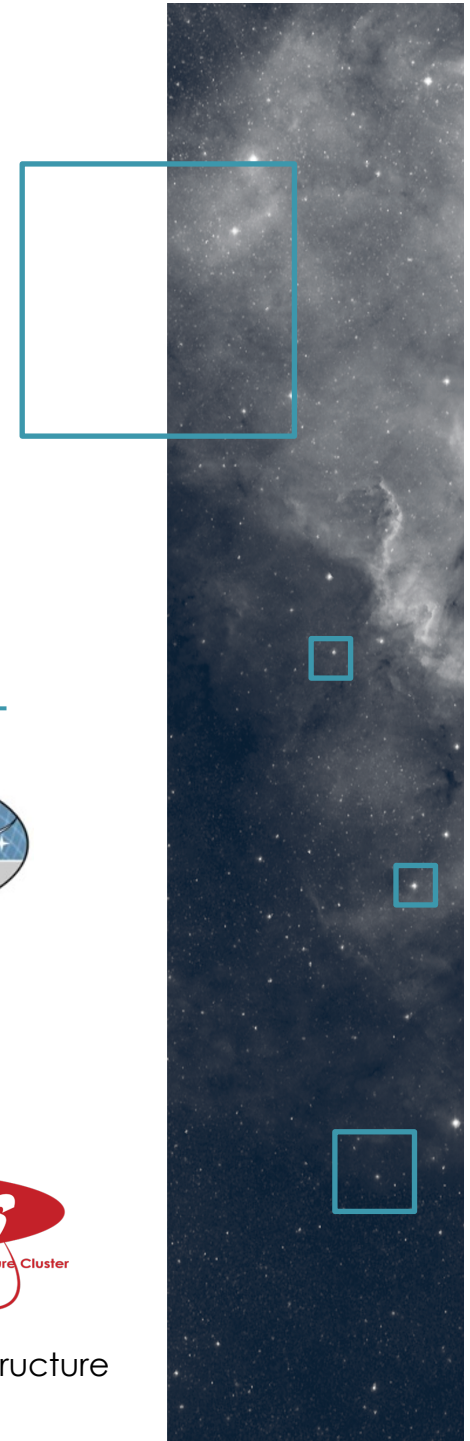


Noémie Wali, Paul Trehou
Université de technologie de Belfort-Montbéliard

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



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

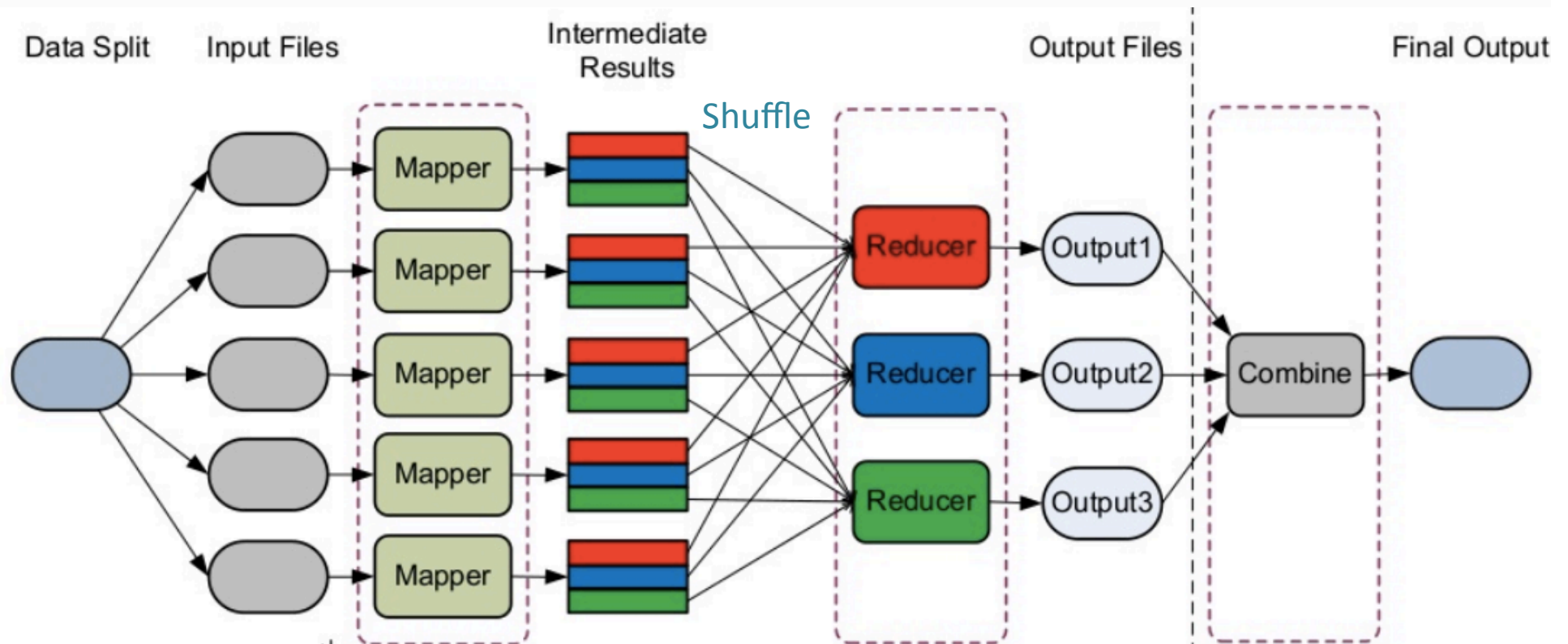
How to face the data revolution in frontend
services which need to stay interactive ?

