Comparison of MongoDB and Cassandra Databases for Spectrum Monitoring As-a-Service

Giuseppe Baruffa, Mauro Femminella, *Member, IEEE*, Matteo Pergolesi, *Student Member, IEEE*, and Gianluca Reali, *Member, IEEE*

***Abstract*—Due to the growing number of devices accessing the Internet through wireless networks, the radio spectrum has become a highly contended resource. The availability of low cost radio spectrum monitoring sensors enables a geographically distributed, real-time observation of the spectrum to spot inefficiencies and to develop new strategies for its utilization. The potentially large number of sensors to be deployed and the intrinsic nature of data make this task a Big Data problem. In this work we design, implement, and validate a hardware and software architecture for wideband radio spectrum monitoring inspired to the Lambda architecture. This system offers Spectrum Sensing as a Service to let end users easily access and process radio spectrum data. To minimize the latency of services offered by the platform, we fine tune the data processing chain. From the analysis of sensor data characteristics, we design the data models for MongoDB and Cassandra, two popular NoSQL databases. A MapReduce job for spectrum visualization has been developed to show the potential of our approach and to identify the challenges in processing spectrum sensor data. We experimentally evaluate and compare the performance of the two databases in terms of application processing time for different types of queries applied on data streams with heterogeneous generation rate. Our experiments show that Cassandra outperforms MongoDB in most cases, with some exceptions depending on data stream rate.**

***Index Terms*—Distributed spectrum sensing, big data, Lambda architecture, NoSQL, data model, MapReduce, data visualization.**

I. Introduction

RADIO spectrum is a valuable and strictly regulated resource for wireless communications. With the proliferation of wireless services, the demand for the radio spectrum is constantly increasing, leading to contention of spectrum resources. On the other hand, since the utilization of the radio spectrum at any time and location could be low, spectrum re-usage is a valid approach. Cognitive radio [1], [2] is a promising solution to improve the spectrum usage in the next generation wireless networks, by taking advantage of transmission opportunities in multiple dimensions [3].

Recently, standardization activities have addressed re-use of the underutilized spectrum by digital TV signals, the so called TV white space (TVWS), in order to extend the coverage of WiFi services with the IEEE 802.11af standard [4]. Moreover, the standard hinges upon a coordinated spectrum access by using a spatial geolocation database [5].

A fundamental task in the development of cognitive radio is spectrum sensing, which allows identifying specific usage patterns in different dimensions (time, frequency, space, angle), where free spectrum portions can be found. Since continuous spectrum sensing is a highly demanding task for a wireless device, we propose the introduction of a monitoring platform that makes spectrum sensing available as a service. The users, such as researchers in cognitive radio and spectrum sensing, wireless network operators, or spectrum regulation agencies, can easily access spectrum data, run custom data analysis, and configure custom sensing campaigns without dealing with the radio devices configuration complexity and deployment. Indeed, the service includes a number of spectrum sensors, implemented through software defined radio (SDR) [6], geographically distributed, which carry out the sensing operation, even opportunistically and on-demand. They send sensing records to a storage and computing platform, having the task of both distributing collected data to requesting terminals and allowing further analyses by executing custom processing jobs. Since different users may have different service requirements in terms of accuracy, service time or time/frequency ranges, we present three different strategies to process sensor data. They allow minimizing spectrum data processing time as a function of the acquisition time windows and frequency ranges.

In more detail, our contribution consists of the design and performance assessment of a cloud platform offering Sensing as a Service (S^2aaS) [7], [8] and, more specifically, Spectrum Sensing as a Service (S^3aaS) [9]. We have designed and implemented a Lambda architecture [10] based on Apache Flink,[1] which is a data processing engine able to run both batch and streaming-oriented jobs [11]. Data ingestion from spectrum sensors is managed through Apache Kafka,[2] a streaming

Manuscript received January 21, 2019; revised July 6, 2019; accepted September 16, 2019. Date of publication September 19, 2019; date of current version March 11, 2020. This work has been performed in the framework of the European projects 5G-EVE and 5G CARMEN. These projects have received funding from the European Union's Horizon 2020 research and innovation programme under grant agreements No. 815974 and 825012. The views expressed are those of the authors and do not necessarily represent the projects. The Commission is not responsible for any use that may be made of the information it contains. The associate editor coordinating the review of this article and approving it for publication was Y. Wu. *(Corresponding author: Mauro Femminella.)*

The authors are with the Department of Engineering, University of Perugia, 06125 Perugia, Italy, and also with Consortium CNIT, 06125 Perugia, Italy (e-mail: mauro.femminella@unipg.it).

Digital Object Identifier 10.1109/TNSM.2019.2942475

[1] https://flink.apache.org/

[2] https://kafka.apache.org/

platform used to decouple data production from consumption [12]. Finally, for data storage, we have selected and compared MongoDB[3] and Cassandra,[4] two popular NoSQL databases [13] that fit our requirements of scalability, support of time-series, and support of binary objects. The relevant details are reported in Section II-C. Our platform can integrate heterogeneous sensors including even opportunistic sensors. To demonstrate the architecture feasibility and operation, we implemented a full-fledged testbed on a private cloud in our lab, used to experimentally evaluate the system performance.

In our previous works [14], [15] we presented a sketch architecture, a simplified prototype implementation and a comparison with a legacy approach. With respect to our previous works, the main novelties of this paper are (i) the design of data models for the two NoSQL databases, specialized for the spectrum data, (ii) the design and implementation of a Lambda architecture, with the possibility of executing streaming jobs, (iii) an in-depth performance evaluation by means of a MapReduce [16] job used to visualize spectrum data. This evaluation allowed us to assess the impact of the database and relevant data model on service time for the three different data processing strategies proposed. The comparison has been done for different types of queries applied on spectrum data streams with heterogeneous generation rate, produced by different spectrum sensors.

The rest of this paper is organized as follows. In Section II, we present the key concepts and related work on the subject. In Section III, we introduce the system architecture, focusing on sensor operation, data models for storage, and algorithms for data processing. Section IV discusses the performance evaluation of the overall system, with a comparison of the impact of querying strategies, processing algorithms, and databases with relevant data models on service time. Finally, Section V reports our concluding remarks.

II. Background and Related Work

A. Spectrum Sensing Technologies and Relevant Applications

A significant literature exists on spectrum sensing, specifically on sensing algorithms for cognitive radio [3], [17]. The typical dimensions of the sensing operations are frequency, time, geographical space, code, and angle. Spectrum sensing can be conducted either individually, with each terminal performing radio detection and taking decisions, or cooperatively, in which a group of users jointly senses the spectrum to detect the presence of signals, e.g., by using consensus algorithms [18]. Some proposals on spectrum sensing leverage the Internet of Things (IoT) paradigm and make use of common mobile devices [19]–[22]. Their goal is to achieve a wide geographical coverage and to acquire real-time data while guaranteeing good sensing performance, even by using low cost mobile devices.

Spectrum sensing techniques can leverage one of the existing energy detection (ED) methods, waveform knowledge, cyclostationarity, radio identification, or matched filtering techniques, mentioned in increasing order of complexity [3]. Clearly, matched filtering is the optimum methodology, but it is the most complex one, as it requires a detailed knowledge of the primary user (PU) signal features [23]. In this work, due to the generality of the proposed approach and since we do not require to differentiate among different PUs, we have selected an ED methodology. This technique is simple, has a low implementation complexity, is signal features agnostic, and may even be used in a cooperative way [24]. In case of wideband spectrum sensing, multiband joint detection with frequency sweep [25] has both complexity and flexibility which fit our purpose of detecting the frequency occupation and/or transmission opportunities available in a given area with suitable temporal and spectral resolution.

Recently, cognitive radio researchers started applying machine learning algorithms [26], [27] and deep learning techniques [28] to classify spectrum portions available for opportunistic transmission (i.e., white spaces). A further application of spectrum sensing is automatic modulation classification techniques [29], [30], which can be used to gain information about the modulation of already present signals. Our architecture can provide easy, as-a-service access to large volumes of data to train and test cognitive radio algorithms, such as [26].

Finally, spectrum sensing is an enabler for cognitive radio solutions addressing peak traffic demands and traffic diversity in novel 5G cellular networks [31], as well as the deployment of LTE networks in unlicensed spectrum (LTE-U) with WLAN networks on 2.4 GHz and 5 GHz bands [19], providing data for managing coexistence issues.

B. Big Data Platforms for Spectrum Sensing

While great attention has been devoted to the sensor implementation by means of SDR, the selection of appropriate storage and computing technologies has not received significant contributions yet. Indeed, legacy SQL databases and classic computing architectures do not cope with the fast growing number of sensors and volume of data. In order to efficiently store data, new database models like NewSQL [32] and NoSQL [13] are designed to provide horizontal scalability. Furthermore, NoSQL databases often provide a non-fixed schema data model that allows for great flexibility.

The requirement of scalability can be met with the cloud computing paradigm. Although public clouds (such as Amazon Web Services[5] and Google Cloud Platform[6]) allow for short deployment time, private clouds are convenient long term solutions [33]. Our proposal is based on a private cloud.

Paper [34] presents an architecture for managing spectrum data trying to fulfil these requirements. They adopt a Lambda architecture similar to our solution, and include some components (i.e., Apache Kafka) used also in our proposal. However, in [34] a description of the data model and a system performance evaluation are missing. Instead, in this paper we present the database selection supported by the analysis of the impact of the relevant data model on system performance.

[3]https://www.mongodb.com/
[4]https://cassandra.apache.org/
[5]http://aws.amazon.com/
[6]https://cloud.google.com/

C. Database for Spectrum Sensing

The sensors relatively high data rate (see the analysis in Section III-A), their potentially growing number, and their heterogeneity make the spectrum sensing a Big Data problem, characterized by high Velocity, Volume, and Variety [35]. We constrain the database selection with the requirement of full clustering support, to allow for horizontal scalability, and availability of open source code. Furthermore, the data structures produced by our sensors impose another strict constraint for the database selection. Indeed, as detailed in Section III-A, a sensor produces an ordered array of hundreds to thousands floating point measurements for each sub-band of the spectrum to be analyzed (in the order of 1-25 MHz) for each complete sensing cycle. Such an array can be efficiently stored as binary data, thus the database is required to support the Binary Large OBject (BLOB) data type (or equivalent). Finally, sensor records includes a timestamp: therefore, a database optimized for time-series data seems to be a proper choice.