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 result 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**”

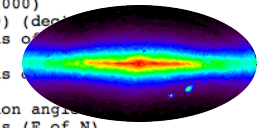
□ The data

- Source catalogues (>10,000 available)
- Examples (number of sources):
 - 2MASS¹, 470,992,970
 - SDSS² DR9, 469,053,874

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

¹2MASS, Two Micron All Sky Survey,
²SDSS, Sloan Digital Sky Survey

Bytes	Format	Units	Label	Explanations
1- 10	F10.6	deg	RAdeg	(ra) Right ascension (J2000)
12- 21	F10.6	deg	DEdeg	(dec) Declination (J2000) (deg)
23- 26	F4.2	arcsec	errMaj	(err_maj) Semi-major axis of error ellipse
28- 31	F4.2	arcsec	errMin	(err_min) Semi-minor axis of error ellipse
33- 35	I3	deg	errPA	(err_ang) Position angle of ellipse major axis (E of N)
37- 53	A17	---	2MASS	(designation) Source designation (1)
55- 60	F6.3	mag	Jmag	?(j m) J selected default magnitude (2)
62- 66	F5.3	mag	Jcmsig	?(j cmsig) J default magnitude uncertainty (3)
68- 72	F5.3	mag	e_Jmag	?(j msigcom) J total magnitude uncertainty (4)
74- 83	F10.1	---	Jsnr	?(j_snr) J Signal-to-noise ratio



VizieR Result Page

The 3 columns in color are computed by VizieR, and are *not part of the original data*.

II/246/out 2MASS All-Sky Catalog of Point Sources (Cutri+ 2003)

The Point Source catalogue of 470,992,970 sources. Please acknowledge the usage of the 2MASS All-Sky Survey; see also the 2MASS Pages. Note that the magnitudes in red correspond to low quality results (upper limits or very poor photometry) (470992970 rows)