In our previous work [14], we showed that a classic relational database management system (DBMS) cannot cope with spectrum sensor data. Anyway, although SQL databases for time-series data exist, they seem not suitable for our use case. For instance, TimescaleDB[7] is a time-series database based on PostgreSQL[8], but it does not support full clustering features yet [36]. At the same time, attempts to rewrite relational DBMS to support scalable performance while not giving up on strong transactional and consistency requirements exist. These new systems are categorized as NewSQL [37]. Anyway, our proposed system does not require strong transactional and consistency requirements, since we do not handle high-profile data such as banking or financial data. As ACID (i.e., atomicity, consistency, isolation, durability) properties are not needed, we exclude the NewSQL database systems from our selection and we focus on NoSQL DBMS, which are natively designed with horizontal scalability in mind. Furthermore, differently from NewSQL, they fully support data variety [37], thus allowing integrating data from heterogeneous sensors.

The NoSQL panorama is vast and varied and a careful knowledge of data characteristic and relevant queries is extremely important for database selection. Among NoSQL time-series databases, we can consider InfluxDB[9] and Riak-TS.[10] The first one is specific for time-series data and supports SQL-like queries with its dedicated language, InfluxQL. However, InfluxDB clustering components are released with a closed source license [38], which is not acceptable for us. Riak-TS is fully open-source and scale horizontally on multiple machines. It also supports a subset of SQL statements for querying data. Anyway, it does not support the BLOB data type [39], thus it is not a compliant candidate. Thus, we considered DBMS not specifically optimized for time-series data. We selected Cassandra and MongoDB, a *columnar* and a *document-oriented* database, respectively. Both are open-source and widespread adopted NoSQL databases with clustering features, and they support the BLOB data type [40], [41]. They are also being used for time-series applications [42].

The following sub-sections briefly summarize operation principles of the two databases, since these are instrumental to the design process carried out in Section III.

1) MongoDB: MongoDB is a document database. A document is a storage unit composed of field-value pairs organized in a data structure based on a hierarchical tree. A document field can also contain multiple *embedded documents*. Embedded documents are used to represent relationships between data entities. By using relational databases, data entities are typically grouped by JOIN or similar operations. Differently, by using MongoDB, it is typical to group the maximum amount of information in a single document. In MongoDB the JSON standard is used to *describe* and examine a document, while it is *stored* in BSON (Binary JSON) data format for improved efficiency. Documents in MongoDB are grouped into *collections*. However, documents in the same collection are not required to have the same structure, except for some fields such as a unique document identifier, used to build the indexes. Collections can be distributed over clusters of multiple servers (*shards*) for horizontal scalability. Data distribution is based on values of one or more document fields, which are common to all documents in the collection, the *shard key*. The latter is used to partition data into *chunks*, sets of documents with shard key value included in a given range. Chunks can grow up to a maximum size (default value 64 MB): when this limit is reached, MongoDB efficiently splits the large chunk into two new chunks. Anyway, if a chunk contains documents with the same shard key value it can not be split. It is then referred to as a *jumbo chunk*, which is inefficient for MongoDB to manage. Thus, a careful design of the shard key is crucial to provide good query performance. Two data distribution strategies are available, based on the shard key: *ranged* and *hashed*. The first one uses the values of the shard key as they are, and documents with "close" values of the shard key are stored in the same chunk. This strategy is helpful to provide isolation of range-based queries, since the needed data probably reside in the same node and on contiguous disk portions. On the other hand, there are no guarantees on even data distribution across the cluster nodes. MongoDB can mitigate unbalanced data distribution by periodically migrating chunks across cluster nodes. The second strategy applies a hash function on the shard key: data are spread across the cluster nodes, obtaining a fair load balancing of write operations. As a drawback, a range-based query would require a scatter search on many nodes, leading to longer waiting times.

2) Cassandra: Apache Cassandra is based on the column-family-store model. Data are stored as rows, which are composed of a *row key* and a set of *columns* (name-value pairs). The row key (also called *partition key*) is used to retrieve data and a set of rows with the same value of the row key is stored sequentially for fast access. This is called column-family, or partition, and it should contain data that are often accessed together. One of the other columns can be used as a *clustering key*, which decides the order for the rows of the partition.

[7]https://www.timescale.com/
[8]https://www.postgresql.org/
[9]https://www.influxdata.com/time-series-platform/influxdb/
[10]http://basho.com/products/riak-ts/

Then, the remaining columns follow. Columns contain data and some meta-data (e.g., a timestamp) to address data expiration, conflicts resolution, etc. Columns in Cassandra can be added dynamically and rows in the same column family can have different columns. For this reason the Cassandra data model is defined "sparse". Cassandra can be distributed on clusters, where participating servers act as peers, organized in a logical ring structure. Data is spread across the cluster nodes using consistent hashing [43] on the partition key. Data can also be replicated. At a higher level of abstraction, data rows are grouped in *tables* and tables are grouped into *keyspaces*. A keyspace is a set of tables that are related to each other (e.g., tables related to the same application). Tables describe the row structure and data types, keys for partitioning and clustering, and indexes. Users can interact with keyspaces and tables by means of the Cassandra Query Language (CQL),[11] which is very similar to the traditional SQL language.

III. System Architecture

The proposed system architecture is depicted in Fig. 1. A sensing device and a software agent running on a local processing node compose the sensor. The agent decouples sensor operations from the core part of the system. It collects data from the sensing device, performs spectrum estimation, serializes processed data with binary encoding by Apache AVRO,[12] and sends the record to the dispatcher implemented by Apache Kafka, which guarantees potentially very high data ingestion rate due to its horizontal scalability, acting as a buffer for the computing and storage part. AVRO serialization ensures compatibility across multiple programming languages and systems, since the AVRO data format is preserved through all the chain components, till to the databases, without sacrificing performance. This configuration supports the ingestion of large data sets coming from many heterogeneous sensors. Data stored in the Kafka queue can be consumed by multiple programs at the same time, and are organized in categories called *topics*. In turn, topics are organized according to a hierarchical tree structure. This way a consumer can easily receive records from multiple topics by subscribing to a subtree with a regular expression. On the lower branch, a software agent receives data from Kafka and stores them in potentially multiple NoSQL databases. On the other branch, data are immediately consumed by a framework for big data analytics for near real-time analysis. We selected Apache Flink, since it supports both batch and streaming jobs [11]. As long as Flink processes the data stream, it can store results in the database for future uses.

On the right-hand side of Fig. 1, users interact with the system via a Web interface, which allows visualizing the status of radio spectrum by means of *heat map* pictures. The pictures are created by submitting a job request to Apache Flink.

The visualization application can produce *heat map* pictures representing the status of radio spectrum. Our application creates spectrum heat maps with three different strategies,

[11]https://docs.datastax.com/en/cql/3.3/cql/cqlIntro.html
[12]https://avro.apache.org/

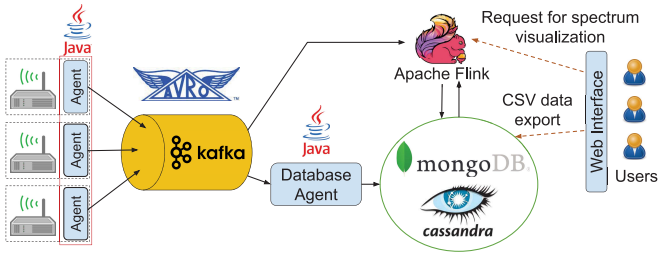


Fig. 1. Proposed system architecture with main software entities.

providing a trade-off between visualization accuracy and service time. Their selection mainly depends on the requested frequency visualization range. The first one, denoted as *draw*, is suitable to visualize medium-sized bandwidths (from 500 MHz to 1 GHz). For this, we designed and implemented a batch MapReduce job [16] executed by Flink on sensor data stored in the database. The color of each pixel of the heat map is generated by using aggregate power values (one per record). The second strategy, named *fft*, is similar to the previous one, but it uses the finer spectral data produced by the sensor to achieve an improved spectral resolution. It is suitable for representing low-medium sized bandwidths (from 1 MHz to few hundreds of MHz). Lastly, the third strategy (named *preview*) exploits the stream processing feature of Flink to create images representing very large portions of the spectrum (from 1 GHz to 6 GHz). Records originating from Kafka are processed by Flink to produce a pixel-based representation of a time-frequency window of configurable size. This aggregate information is stored in the database and made available for future requests to *quickly* produce the final image by combining window pixels. Anyway, this is only a client example: users could also provide custom Flink jobs for both batch and streaming data analysis to make use of the full potential of our platform. New jobs can be realized also by using our spectrum visualization as a template.

Moreover, users can export sensor data in CSV format for further analysis. We also foresee two more advanced ways to access measures. The first is to use an available REST API, which allows external users to query the database by interfacing with Flink, so that they can do batch analysis in their premises. Furthermore, we plan to implement a secure Kafka interface allowing external users to run streaming jobs on online data on their own infrastructure.

Finally, the system is modular and extensible. For instance, it can be extended to include edge computing elements by deploying other Kafka brokers on edge servers co-located with agents. They could run selected user jobs, able to fetch local data streams from these brokers, thus providing a low-latency service. The central Kafka broker would still continue collecting data from these local brokers and pass measures to the central database for persistent storage. The next sub-sections detail the core components of the system architecture shown in Fig. 1.

A. Spectrum Sensors

Spectrum Sensing is performed by SDR sensing devices with a multiband joint energy detection algorithm. These

TABLE I
THEORETICAL SCAN PARAMETERS FOR THE USED SDR DEVICES AND OPERATIONAL VALUES USED IN THE TESTBED

Parameter	RTL-SDR	USRP N210
Maximum digitization bandwidth, $r_{s,\max}$	3 MHz	25 MHz
Bits for I/Q samples	16 bit	32 bit
Maximum sensor data rate, $R_{S,\max}$	48 Mb/s	800 Mb/s
Actual digitization bandwidth, r_s	1 MHz	
Actual sensor data rate, R_S	16 Mb/s	32 Mb/s
FFT size, N_{FFT}	512	
Data averaging, N_{avg}	500	100
Scan bandwidth, B_S ($B_{S,min}, B_{S,max}$)	94 MHz (24-118 MHz)	1.6 GHz
	94 MHz (118-212 MHz)	(0.4-2 GHz)
	94 MHz (212-306 MHz)	2 GHz
	94 MHz (306-400 MHz)	(2-4 GHz)
Data precision, P_b	4 B	2 B
Freq. resolution, Δ_f	1.95 kHz	
Sub-band scan period, T	306 ms	52 ms
Freq. switching time, T_{lat}	50 ms	0.8 ms
Whole scan time, D	28.764 s	83.2 s
		104 s
Per sensor data agent rate, R_T	57.3 kb/s	180 kb/s
Agent data rate, R_A	$4\times R_T$ = 229kb/s	180 kb/s

devices are connected to the core platform through a dedicated agent. Table I reports the operational scan parameters used in our testbed, for different sensing devices. The Realtek SDR (RTL-SDR) device is a cheap USB 2.0 dongle with a maximum sampling rate of $r_{s,\max} = 3$ Msample/s and a maximum data rate of $R_{S,\max} = q \times r_{s,max} = 48$ Mbit/s, where q is the number of bits used to encode I & Q samples. Since R_S is small if compared with the maximum USB 2.0 rate, a single agent can potentially manage a few RTL-SDR sensors. On the other hand, the Ettus USRP N210 is a high-end SDR sensor, with a much higher sampling ($r_{s,\max} = 25$ Msample/s) and data rate ($R_{S,\max} = 800$ Mb/s) over IP networks.

Differently from common IoT scenarios, our sensors may produce data flows with significant data rate. When the number of sensors increases, streaming their data directly into the core portion of the platform is challenging, mainly due to ingestion bandwidth requirements. Thus, we decoupled the acquisition segment from the storage and processing segments by means of intermediate software agents. Their role is twofold. First, they decouple the sensor data sample format from the core system data format. This allows the platform to transparently integrate heterogeneous sensors. The agent is equipped also with a global navigation satellite system (GNSS) receiver, which allows it to accurately locate and timestamp geographically heterogeneous data. The second function consists of pre-processing sensors' data. This function reduces the data rate towards the core system (see R_S vs. R_T and R_A in Table I). The agent is logically split in two modules. The first one (acquisition module) manages the SDR device using the relevant software development kits (*librtlsdr* for RTL-SDR and *libuhd* for Ettus USRP), captures the baseband (BB) data, and processes them. Since this module interacts directly with the device driver, it is written in C/C++. An acquisition module

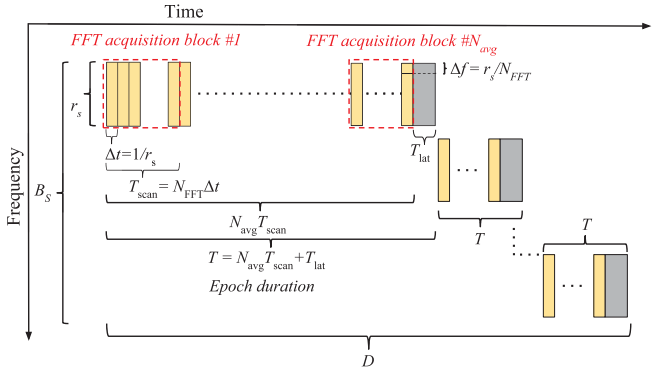


Fig. 2. Visualization of the spectrum sensing operation.

can manage a single SDR device. The second module (ingestion module) receives data from the former, performs AVRO formatting, and finally sends the processed records by using a Kafka client. It is implemented in Java. The two modules communicate by using a lightweight TCP-based protocol, over which both spectrum data and commands are exchanged. This software structure allows connecting multiple acquisition modules with a single ingestion one, which can run in a different device. In our system configuration, a single agent manages 4 RTL-SDR sensors, or a single USRP device.

B. Multiband Joint ED Algorithm

The time-frequency layout of the proposed multiband joint ED algorithm is shown in Fig. 2. The spectrum to be analyzed is in the range $[B_{S,min}, B_{S,max}]$, with $B_S = B_{S,max} - B_{S,min}$. Periodically, an SDR sensor scans a bandwidth of r_s Hz. We define the center frequency of the m-th scan epoch as $f_m = B_{S,min} + \frac{r_s}{2} + m\frac{B_S}{r_s} \bmod B_S$. The agent performs a short-term spectrum estimation based on Fast Fourier Transform (FFT) with the averaged periodogram method [19], [20], [44], [45] over N_{FFT} frequency bins with N_{avg} temporal averages. Thus, the time required to collect the samples for a single FFT is equal to $T_{scan} = N_{FFT}\Delta t = N_{FFT}/r_s$, where $\Delta t = 1/r_s$ is the time to obtain a single time sample, and each acquisition collects $N_{avg}N_{FFT}$ consecutive samples. After energy calculation, the resulting frequency bins represent the energy spectrum over a frequency-time slice of $\Delta_f = r_s/N_{FFT}$ Hz and $N_{avg}T_{scan}$ s. The duration of each scan epoch is equal to

$$T = N_{avg}\,T_{scan} + T_{lat} = N_{avg}N_{FFT}/r_s + T_{lat}, \quad (1)$$

where T_{lat} represents a delay due to latencies introduced by the SDR device when a frequency switch is needed, as well as the time to manage the request/response transaction.

Let $x_m^{(i)}[n]$ represent the i-th digitized BB acquisition block of the m-th analysis epoch obtained from the radio frequency (RF) analog signal $x_{RF}(t)$ after frequency down-conversion, low-pass filtering, and analog-to-digital conversion (ADC), as

$$x_m^{(i)}[n] = \left(\left(x_{RF}\left(t + t_{m,i}\right)e^{-j2\pi f_m t}\right) * h_{LP}(t)\right)_{t=\frac{n}{r_s}} \quad (2)$$

where $n = 0, 1, \ldots, N_{FFT} - 1$ and $t_{m,i} = t_0 + mT + iT_{scan}$ is the starting time of the i-th acquisition block of the m-th

epoch, t_0 is the starting time, and $h_{LP}(t)$ is the combined impulse response of the RF chain BB equivalent and ADC analog filters. After digitization, the averaged periodogram is

$$P_m[k] = \frac{1}{N_{avg}} \sum_{i=0}^{N_{avg}-1} \left| X_m^{(i)}[k] \right|^2, \; k = 0, \ldots, N_{FFT} - 1, \quad (3)$$

where $X_m^{(i)}[k]$ is the Discrete Fourier Transform (DFT) of the generic sub-chunk, $X_m^{(i)}[k] = \sum_{n=0}^{N_{FFT}-1} x_m^{(i)}[n] e^{-j\frac{2\pi}{N_{FFT}}kn}$. From the periodogram bins the average power value for the m-th chunk can be calculated as $\bar{P}_m = \sum_{k=0}^{N_{FFT}-1} P_m[k]$. Once the assigned scan epoch has elapsed, the SDR sensor switches to the next adjacent sub-band of r_s Hz, and begins another scan epoch (Fig. 2). Once the assigned scan bandwidth B_S has been covered, the system restarts the scan from the initial frequency. Since the single epoch scans are executed $\lceil B_S/r_s \rceil$ times, the total scan time is equal to

$$D = T\lceil B_S/r_s \rceil = \left(N_{avg} N_{FFT}/r_s + T_{lat}\right)\lceil B_S/r_s \rceil. \quad (4)$$

C. System Operational Range and Tradeoffs

Depending either on the expected change rate or on the wanted observation rate of the single frequency bins, from (4) it is possible to obtain the bounds of the operational range of the system. All data produced in an epoch are stored into a single record in the core system. The record production rate for a SDR sensor is $R_R = 1/T$; for instance, the USRP in the band 0.4-2.0 GHz has $R_R = 19.23$ records/s. The record production rate $r_{R,m}$ relevant to the m-th sub-band is given by $r_{R,m} = \frac{r_s}{B_s T}$. In the previous example, $r_{R,m} = 0.012$ Hz, which in a day produces 1038.4 records. As another example, let us consider an LTE band of size r_s =10 MHz. The band is divided in resource blocks of size $B_{rb} = 180$ kHz and duration $T_{rb} = 0.5$ ms. To capture the dynamicity of the transmission, we can setup a first inequality to obtain an appropriate N_{FFT}:

$$\Delta_f = r_s/N_{FFT} \leq B_{rb}. \quad (5)$$

From this we can find $N_{FFT} \geq r_s/B_{rb} = 55.5$. We can select a minimum value of $N_{FFT,min} = 64$. To properly select the N_{avg} parameter, we can use this second equation:

$$T = N_{avg} N_{FFT}/r_s \leq T_{rb}. \quad (6)$$

Using $N_{FFT,min} = 64$, we can find $N_{avg} \leq \frac{T_{rb} r_s}{N_{FFT}} = 78$. We can select a value of $N_{avg,min} = 50$. For different technologies, we can repeat the same procedure, obtaining different bounds.

The net data rate produced per SDR sensor is 8 $N_{FFT}P_b/T$ bit/s. In our case, each BLOB field stores the uncompressed averaged periodogram values, in machine-endian format, as a consecutive array of N_{FFT} elements, each one of P_b bytes. Table II reports the metadata that are associated with each transmission of the agent, which occurs every T seconds and consists of $M_b = 144$ additional bytes. Thus, for each sensor, the overall bit rate transmitted by the agent is equal to

$$R_T = 8(M_b + N_{FFT} P_b)/\left(N_{avg} N_{FFT}/r_s + T_{lat}\right) \quad (7)$$

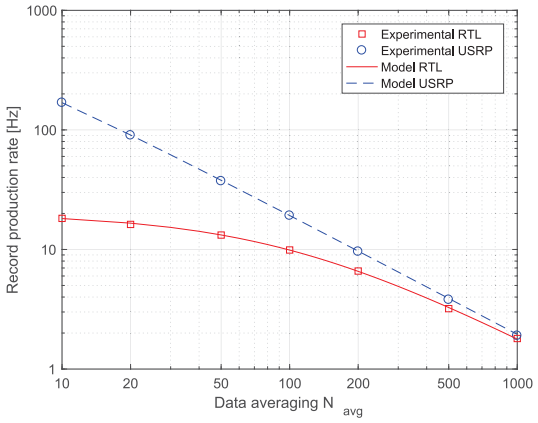


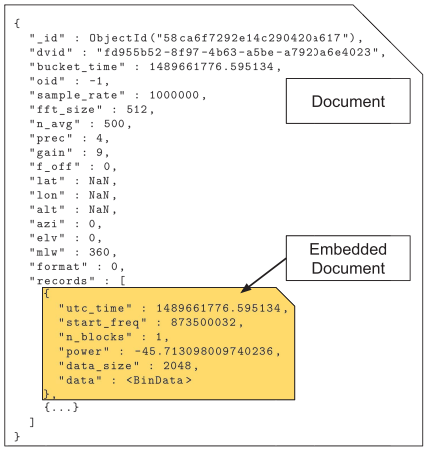
Fig. 3. Record production rate as a function of data averaging value for both the model based on (1) and the measurements of R_T by inverting (7).