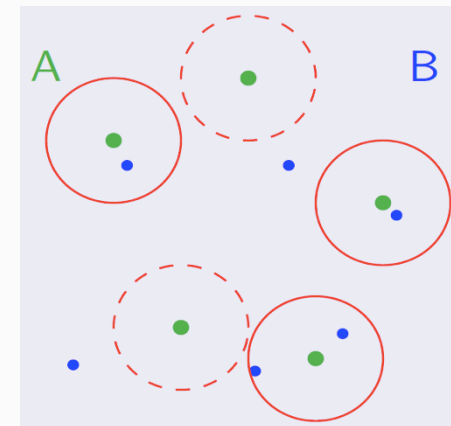
Full	r	RAJ2000	DEJ2000	RAJ2000	DEJ2000	2MASS	Jmag	e	Hmag	e	Kmag	e	Oflg	Rflg	Bflg	Cflg	Xflg	Aflg
1	0.0171	00 42 44.337	+41 16 08.53	010.684737	+41.269035	00424433+4116085	9.453	0.052	8.668	0.051	8.475	0.051	EEB	222	111	000	2	0
2	0.0568	00 42 44.033	+41 16 06.91	010.683469	+41.268585	00424403+4116069	9.321	8.614	10.601	0.025	UEE	002	001	00c	2	0	0	0
3	0.0643	00 42 44.558	+41 16 10.38	010.685657	+41.269550	00424455+4116103	10.773	0.069	8.532	8.254	AAU	200	200	c0c	2	0	0	0
4	0.0659	00 42 44.646	+41 16 09.21	010.686026	+41.269226	00424464+4116092	9.299	8.606	10.119	0.056	UEU	002	001	00c	2	0	0	0
5	0.0789	00 42 44.032	+41 16 10.83	010.683465	+41.269630	00424403+4116108	11.507	0.056	8.744	8.489	EEU	200	100	c0c	2	0	0	0
6	0.0791	00 42 44.644	+41 16 10.67	010.686015	+41.269630	00424464+4116106	9.399	9.985	0.070	8.429	UEU	020	020	0c0	2	0	0	0
7	0.1008	00 42 44.465	+41 16 01.65	010.685270	+41.267124	00424464+4116016	12.070	0.035	9.301	9.057	EEU	200	200	c0c	2	0	0	0
8	0.1014	00 42 43.983	+41 16 02.84	010.683263	+41.267456	00424398+4116028	12.136	0.040	9.226	8.994	AAU	200	100	c0c	2	0	0	0
9	0.1111	00 42 44.203	+41 16 00.99	010.684180	+41.266941	00424420+4116009	10.065	9.374	11.504	0.052	UAU	002	002	00c	2	0	0	0
10	0.1160	00 42 43.772	+41 16 04.53	010.682383	+41.267925	00424377+4116045	12.446	0.061	11.753	0.063	9.075	AAU	220	110	cc0	2	0	0
11	0.1194	00 42 43.866	+41 16 12.40	010.682777	+41.270111	00424386+4116123	9.977	11.683	0.056	11.839	0.062	UAA	022	011	cc0	2	0	0
12	0.1221	00 42 44.601	+41 16 14.16	010.685837	+41.270599	00424460+4116141	9.880	12.051	0.068	8.934	UAU	020	020	0c0	2	0	0	0
13	0.1288	00 42 44.147	+41 16 00.06	010.683944	+41.266682	00424414+4116000	12.565	0.055	9.510	9.274	AAU	200	200	c0c	2	0	0	0
14	0.1326	00 42 44.167	+41 16 15.24	010.684029	+41.270901	00424416+4116152	10.063	9.359	11.409	0.055	UUA	002	001	00c	2	0	0	0
15	0.1358	00 42 43.851	+41 16 01.40	010.682713	+41.267056	00424385+4116014	10.176	11.876	0.050	9.252	UEU	020	010	cc0	2	0	0	0
16	0.1392	00 42 44.979	+41 16 03.48	010.687414	+41.267632	00424497+4116034	12.371	0.036	9.627	9.379	EEU	200	100	c0c	2	0	0	0
17	0.1522	00 42 44.843	+41 16 14.57	010.686846	+41.270714	00424484+4116145	12.872	0.061	9.433	9.178	AAU	200	200	c0c	2	0	0	0
18	0.1538	00 42 44.871	+41 16 00.58	010.686963	+41.266827	00424487+4116005	10.450	12.094	0.033	11.728	0.039	UEE	622	021	bcc	2	0	0
19	0.1606	00 42 45.027	+41 16 13.09	010.687611	+41.270302	00424502+4116130	13.055	0.109	9.504	9.246	AAU	200	200	c0c	2	0	0	0
20	0.1947	00 42 45.265	+41 16 12.54	010.688605	+41.270149	00424526+4116125	12.896	0.071	9.732	9.480	AAU	200	100	c0c	2	0	0	0
21	0.2038	00 42 43.804	+41 16 18.20	010.682517	+41.271721	00424380+4116181	12.933	0.052	9.900	11.862	0.061	UAU	022	201	cc0	2	0	0
22	0.2085	00 42 43.450	+41 15 59.88	010.681043	+41.266632	00424345+4115598	10.511	9.815	11.861	0.049	UEE	002	001	00c	2	0	0	0
23	0.2248	00 42 43.819	+41 15 55.30	010.682581	+41.265362	00424381+4115553	10.643	9.954	12.833	0.138	UUB	002	002	00c	2	0	0	0
24	0.2372	00 42 43.124	+41 16 03.31	010.679682	+41.263550	00424312+4116032	10.684	9.085	12.268	0.051	UEE	002	001	00c	2	0	0	0