TABLE II
DATA INGESTED IN THE PLATFORM BY AN AGENT

Field name	Type and size	Description
dvid	ObjectId, 12 B	Device ID
oid	ObjectId, 12 B	Order ID
utc_time	double, 8 B	UTC time stamp
start_freq	double, 8 B	Start frequency
sample_rate	double, 8 B	Sampling rate r_s
fft_size	int, 4 B	Analysis FFT size N_{FFT}
n_avg	int, 4 B	Number of averages N_{avg}
n_blocks	int, 4 B	Reserved (always set at 1)
prec	int, 4 B	Data precision P_b
gain	double, 8 B	Device RX gain
f_off	double, 8 B	Device frequency offset
lat	double, 8 B	Device position latitude
lon	double, 8 B	Device position longitude
alt	double, 8 B	Device position altitude
azi	double, 8 B	Antenna orientation azimuth
elv	double, 8 B	Antenna orientation elevation
mlw	double, 8 B	Antenna main lobe width
power	double, 8 B	Average power of scan data
format	int, 4 B	Data storage format
data_size	int, 4 B	Size of data field
data	blob, $N_{FFT}P_b$ B	Periodogram data
Total	144 + $N_{FFT}P_b$ B	Metadata + data size

where P_b is the number of bytes used to store each frequency bin value $P_m[k]$. In order to reduce the acquisition delay D, n_S parallel SDR sensors are deployed for acquisition. Its duration reduces to D/n_S, whereas the aggregated data rate increases to $R_A = R_T n_S$. Note that increasing n_S potentially implies an increased number of agents. Indeed, the number of SDR sensors attached to each agent cannot be unbounded, depending both on the processing and network load generated by the sensor and on the relevant capabilities of the agent.

While most values in Table I depend on system configuration, T_{lat} is a physical parameter characterizing each SDR device. Its value has been estimated by fitting multiple measurements of $R_R = 1/T$ (see (1)). The obtained values, reported in Table I, show excellent matches with experimental data (Fig. 3). It is also possible to artificially increase T_{lat}, thus decreasing the overall agent data rate. This may be useful to save network bandwidth or battery power of the sensor/agent.



(a)

dvid	bucket_time	bucket_freq	utc_time	data	data_size	...
61f5e41d...	2017-09-21	1	1505996700.000	0xAAB...	1024	
61f5e41d...	2017-09-21	1	1505996702.120	0xAAB...	1024	
61f5e41d...	2017-09-21	1	1505996705.200	0xAAB...	1024	

(b)

Fig. 4. Data model visualizations: (a) a MongoDB document with embedded documents; (b) Cassandra partition.

D. NoSQL Database and Data Model

In the following, we present a data model for MongoDB and Cassandra, discussing constraints, tradeoffs, and optimizations.

1) MongoDB: A first naive data model design for MongoDB consists of mapping each AVRO record coming from Kafka to a single document. However, this would imply to retrieve a lot of documents for large queries, whereas a good practice is to retrieve the minimum number of documents to satisfy a query. Therefore, we apply a *bucketing pattern* to include multiple records from the same sensor into a single document by means of MongoDB support for embedded documents. Fig. 4(a) shows a sample document from our system. We have added an extra field named `bucket_time`, including a timestamp and another field named `records` containing an array of embedded documents. We have stored as embedded documents all records arriving in the time window `[bucket_time, bucket_time+bucket_time_window]`. This data model allows skipping unfrequently changing fields for each record, while preserving the ability to perform range queries using `bucket_time` in place of `utc_time`, which is nevertheless stored within the embedded document. The `bucket_time_window` has been selected according to the MongoDB document size limit, which is 16MB. As a safe choice, we set it to 60 seconds.

The shard key selection faces the trade-off mentioned in Section II-C about even distribution of writes versus query isolation. Since our application mainly focuses on range-based queries (i.e., time ranges in spectrum data), we have decided to use a ranged shard key with the goal of optimizing read operations. In fact, using the device identifier only as a shard key, we would have optimal data locality since data coming from the same sensor are always stored by the same cluster node. Unfortunately, this leads to unsplittable chunks, since all documents have the same key value. Usage of `bucket_time` allows avoiding the large chunks problem. However, since the timestamp is monotonically increasing, all write operations would be routed to the same node, creating a "hot shard" and extremely poor load balancing and data distribution. Although MongoDB provides a background process to periodically balance the chunk distribution across cluster nodes, the presence of a "hot shard" would require a significant amount of both disk and network resources for the balance operations. In addition, the continuous writing of new data in our scenario on the "hot shard" is too fast to benefit from the balance effect. A compound key made of `dvid` and `bucket_time` is the suitable trade-off between the above issues.

```
{ "dvid":
  { "$in": ["dev-uuid-1" , "dev-uuid-2"] },
 "bucket_time":
  {"$gte":1.5056928E9, "$lt":1.5056964E9},
 "records": {
  "$elemMatch":
   {"start_freq": {"$gte":7.95E8, "$lt":8.05E8}}}
}
```

Code 1. MongoDB full query to produce a spectrum visualization.

```
SELECT * FROM raw_data
WHERE dvid='dev-uuid-1' AND bucket_time
IN ('2017-09-21','2017-09-22','2017-09-23')
```

Code 2. Cassandra query with the IN keyword on partition key values.

Queries in MongoDB are represented in JSON format, with the addition of special keywords to express equality or range conditions on selected fields. Code 1 shows a query example for our collection of spectrum sensor data. We select multiple device identifiers with the keyword `$in` and filter ranges of time and frequency with keywords `$gte` and `$lt`.

2) Cassandra: Figure 4(b) shows a graphical visualization of a partition. The first three columns, consisting of a device identifier (`dvid`), the acquisition time (`bucket_time`), and a frequency sub-band identifier (`bucket_freq`), form the partition key, which is the same for all rows. The fourth column, containing a timestamp, is the clustering key. Our data model for Cassandra consists in a table with a set of columns mapping the fields of the AVRO record. In a first version of our table, we followed the common and simple design pattern of using the device identifier (`dvid`) as partition key and a timestamp (`utc_time`) as clustering key. This pattern allows retrieving all data produced by a specific sensor and filter it out with a range condition on the timestamp. On the other hand, partitions would grow indefinitely as the sensor is active over time, with two major culprits: first, write operations from the same device will always be routed to the same cluster node; second, the Cassandra framework documentation explicitly advises against large partitions (>100MB)[13] because

[13]http://docs.datastax.com/en/dse-planning/doc/planning/planningPartitionSize.html

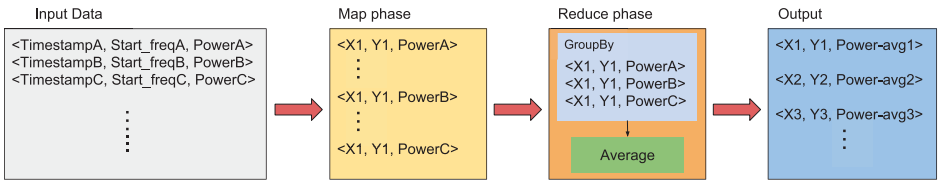
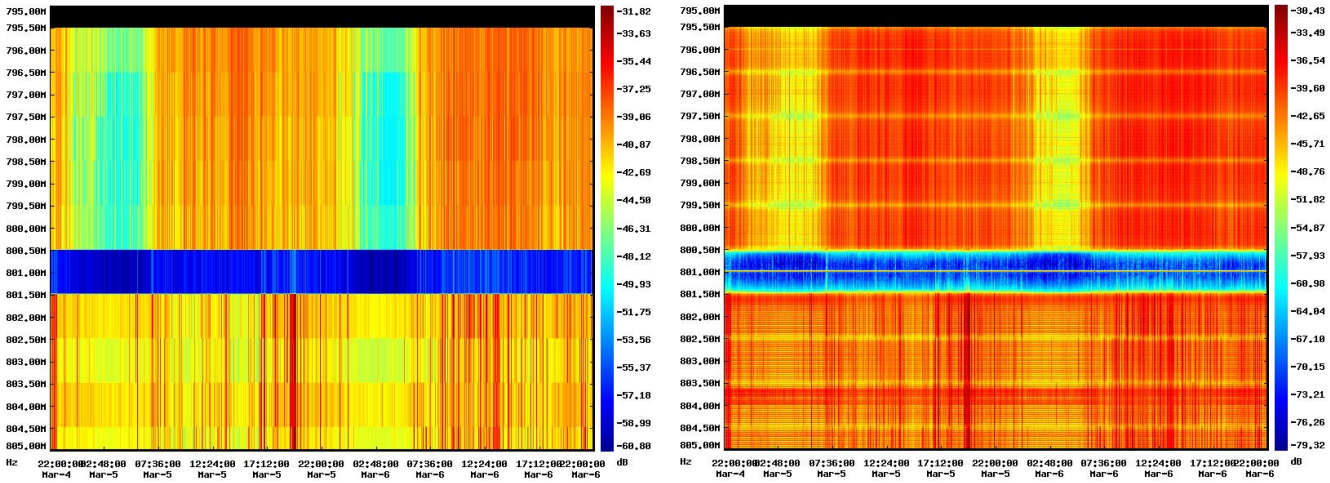


Fig. 5. MapReduce job flow diagram.



(a) *draw* (b) *fft*

Fig. 6. Radio spectrum images for a time period of 48 hours and a bandwidth going from 795 MHz to 805MHz, created with *draw* and *fft* strategy respectively.

they lead to inefficiencies. Therefore, we applied a *data bucketing* strategy to limit the partition size. We added an extra column named `bucket_time` (see Fig. 4(b)), representing the date to define a compound primary key. The drawback of this model is that the queries become more complex, since Cassandra does not allow range conditions on the partition key. To obtain the same result it is necessary to use the CQL keyword `IN`, as shown in Code 2. Anyway, this solution is quite inefficient since the query output can be huge, possibly leading to memory problems on the nodes and to long execution times, meaning that if the query fails a large amount of time and resources have been wasted. To overcome these issues, we programmatically split the query into multiple small sub-queries in our application. Each sub-query has a single equality condition on each value of the partition key. Details can be found in Section III-E.

Despite the aforementioned bucketing strategy on date values, we observed that partitions were still too large due to the high data rate of a single spectrum sensor (the data column contains a byte array representing N_{FFT} float values, see Table I). Hence we added another column, `bucket_freq`, to create a three-column partition key. The final table schema is shown in Fig. 4(b). The frequency based bucket is sized to identify a 100 MHz bandwidth and it is represented by an integer (i.e., 0 for frequencies in the range 0-99 MHz, 1 for 100-199 MHz frequencies, and so on). This strategy keeps partitions under the maximum suggested size and allows for coarse filtering on the frequency dimension when querying data.

E. Visualization Job

We designed a Flink application that produces images of the radio spectrum occupancy. Some examples are shown in Figs. 6 and 7. The image is a heat map based on signal power strength where the abscissa reports acquisition time, and the ordinate axis reports frequency. The application can be used as a template to create other ones, like classification algorithms for cognitive radio [26]. We designed two drawing algorithms providing different levels of spectral resolution: for wide bandwidths we use the *power* record field (*draw* algorithm), while for tighter bandwidth we use the *data* record field (*fft* algorithm). For the sake of simplicity, we illustrate in detail the *draw* algorithm, since the second is essentially the same algorithm applied to a larger data set.

We exploit the MapReduce paradigm to distribute processing across multiple machines. The job phases are shown in Fig. 5. The *Map* phase retrieves records from the NoSQL database and associates each power value with one pixel of the image based on requested picture size, time span, and frequency bandwidth. Multiple records can be associated with the same pixel and represent a contribution to the power value that will be shown in that pixel.[14] The result of this phase is a temporary dataset of *tuples* containing the coordinates of the mapped pixels and the power reported by the sensor record. The tuples are then grouped by pixel coordinates and in the *Reduce* phase the algorithm computes the average power in

[14]The same record can also be associated to multiple pixels, with weighted contributions on each one calculated as the portion of covered pixel area.

```
SELECT * FROM sensors_radio.draw
WHERE dvid IN ('dev-uuid-1','dev-uuid-2')
AND bucket_time IN ('2017-09-23','2017-09-24')
AND bucket_freq IN (1,2,3,4,5)
AND utc_time>=1.5025824E9 AND utc_time<1.502668799E9;
```

Code 3. Cassandra query to produce a spectrum visualization.

```
SELECT * FROM sensors_radio.draw
WHERE dvid = 'dev-uuid-1'
AND bucket_time = '2017-09-23'
AND bucket_freq = 3
AND utc_time>=1.5025824E9 AND utc_time<1.502668799E9;
```

Code 4. Cassandra sub query.

```
{ "_id" : ObjectId("59414d1887744d136d06147d"),
 "dvid" : "dev-uuid-1",
 "window_start" : 1497451740,
 "window_end" : 1497451770,
 "frequency_min" : 0,
 "frequency_max" : 6000000000,
 "spectrum" : [
  { "y" : 522, "count" : 1, "power" : -67.4801 },
  { "y" : 1860, "count" : 1, "power" : -60.3108 },
 ... ] }
```

Code 5. MongoDB document for preview data.

```
CREATE TABLE sensors_radio.pre_draw
 (dvid uuid, bucket_time date, window_start double,
 frequency_max double, frequency_min double,
 spectrum blob, window_end double,
 PRIMARY KEY ((dvid, bucket_time), window_start));
```

Code 6. Cassandra table for preview data.

each *tuple*. The *Reduce* result consists of a set of tuples. Each one represents *one* pixel with its associated power value. We use them to create a heat map image with the relevant color map. Black pixels represent lack of data.

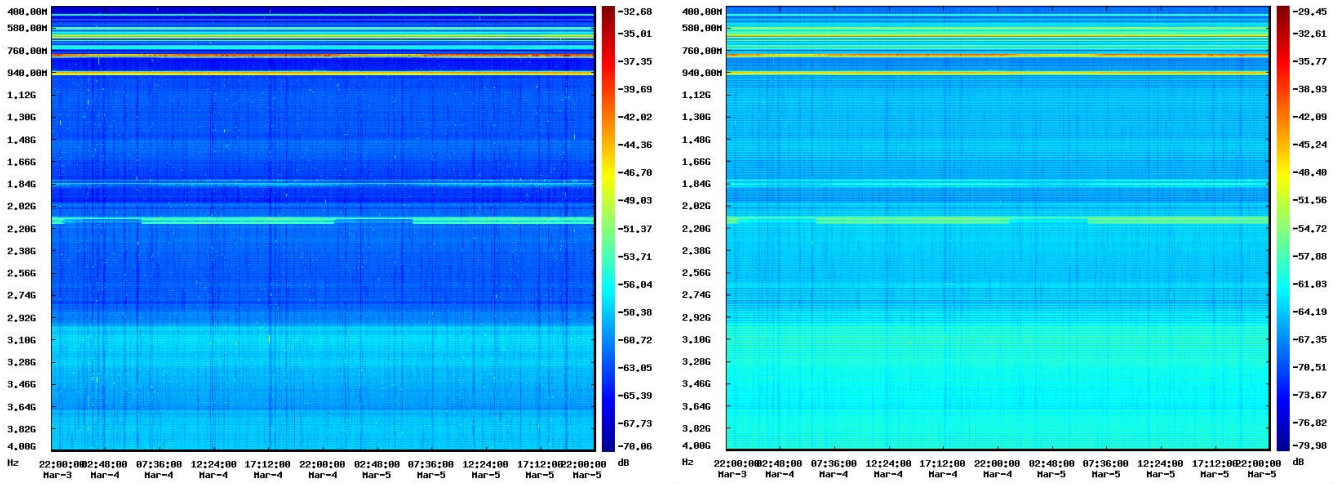
The second strategy uses the data field to provide higher spectrum resolution when tight bandwidths are visualized. Each record typically provides as many contributions to the image as the number of FFT samples (typically 512 or 1024). The result of the *Map* phase is a much larger datasets of tuples, if compared to the previous strategy. The *Reduce* phase is the same, producing at the end one tuple for each pixel.

Fig. 6 shows a comparison between an image produced by the *draw* strategy versus another one created by the *fft* strategy on the same 10 MHz bandwidth. Since color shapes are based only on one value per digitization bandwidth ($r_s = 1$ MHz), Fig. 6(a) shows 10 rows of thin rectangles. A more detailed visualization of the same bandwidth is shown in Fig. 6(b), where FFT data are used to achieve higher spectral resolution.

It is worth to note that our algorithm distributes not only data processing, but also data retrieval from database. The code to support parallel and distributed queries across Flink nodes has been written from scratch. A query to retrieve data for a spectrum visualization is split into multiple sub-queries. Thanks to task independence in the Map phase, sub queries can be distributed in any way across the nodes of the Flink cluster, and the resulting data can be immediately processed as they are produced. We use different strategies for splitting the original query in MongoDB and in Cassandra. For both databases, a query for a visualization job is characterized by a selective condition on device, time period, and bandwidth. A Cassandra example is shown in Code 3. The main query is split into multiple sub-queries with simple equality conditions by computing all the possible combinations of device identifier, date, and frequency bucket, as shown in Code 4. This is also compliant with the recommendation of avoiding queries with the `IN` keyword[15] in Cassandra. The set of sub-queries is then equally distributed across the Flink nodes. For MongoDB we create sub-queries by splitting the requested time range in multiple sub-ranges of configurable size (e.g., 1800 or 3600 seconds) and assign them to Flink nodes. Code 1 shows a query example to produce a spectrum heat map. Sub-queries keep the same structure, although with a modified range on the `bucket_time` field.

Apache Flink also supports the execution of jobs on streams of real-time data. As shown in Fig. 1, the system architecture allows Flink to directly read and process streams of sensor records from Kafka. We exploited this feature by developing a job to pre-process sensor data with the goal of speeding up creation of spectrum images for large bandwidths and acquisition time. We call this strategy *preview*. In fact, although the *draw* approach uses a single power value for each record to produce an image, this operation can take a consistent amount of time for a large bandwidth (i.e., 2-3 GHz or more) combined with large time periods (2 days or more). The *preview* job is still a MapReduce job working on a continuous stream of data. Time is split into *windows* of customizable size and MapReduce operations are executed for each window. The output produced for each window is a vertical line of pixels of fixed height and width. We selected 6000 pixels in height, each one representing a bandwidth of 1 MHz in a time window of 30 seconds. Clearly, these dimensions would result in poor spectral resolution on tight bandwidths, but we find this limitation acceptable, since we plan to use this strategy on very wide time and frequency ranges. The stream of records is read from Kafka and split into sub-streams, one for each sensor. The Map phase is very similar to that of the *draw* strategy: it maps the power value of each sensor record to a vertical coordinate. The Reduce phase produces the final pixel descriptions by grouping and averaging data, which are then written in the database. For MongoDB, a dedicated collection is created to store *preview* data. A document example is shown in Code 5. Data produced in a time window by a single sensor are stored in an array of embedded documents in the "spectrum" data field. Similarly, a dedicated table is created in Cassandra (Code 6). Each row stores the window data in the "spectrum" field in binary format (BLOB type) for better efficiency. When the user requests an image with the *preview* strategy, the final image is produced by merging windows related to different sensors and by placing the resulting pixel lines side by side. The output is an image similar to the one produced by the visualization job for large bandwidths.

[15]https://docs.datastax.com/en/dse/5.1/cql/cql/cql_reference/cql_commands/cqlSelect.html



(a) *draw* (b) *preview*

Fig. 7. Radio spectrum images for a time period of 48 hours and a bandwidth going from 400 MHz to 4 GHz, created with *draw* and *preview* strategies, respectively. Please note the *different* minimum and maximum power levels in the color map.

Fig. 7(b) shows an example. The result is close to that obtained with the *draw* strategy.

IV. Performance Evaluation

A. Testbed Configuration

The whole system architecture in Fig. 1 has been implemented in a real testbed. We used 4 RTL-SDR sensors connected to a PC via USB and two USRP N210 sensors connected to another PC via Ethernet LAN. The sensors are deployed in a room at the University of Perugia, department of Engineering, first floor. The room has large windows and the devices sense an urban area with medium density population. All the other software components are deployed on virtual machines (VMs) hosted on a private cloud managed through Openstack Mitaka.[16] We used three servers (with an aggregate amount of 118 CPU cores and 320 GB RAM) and a Network Attached Storage (NAS) with 32 TB capacity, served by a 10 Gbps LAN. Kafka is hosted on a single VM, together with the VM running the Apache Web server, in the server acting as cloud controller. Database VMs are hosted by a single server, which is connected at 10 Gbps to the NAS providing iSCSI volumes. The NAS is configured in a RAID 10 disk array, ensuring high read performance and fault tolerance. Flink runs in a cluster of five VMs on the third server. Four VMs operate as *Task Managers* to execute parallel operations on data, while the other VM acts as *Job Manager*, a coordinator and central hub for job submission. Each Flink node is configured with 8 virtual CPUs (vCPUs), 10 GB of RAM and offers one task slot per each vCPU.

B. Analysis of Experimental Data

Table III reports the frequency (Δf) and time resolution (T) service metrics that our platform can offer to users. The values depend on both the sensor features and the configuration parameters of the sensing algorithm, including the digitization bandwidth r_S, the number of time averages N_{avg}, and number of FFT points N_{FFT} (see equations (5) and (6)). Table III reports the values for r_S =1 MHz, and a set of values (minimum, recommended, and maximum) for N_{avg} and N_{FFT}. These values illustrate of the resolution ranges of our platform.