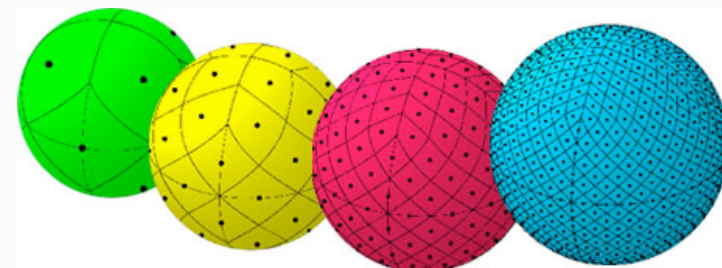
□ ...and the CDS “cross-match” service

- 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



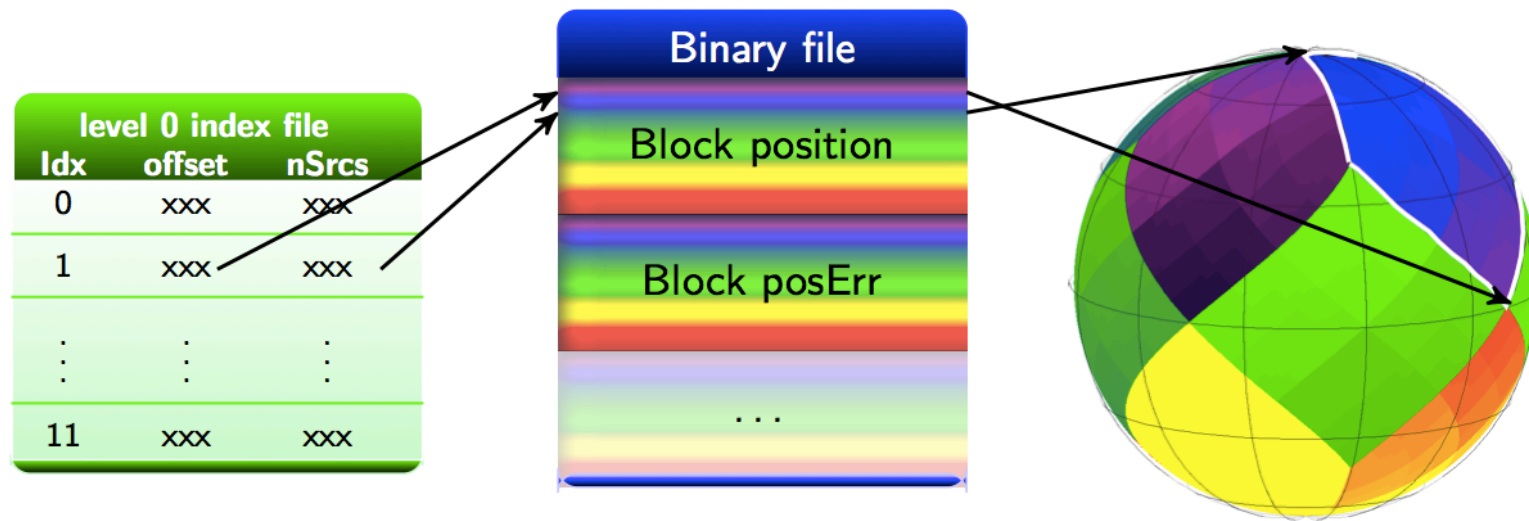
- Which area ?
 - Full sky: all the sources
 - A cone: only the sources which are at a certain angular distance from a given position
 - A HEALPix cell



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)

Choose tables to cross-match

2MASS All-Sky Catalog of Point Sources (Cutri+ 2003)
470,992,970 rows

The SDSS Photometric Catalog, Release 9 (Adelman-McCarthy+, 2012)
794,013,950 rows

Cross-match criteria

By position
Radius: 5 arcsec

By position including error
Sigma: 3.43935 (completeness: 99.73 %)
Max. distance: 5 arcsec

Cross-match area

All sky
Center: Position/Object name
Radius: deg
Healpix cell (ICRS, NESTED scheme)
Nside: 4
Index: 0

Begin the X-Match

Visualize and manage your cross-match jobs

Table 1	Table 2	Options	Begin	Status	Actions
2MASS	SDSS DR9	fixed radius	06/04/2016 at 10:21	executing	Abort

For the selected job(s): Delete

Visualize and manage your cross-match jobs

Table 1	Table 2	Options	Begin	Status	Actions
2MASS	SDSS DR9	fixed radius	06/04/2016 at 10:21	completed	Get result

Download as CSV
Download as ASCII
Download as VOTable

GAIA DR1 X SDSS DR9 (1 arcsec) in 17' (100.10⁶ matches, 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 (**hashpartitioning** => **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)

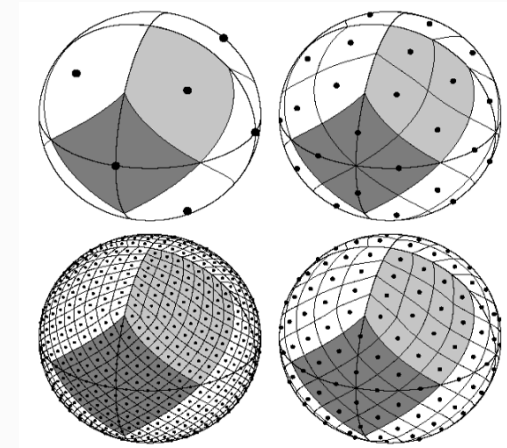
□ Example, Java API (2) (bis)

Join phase of our use case

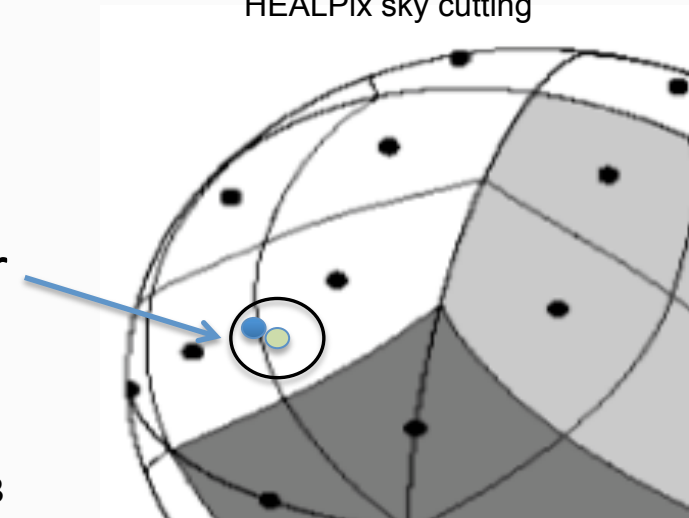
```
private void performXmatch(final JavaSparkContext jsc,
    final String rdd1URL, final String rdd2URL, final String txtResultURL) {
    // Load HDFS file 1 into JavaPairRDDs
    final JavaPairRDD<LongWritable, RowWritable> pairRDD1 =
        jsc.sequenceFile(rdd1URL, LongWritable.class, RowWritable.class)
            .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 result in HDFS
    joinRes.saveAsTextFile(txtResultURL);
}
```

□ Illustration

- A X-Match implementation in MapReduce, couples (Key = pixel number, Value)
- Side effects
 - Fuzzy join
 - Source duplication in the neighbour cells if needed