TABLE III
Frequency and Time Resolution for r_S =1 MHz

N_{avg}	N_{FFT}		
	64	**512**	**2048**
10	Δf = 15.6 kHz, T = 0.64 ms	Δf = 1.95 kHz, T = 5.12 ms	Δf = 488 Hz, T = 20.5 ms
100	Δf = 15.6 kHz, T = 6.4 ms	Δf = 1.95 kHz, T = 51.2 ms	Δf = 488 Hz, T = 204.8 ms
1000	Δf = 15.6 kHz, T =64 ms	Δf = 1.95 kHz, T = 512 ms	Δf = 488 Hz, T = 2.048 s

The rest of our performance evaluation is focused on the measurement of job execution times of different combinations of both databases and visualization algorithms (user experience). Tests are executed during data ingestion from sensors, as in normal system operation. Figures show the amount of time necessary to execute a visualization job (in seconds) versus the observation time range, expressed in hours, for a given frequency bandwidth. We evaluate six meaningful time ranges: 12, 24, 48, 96, 168, and 240 hours. In each figure, the frequency range is reported in the figure title. We selected a tight bandwidth of 10 MHz (795 MHz-805 MHz), a medium sized bandwidth of 100 MHz (795 MHz-895 MHz), and a very large bandwidth of ~ 4 GHz comprising the whole spectrum we are able to sense with our devices. Error bars show the 95th percentile confidence intervals.

1) Parallel Queries: Figs. 8(a) and 8(b) show the execution times for visualization jobs implementing the *draw* strategy (Section III-E). Fig. 8(a) shows the results for a bandwidth of 100 MHz. The parallel query strategy allows significantly improving job completion time for both MongoDB and Cassandra. This reduction ranges from 24% to 46% for MongoDB and from 50% to 63% for Cassandra, which outperforms MongoDB. To better show the performance

[16]https://www.openstack.org/software/mitaka/

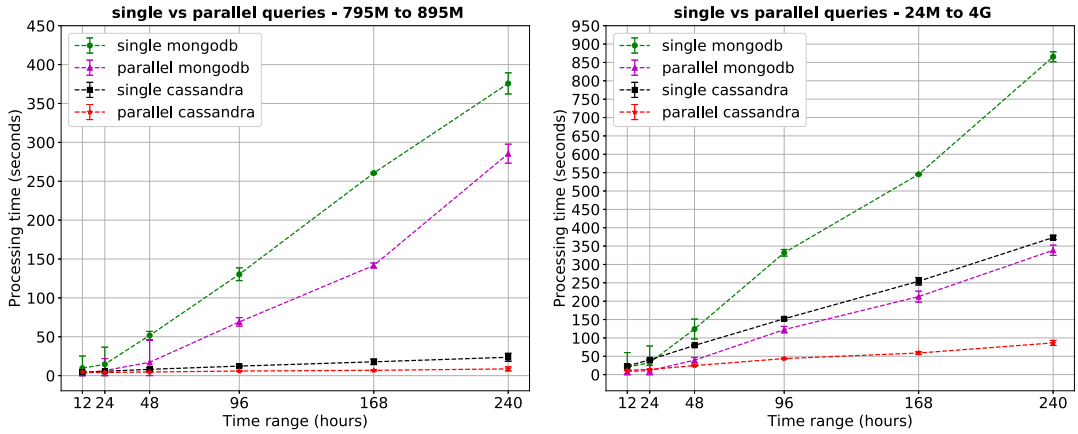


Fig. 8. Performance comparison of different query strategies and databases for: (a) $B_S = 100$ MHz; (b) $B_S \sim 4$ GHz.

improvement in Cassandra, we run the same job on a very large bandwidth, from 24 MHz to 4 GHz. The relevant image should be rendered by the *preview* strategy; we use this configuration as a stress test to retrieve and process a large amount of data. Fig. 8(b) shows the results of these experiments. The single query strategy on Cassandra shows a quite constant increasing behavior from small to large time ranges. The parallel query execution can scale better and achieves a nearly constant improvement of $\sim 75\%$ on the whole time range. For MongoDB, the figure shows a consistent improvement ($\sim 67\%$), confirming our previous observations. In synthesis, splitting each query in a number of small sub-queries always improves job completion times, especially with large datasets.

2) Comparing Drawing Strategies and Databases: Fig. 9 shows four charts with a performance comparison among the *draw*, *fft*, and *preview* strategies. Each chart shows results for two visualization strategies executed both on Cassandra and MongoDB. We first observe that processing times for MongoDB show generally larger error bars compared to Cassandra, since the former is more sensitive to cache warm-up, which influences results more for small time ranges. Figs. 9(a) and 9(b) compare the *draw* and the *fft* strategies on a bandwidth of 10 MHz and 100 MHz, respectively. We expect that the *fft* strategy takes more time to visualize the data set for the same database. This is suggested by the fact that more data (i.e., FFT samples) are required to obtain a more detailed image of the radio spectrum. Anyway, from Fig. 9(a) it appears that MongoDB shows comparable performance for both strategies. We ascribe this to the fact that, for a tight bandwidth (corresponding to small volume of data), more time is spent searching for data rather than for returning them to Flink. On the other hand, Cassandra generally outperforms MongoDB in this configuration, since the introduction of frequency buckets allows for an accurate frequency filtering and a more efficient data search. We can also observe that the *fft* strategy needs more time in Cassandra, as expected, but it keeps the performance acceptable even for large time windows. In Fig. 9(b) we use a larger bandwidth of 100 MHz. The *draw* strategy on Cassandra shows the same behavior observed in the previous figure, with a very modest increase in job completion time (only $\sim$5% for large time ranges). This is expected, since the frequency buckets are bounded to 100 MHz: the queries for 10 MHz or 100 MHz require retrieving the same volume of data, since both ranges include the edge between 700 and 800 MHz (two buckets are retrieved, although the desired frequency interval is 10 times larger). Clearly, more data must be processed by Flink but, since every record contributes to the heat map with a single value, the amount of work does not increase enough to show an appreciable difference. However, this difference can be shown when the *fft* strategy is used on Cassandra, where the execution time grows with the time range considerably. For this bandwidth (100 MHz), the performance of MongoDB is that expected with the *fft* strategy, which takes more time than the *draw* one. A comparison between the two databases shows that Cassandra outperforms MongoDB for large time ranges, while for shorter ones they have a comparable performance.

Fig. 9(c) shows a comparison between *draw* and *preview* strategies for the frequency range 24 MHz-4 GHz. The chart highlights the benefit of using pre-processed data on very large frequency bandwidths, resulting in a significant reduction in job execution time of $\sim 3-3.5$ times, especially for large time range queries, for both MongoDB and Cassandra. Cassandra still outperforms MongoDB on large time ranges not only with the *draw* strategy, but also when *preview* is used. In particular, producing an heat map with the *draw* strategy with Cassandra performs like doing it with the *preview* strategy on MongoDB.

Although *preview* has been designed to be used with wide frequency ranges (a few GHz), by visual inspection we noticed that even for tight frequency ranges it can give a very coarse, but still meaningful information. Therefore, in Fig. 9(d) we evaluate the performance of *draw* and *preview* strategies also on a 100 MHz bandwidth. The performance of MongoDB is as expected, whereas we can observe the opposite for Cassandra, with the *draw* strategy slightly outperforming *preview* on large time ranges. This happens since the frequency bucketing storage pattern used in Cassandra allows a better filtering of unnecessary data in advance, which is very effective for tight bandwidths. On the other hand, the *preview* strategy uses pixel

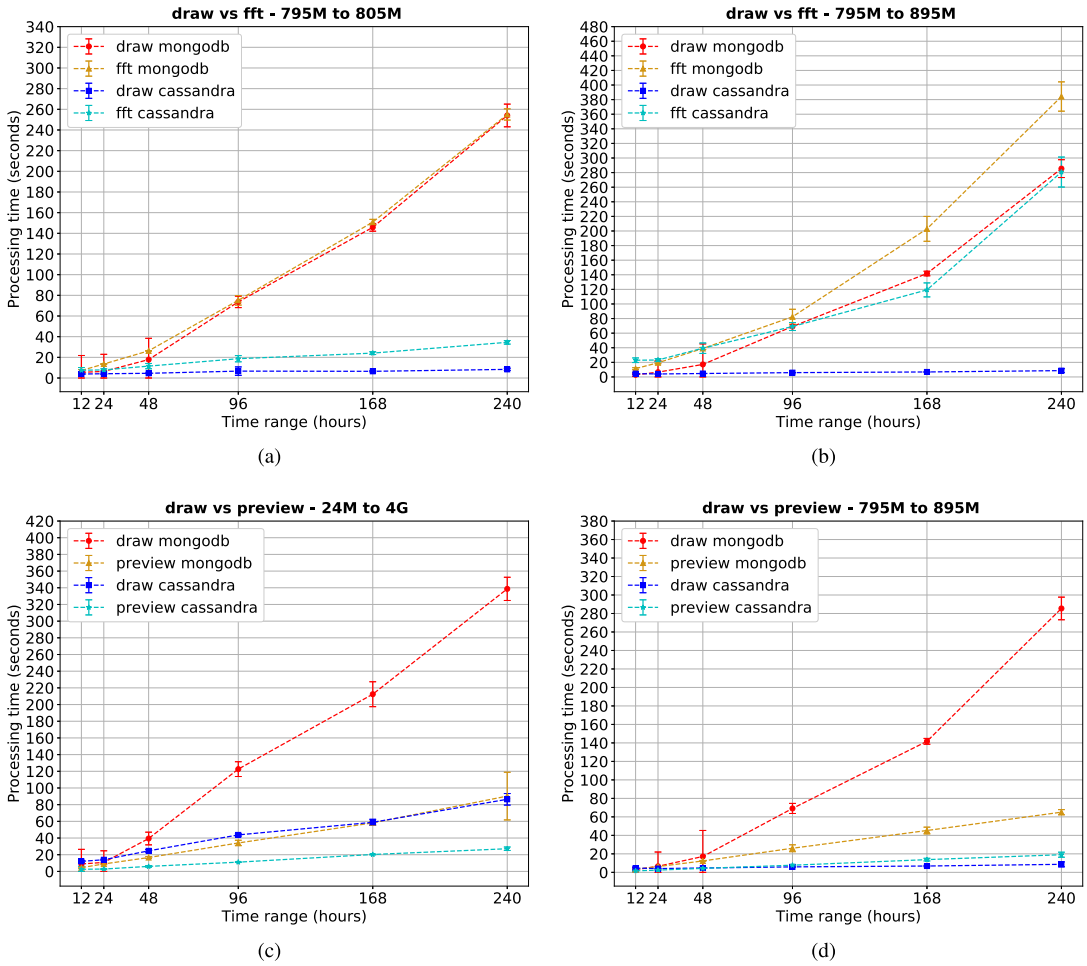


Fig. 9. Performance comparison of different visualization strategies for various frequency bandwidths: (a) *draw* vs. *fft* visualization strategy for $B_S = 10$ MHz; (b) *draw* vs. *fft* visualization strategy for B_S =100 MHz; (c) *draw* vs. *preview* visualization strategy for $B_S \sim 4$ GHz; (d) *draw* vs. *preview* visualization strategy for B_S =100 MHz.

data records for a very large bandwidth (provisionally set to 6 GHz) and a large part of these data are discarded when a 100 MHz bandwidth visualization is requested.