HEALPix sky cutting



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	1,4
Phase 2: join	31,0	21,0	13,0	11,0	9,9
mapToPair (sdss7.bin)	7,2	4,7	3,5	3,0	2,5
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

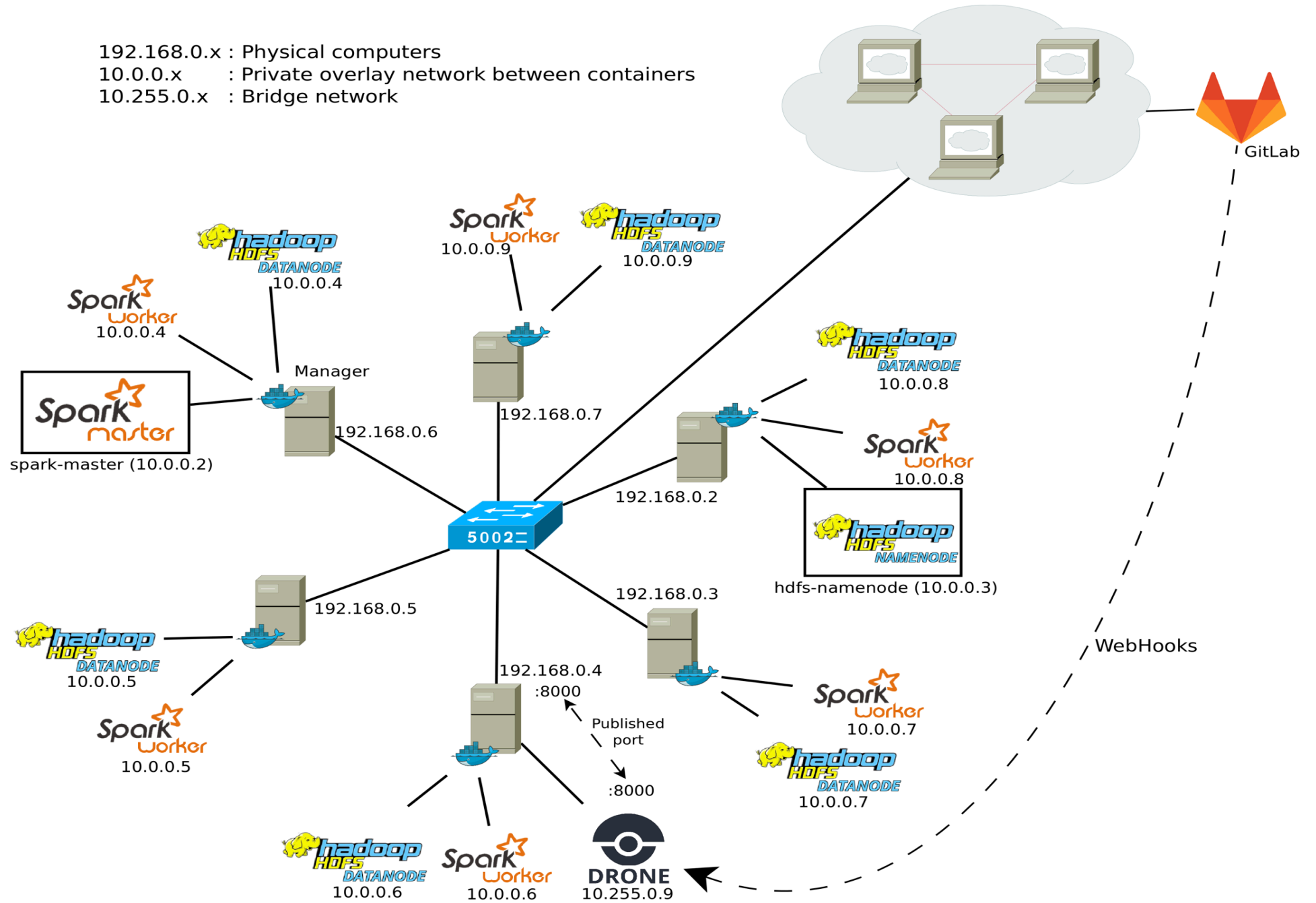
□ What we have learned

- 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



□ 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).