From the performance figures presented up to now, it is evident that Cassandra manages more efficiently the data retrieving operation than MongoDB. This is true always, and it is more evident when the query requires a narrow frequency filtering operation, as in Fig. 9(d). This is due to the way the query itself is built. Indeed, by looking to Code 1 versus Code 3 and 4, it is evident that MongoDB trades flexibility for performance. When using MongoDB, the query to be prepared by the application job is really simple, since standard range-based queries are allowed. From Code 1, the `$elemMatch` operation done by MongoDB takes care of going down to each sub-document, retrieving only those documents whose at least one sub-document matches the relevant condition. This also implies that the database returns a lot of data that have to be filtered out by the application server (Flink in our case), since for each document only a few sub-documents are useful. This strongly depends on the acquisition process. If we consider the range of 100 MHz analyzed in Fig. 9(d) (795-895 MHz), since $R_R^{USRP} = 19.23$ record/s, thus the net production rate of a single MongoDB sub-document is $r_{R,m}^{USRP}$ =0.012 record/s. This means that in *each* MongoDB document there are, on average, $r_{R,m}^{USRP} \times \tau = 0.72$ embedded documents relevant to the desired sub-band out of $R_R^{USRP} \times \tau = 1153.8$ sub-documents embedded in a single document, where $\tau = 60$ s is the MongoDB bucket time. The percentage of *interesting* documents is only $\rho_{USRP}^{Mongo} = \frac{r_{R,m}^{USRP} z\tau}{R_R^{USRP}\tau} = \frac{zr_s}{B_S} = 6.25\%$, where z represents the useful frequency interval (in units of r_S Hz) out of the whole scan bandwidth of the device, $B_S = 1600$ MHz.

Instead, when using Cassandra, the application itself has to indicate in the query the specific partitions where picking the data. This implies a deeper knowledge of how data are organized *inside* the database. In this specific case, the query spans the two frequency buckets with ranges 700-800 MHz and 800-900 MHz; by considering them together (i.e., $B_S = 200$ MHz), we obtain an overall efficiency of $\rho_{USRP}^{Cass} = 50\%$, which is definitely higher than the MongoDB case.

In Fig. 10, we compare the service time of the two databases for a fixed time window of a week, for the three drawing strategies presented above, as a function of the frequency interval to monitor. Fig. 10a shows absolute numbers, whereas Fig. 10b shows the service times of MongoDB normalized

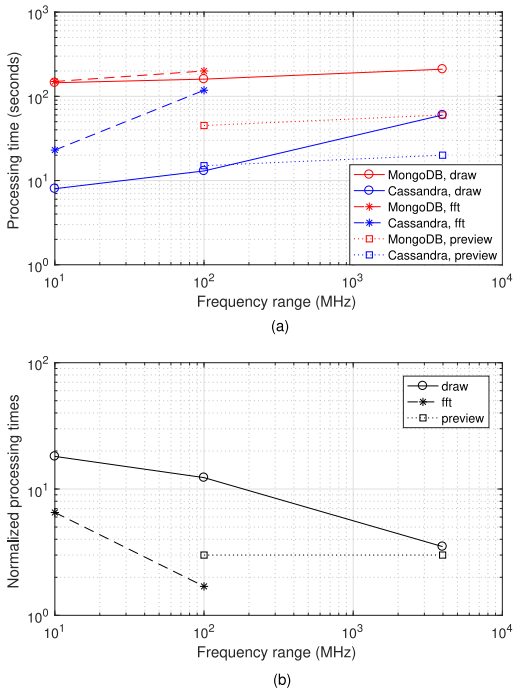


Fig. 10. Performance comparison for a time window of 7 days of the three drawing strategies (draw, fft, and preview) over different frequency ranges: (a) MongoDB vs. Cassandra processing times, and (b) MongoDB processing times normalized to Cassandra's ones.

to those of Cassandra. When the *draw* strategy case is used, Cassandra outperforms MongoDB with a speed-up of about 20x for small frequency intervals for the reason explained, whereas, when the frequency range increases to about 4 GHz and thus both ρ^{Cass} and ρ^{Mongo} are equal to 1, it shrinks to about 3x, which is the performance penalty associated with the usage of MongoDB with respect to Cassandra. This is also confirmed by the analysis of the *preview* case, where both databases use a separate data structure, illustrated in Code 5 and 6. When a preview image is produced, the application always retrieves the data record associated with the whole frequency bandwidth of 6 GHz for the requested time windows, and extracts the useful info. Thus, the difference in service time is only associated to the difference of the database. Fig. 10b confirms that the Cassandra speedup is around 3x, independently of frequency range.

As for the *fft* strategy, for very small frequency ranges (i.e., 10 MHz), the Cassandra speedup is about 6x, but it decreases with frequency range to about 1.5x for 100 MHz. In particular, from Fig. 10a it emerges that, for small frequency ranges, the MongoDB service time for *fft* and *draw* is very similar. This is due to the fact that, even if the *fft* task is more computationally expensive, the application server is able to process data as they come from the database, which is the main performance bottleneck. When the amount of data increases, also the computing requirements increase, thus the production of a *fft* images requires more time. We have verified this with specific micro-benchmarks, not shown here due to space limitations. This is also confirmed by the Cassandra case, showing that the larger processing time almost frustrates the advantage provided by the database.

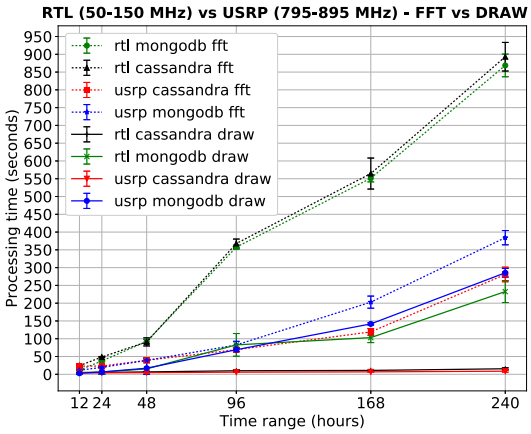


Fig. 11. System performance on two different bandwidths of the same size (100 MHz) respectively sensed with different devices.

3) Impact of Sensor Heterogeneity: To extend the analysis and evaluate the impact of heterogeneous sensors (different model and configuration, shown in Table I) on our architecture, we have compared the system performance for two different bandwidths of 100 MHz, sensed by different devices. Fig. 11 shows the processing times for a visualization job using both the *fft* and *draw* strategies for the two databases. In the comments to Fig. 10, we have highlighted how, for the *ftt* strategy, the performance of the two databases tends to become similar with USRP devices in the 100 MHz configuration. We will now explain why, when the sensing device is the RTL-SDR, differences basically vanish for the *fft* case, stressing again the impact of data models on system performance.

When considering the bandwidth of 50-150 MHz, two RTL-SDR devices were involved, whereas the frequency interval 795-895 MHz was scanned by a USRP device (see Table I). The first comment is that the service times of the two databases are basically overlapped for RTL-SDR sources. Since each RTL-SDR device produces $R_R^{RTL} = 3.2$ record/s over a B_S =94 MHz, the rate for a single sub-band of 1 MHz is $r_{R,m}^{RTL}$ =0.034 record/s. Thus, repeating the reasoning done for the USRP in the comment to Fig. 10, we get an average percentage of *interesting* documents $\rho_{RTL}^{Mongo} = 53.19\%$ for each query. *Exactly* the same value holds for Cassandra, since the frequency bucket coincides with B_S for each device. Thus, it is reasonable to have similar service times, considering also the increased processing requirements associated with the *fft* strategy, which masks the advantages of Cassandra seen in Fig. 10. When data density becomes high on tight frequency ranges, the MongoDB data model increases its efficiency, since each document includes many records contributing to produce a visualization, with lower metadata overhead than Cassandra, due to its hierarchical structure. If ρ is equal, we expect the two databases to perform similarly, especially if the load on the application server masks the residual performance differences. This is confirmed by the experiments.

Instead, when the data density of MongoDB and Cassandra is quite different, as it happens with the USRP device, the impact of ρ becomes dominant, especially for large jobs. Furthermore, results are different for the *draw* strategy, which

is less computationally intensive than the *fft* one. Looking to the *draw* curves in Fig. 11, it results that Cassandra definitely outperforms MongoDB, even if their queries have the same ρ (RTL-SDR case). Also, it is interesting to observe that, in the MongoDB case (*draw* strategy), even if the amount of useful data to process is larger for the RTL-SDR source, the amount of data to retrieve is much larger in the USRP case (very low value of ρ). Since the processing load of Flink is low, this causes the database to be the bottleneck, resulting in longer service times in the USRP case for large time windows.

The second comment is that a longer time is needed to process data coming from RTL-SDR devices (circle and square markers) for the *fft* strategy. Indeed, although these devices are cheaper and less performing than USRPs, they produce data with a higher density of records per digitization frequency due to their configuration (Table I). If we look to the corresponding value in the USRP case calculated in Section IV-B2, we can observe that, given its configuration and capabilities, an RTL-SDR device produces three times more records per digitization frequency ($r_{R,m}$) than a USRP device (0.034 vs. 0.012 records/s). This leads to more data to process per job, although the requested total bandwidth and time range have the same values. Thus more time is needed to complete it.

V. Conclusion

This work presents the design and implementation of a sensing as a service system, specialized for the spectrum sensing as a service use case (S^3aaS). We characterized spectrum sensors data flow and, by using the results of this analysis, we focused on the design process of sensor data model in two popular open-source NoSQL databases, highlighting design choices and relevant motivations.

To analyze the system, we developed a MapReduce application for spectrum visualization and designed different drawing strategies to fit image quality with given requested time and frequency ranges, by trading off accuracy for job execution times. We carried out an experimentation to assess the system operation, which allowed us to analyze its performance in detail. This allowed us to carry out query optimization on Apache Flink and to quantify the impact of data models on the overall system performance, finding that Cassandra outperforms MongoDB in most cases.

References

[1] I. F. Akyildiz, W.-Y. Lee, M. C. Vuran, and S. Mohanty, "A survey on spectrum management in cognitive radio networks," *IEEE Commun. Mag.*, vol. 46, no. 4, pp. 40–48, Apr. 2008.

[2] B. Wang and K. J. R. Liu, "Advances in cognitive radio networks: A survey," *IEEE J. Sel. Topics Signal Process.*, vol. 5, no. 1, pp. 5–23, Feb. 2011.

[3] T. Yucek and H. Arslan, "A survey of spectrum sensing algorithms for cognitive radio applications," *IEEE Commun. Surveys Tuts.*, vol. 11, no. 1, pp. 116–130, 1st Quart., 2009.

[4] A. B. Flores, R. E. Guerra, E. W. Knightly, P. Ecclesine, and S. Pandey, "IEEE 802.11af: A standard for TV white space spectrum sharing," *IEEE Commun. Mag.*, vol. 51, no. 10, pp. 92–100, Oct. 2013.

[5] X. Chen and J. Huang, "Database-assisted distributed spectrum sharing," *IEEE J. Sel. Areas Commun.*, vol. 31, no. 11, pp. 2349–2361, Nov. 2013.

[6] T. Ulversoy, "Software defined radio: Challenges and opportunities," *IEEE Commun. Surveys Tuts.*, vol. 12, no. 4, pp. 531–550, 4th Quart., 2010.

[7] X. Sheng, J. Tang, X. Xiao, and G. Xue, "Sensing as a service: Challenges, solutions and future directions," *IEEE Sensors J.*, vol. 13, no. 10, pp. 3733–3741, Oct. 2013.

[8] A. Zaslavsky, C. Perera, and D. Georgakopoulos, "Sensing-as-a-Service and big data," in *Proc. ACC*, Bengaluru, India, 2012.

[9] A. Ghasemi and E. S. Sousa, "Spectrum sensing in cognitive radio networks: Requirements, challenges and design trade-offs," *IEEE Commun. Mag.*, vol. 46, no. 4, pp. 32–39, Apr. 2008.

[10] N. Marz and J. Warren, *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. Shelter Island, NY, USA: Manning, 2015.

[11] P. Carbone *et al.*, "Apache flink: Stream and batch processing in a single engine," *Data Eng.*, vol. 38, no. 4, pp. 28–38, 2015.

[12] J. Kreps, N. Narkhede, and J. Rao, "Kafka: A distributed messaging system for log processing," in *Proc. NetDB*, 2011.

[13] A. Corbellini, C. Mateos, A. Zunino, D. Godoy, and S. Schiaffino, "Persisting big-data: The NoSQL landscape," *Inf. Syst.*, vol. 63, pp. 1–23, Jan. 2017.

[14] G. Baruffa, M. Femminella, M. Pergolesi, and G. Reali, "A big data architecture for spectrum monitoring in cognitive radio applications," *Ann. Telecommun.*, vol. 73, nos. 7–8, pp. 451–461, Aug. 2018.

[15] G. Baruffa, M. Femminella, M. Pergolesi, and G. Reali, "A cloud computing architecture for spectrum sensing as a service," in *Proc. IEEE CIoT*, Paris, France, Nov. 2016, pp. 1–5.

[16] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[17] H. Sun, A. Nallanathan, C.-X. Wang, and Y. Chen, "Wideband spectrum sensing for cognitive radio networks: A survey," *IEEE Wireless Commun.*, vol. 20, no. 2, pp. 74–81, Apr. 2013.

[18] Z. Li, F. R. Yu, and M. Huang, "A distributed consensus-based cooperative spectrum-sensing scheme in cognitive radios," *IEEE Trans. Veh. Technol.*, vol. 59, no. 1, pp. 383–393, Jan. 2010.

[19] T. Zhang *et al.*, "A wireless spectrum analyzer in your pocket," in *Proc. HotMobile*, 2015, pp. 69–74.

[20] A. Chakraborty and S. R. Das, "Designing a cloud-based infrastructure for spectrum sensing: A case study for indoor spaces," in *Proc. IEEE DCOSS*, Washington, DC, USA, May 2016, pp. 17–24.

[21] S. Rajendran, W. Meert, D. Giustiniano, V. Lenders, and S. Pollin, "Deep learning models for wireless signal classification with distributed low-cost spectrum sensors," *IEEE Trans. Cogn. Commun. Netw.*, vol. 4, no. 3, pp. 433–445, Sep. 2018.

[22] A. Nika *et al.*, "Empirical validation of commodity spectrum monitoring," in *Proc. SenSys*, 2016, pp. 96–108.

[23] D. Cabric, S. M. Mishra, and R. W. Brodersen, "Implementation issues in spectrum sensing for cognitive radios," in *Proc. IEEE ACSSC*, Pacific Grove, CA, USA, Nov. 2004, pp. 772–776.

[24] I. Sobron, P. S. Diniz, W. A. Martins, and M. Velez, "Energy detection technique for adaptive spectrum sensing," *IEEE Trans. Commun.*, vol. 63, no. 3, pp. 617–627, Mar. 2015.

[25] Z. Quan, S. Cui, A. H. Sayed, and H. V. Poor, "Optimal multiband joint detection for spectrum sensing in cognitive radio networks," *IEEE Trans. Signal Process.*, vol. 57, no. 3, pp. 1128–1140, Mar. 2009.

[26] F. Azmat, Y. Chen, and N. Stocks, "Analysis of spectrum occupancy using machine learning algorithms," *IEEE Trans. Veh. Technol.*, vol. 65, no. 9, pp. 6853–6860, Sep. 2016.

[27] K. M. Thilina, K. W. Choi, N. Saquib, and E. Hossain, "Machine learning techniques for cooperative spectrum sensing in cognitive radio networks," *IEEE J. Sel. Areas Commun.*, vol. 31, no. 11, pp. 2209–2221, Nov. 2013.

[28] F. Paisana *et al.*, "Context-aware cognitive radio using deep learning," in *Proc. IEEE DySPAN*, 2017, pp. 1–2.

[29] O. A. Dobre, A. Abdi, Y. Bar-Ness, and W. Su, "Survey of automatic modulation classification techniques: Classical approaches and new trends," *IET Commun.*, vol. 1, no. 2, pp. 137–156, Apr. 2007.

[30] G. J. Mendis, J. Wei, and A. Madanayake, "Deep learning-based automated modulation classification for cognitive radio," in *Proc. IEEE ICCS*, Shenzhen, China, 2016.

[31] X. Hong, J. Wang, C.-X. Wang, and J. Shi, "Cognitive radio in 5G: A perspective on energy-spectral efficiency trade-off," *IEEE Commun. Mag.*, vol. 52, no. 7, pp. 46–53, Jul. 2014.

[32] C. Rudolf, "SQL, noSQL or newSQL—Comparison and applicability for Smart Spaces," in *Proc. Seminars FI/IITM WS*, Munich, Germany, 2017, pp. 39–46.

[33] J. Emeras, S. Varrette, V. Plugaru, and P. Bouvry, "Amazon elastic compute cloud (EC2) versus in-house HPC platform: A cost analysis," *IEEE Trans. Cloud Comput.*, vol. 7, no. 2, pp. 456–468, Apr.-Jun. 2019.

[34] S. Rajendran *et al.*, "Electrosense: Open and big spectrum data," *IEEE Commun. Mag.*, vol. 56, no. 1, pp. 210–217, Jan. 2018.
[35] A. De Mauro, M. Greco, and M. Grimaldi, "A formal definition of big data based on its essential features," *Library Rev.*, vol. 65, pp. 122–135, Mar. 2016.
[36] (2018). *Timescaledb Docs|Faq Clustered*. [Online]. Available: https://docs.timescale.com/v0.12/faq#clustered
[37] J. Kepner *et al.*, "Associative array model of SQL, NoSQL, and NewSQL databases," in *Proc. IEEE HPEC*, Waltham, MA, USA, Sep. 2016, pp. 1–9.
[38] (2018). *Update on Influxdb Clustering, High-Availability and Monetization*. [Online]. Available: https://www.influxdata.com/blog/update-on-influxdb-clustering-high-availability-and-monetization/
[39] (2018). *Riak TS Data Types*. [Online]. Available: http://docs.basho.com/riak/ts/1.2.0/learn-about/tablearchitecture/#fields
[40] (2018). *Blob Type|CQL for Cassandra*. [Online]. Available: https://docs.datastax.com/en/cql/3.1/cql/cql_reference/blob_r.html
[41] (2018). *Storing Large Objects and Files in Mongodb*. [Online]. Available: http://blog.mongodb.org/post/183689081/storing-large-objects-and-files-in-mongodb
[42] T. W. Wlodarczyk, "Overview of time series storage and processing in a cloud environment," in *Proc. IEEE CloudCom*, Taipei, Taiwan, Dec. 2012, pp. 625–628.
[43] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web," in *Proc. ACM Symp. Theory Comput.*, 1997, pp. 654–663.
[44] P. Welch, "The use of fast Fourier transform for the estimation of power spectra: A method based on time averaging over short, modified periodograms," *IEEE Trans. Audio Electroacoust.*, vol. AU-15, no. 2, pp. 70–73, Jun. 1967.
[45] A. Chakraborty, U. Gupta, and S. R. Das, "Benchmarking resource usage for spectrum sensing on commodity mobile devices," in *Proc. ACM HotWireless*, 2016, pp. 7–11.

Giuseppe Baruffa received the Laurea and Ph.D. degrees in electronic engineering from the University of Perugia in 1996 and 2001, respectively. In 2005, he spent six months with EPFL, Switzerland. Since 2005, he has been an Assistant Professor with the Department of Engineering, University of Perugia. He has authored about 60 scientific papers. His main research interests include digital television broadcasting, joint source/channel coding for video, software defined radio, and hybrid positioning techniques.

Mauro Femminella received the master's and Ph.D. degrees in electronic engineering from the University of Perugia in 1999 and 2003, respectively, where he has been an Assistant Professor with the Department of Engineering since November 2006. He has coauthored about 100 papers in international journals and refereed international conferences. His current research interests focus on molecular communications, big data systems, and architectures and protocols for 5G networks.

Matteo Pergolesi received the master's degree in computer and automation engineering and the Ph.D. degree in industrial and information engineering from the University of Perugia in 2015 and 2019, respectively. His current research interests focus on 5G, big data architectures, and performance benchmarking.

Gianluca Reali received the Ph.D. degree in telecommunications from the University of Perugia, Italy, in 1997, where he has been an Associate Professor with the Department of Engineering since January 2005. From 1997 to 2004, he was a Researcher with the Department of Electronic and Information Engineering, University of Perugia. In 1999, he visited the Computer Science Department, UCLA. His research activities include resource allocation over packet networks, wireless networking, network management, multimedia services, big data management, and nanoscale communications.